whitehorses®
member of the human network

WHITEBOOK

MUnit in de praktijk

Unittesten van flows in Mulesoft

Auteur:
Martijn van de Goor
INTEGRATIESPECIALIST

Inleiding

Mulesoft biedt een complete integratieoplossing waarmee applicaties aan elkaar gekoppeld kunnen worden waardoor data uitgewisseld kan worden. Het biedt oplossingen voor onder andere API Design, Runtime Management en API management.

MuleSoft APIKit is een tool binnen het Anypoint Platform die het mogelijk maakt om op een eenvoudige manier een API-implementatie te maken vanuit een RAML-definitie. Het is een framework met de mogelijkheid Mule-flows te maken die door de ontwikkelaar verder kunnen worden uitgebreid. De resources die in de RAML zijn gedefinieerd worden op de juiste wijze door de APIKit geïmplementeerd.

Tests voor de flows die door APIKit zijn gegenereerd, kunnen vrij eenvoudig worden ontwikkeld met een ander framework van MuleSoft genaamd MUnit. Dit framework biedt uitgebreide mogelijkheden om unit tests te schrijven. In tegenstelling tot JUnit worden de tests in MUnit ook door middel van drag-and-drop gebouwd, net zoals men een normale MuleSoft flow zou bouwen. Het is volledig geïntegreerd in Maven waardoor het mogelijk is om tests automatisch te laten uitvoeren tijdens het bouwproces van Maven.

Er zijn twee manieren om flows te testen die zijn gegenereerd door de APIKit. De eerste manier is door de flows aan te roepen vanuit een Flow Reference. De tweede manier is door de API aan te roepen via de APIKit router. Het voordeel hiervan is dat de flows worden aangeroepen zoals ze ook tijdens runtime zouden worden aangeroepen. Beide methoden worden in dit Whitebook uiteengezet, aangevuld met diverse opties die MUnit biedt.

Het s-machine-api project

Om de werking van MUnit uit te leggen wordt gebruik gemaakt van een voorbeeld-project: s-machine-api (Het project is te vinden in de Bitbucket van Whitehorses: https://bitbucket.org/whitehorsesbv/s-machine-api en is gemaakt in Anypoint Studio 6.5 en Mule 3.9 met MUnit Plugin 1.7.2).

De API is gebaseerd op de volgende RAML-definitie:

```
/machine/{machine_id}:
  post:
    body:
      application/json:
       example: |
        [
          {
            “motor nummer”: “1”,
            “meetwaarde motor” : 1533
          },
          {
            “motor nummer”: “2”,
            “meetwaarde motor” : 2612
          },
          {
            “motor nummer”: “3”,
            “meetwaarde motor” : 1211
          }
        ]
    responses:
      200:
        body:
          application/json:
            example: |
              {
                “bericht”: “motoren met de juiste meetwaarden zijn
opgeslagen voor machine 16”
              }
      500:
        body:
          application/json:
            example: |
              {
```

```
                "bericht": "motoren met de juiste meetwaarden zijn
opgeslagen voor machine 16. Er waren/was 1 motor(en) met een foute
meetwaarde."
              }
```

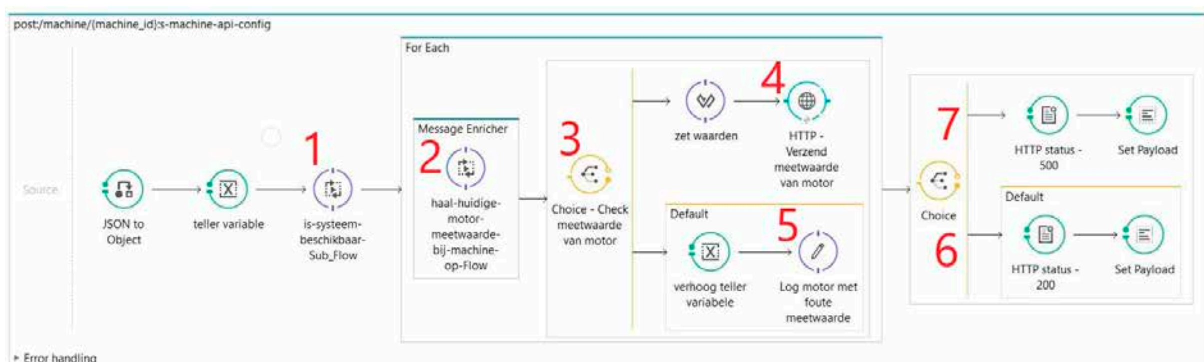
De resource POST maakt het mogelijk gegevens met betrekking tot motorstanden van een machine te posten, bijvoorbeeld met de volgende URL:

```
http://localhost:8081/api/machine/16
```

```
en de volgende body in het bericht:
```

```
[
  {
      "motor nummer": "1",
      "meetwaarde motor": 1233
  },
  {
      "motor nummer": "2",
      "meetwaarde motor": 877
  },
  {
      "motor nummer": "3",
      "meetwaarde motor": 1899
  }
]
```

De RAML definieert tevens wat de response zal zijn wanneer de API een status 200 en een status 500 teruggeeft. Op basis van deze RAML is er een flow gebouwd:



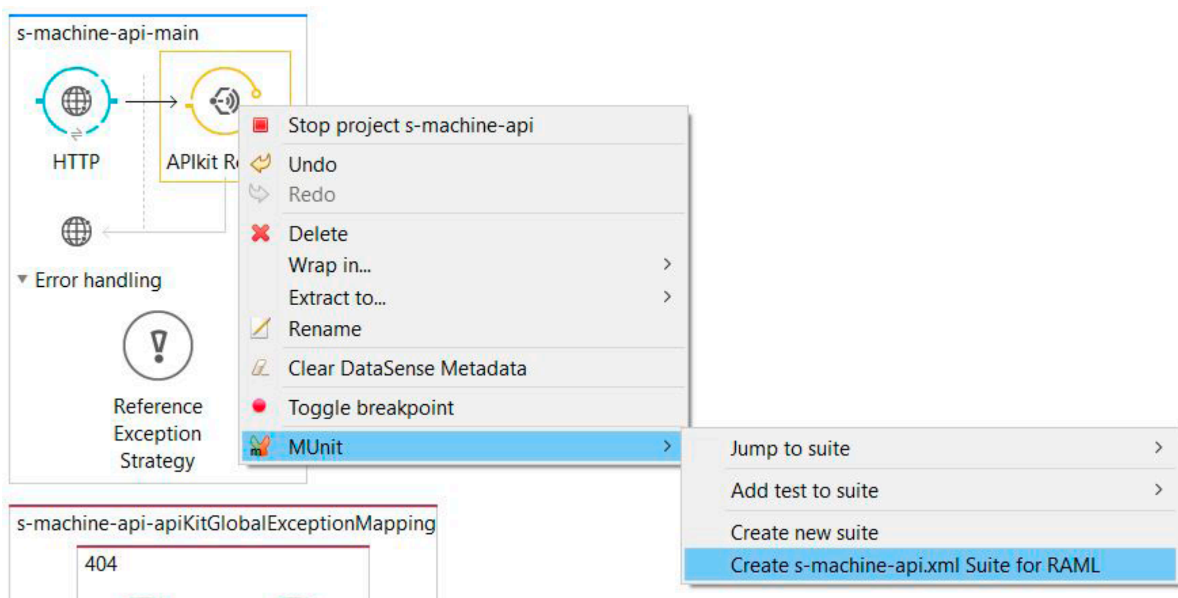
De stappen in de flow zijn als volgt:

1. Is het systeem waar de gegevens naar toe gaan beschikbaar? Dit wordt in een subflow afgehandeld.
2. Per motor wordt de huidige waarde opgehaald. Ten behoeve van deze voorbeeld-API wordt er altijd de waarde 1000 teruggegeven.
3. Er wordt gecontroleerd of de huidige waarde kleiner is dan de nieuwe waarde die de API binnen krijgt.
4. Is de nieuwe waarde correct dan wordt een andere flow aangeroepen via een POST-request om de motorwaarde op te slaan.
5. Is de nieuwe waarde niet juist, dan wordt een teller opgehoogd die bij houdt hoeveel motoren een foute meterstand hebben doorgekregen.
6. Zijn alle nieuwe motorstanden groter dan de huidige motorstanden, dan geeft de API een response met de status 200.
7. Wanneer er motorstanden zijn die niet juist zijn, dan geeft de API een response met de status 500.

In het volgende hoofdstuk wordt een Test Suite gemaakt om de flow te testen.

Het maken van een Test Suite

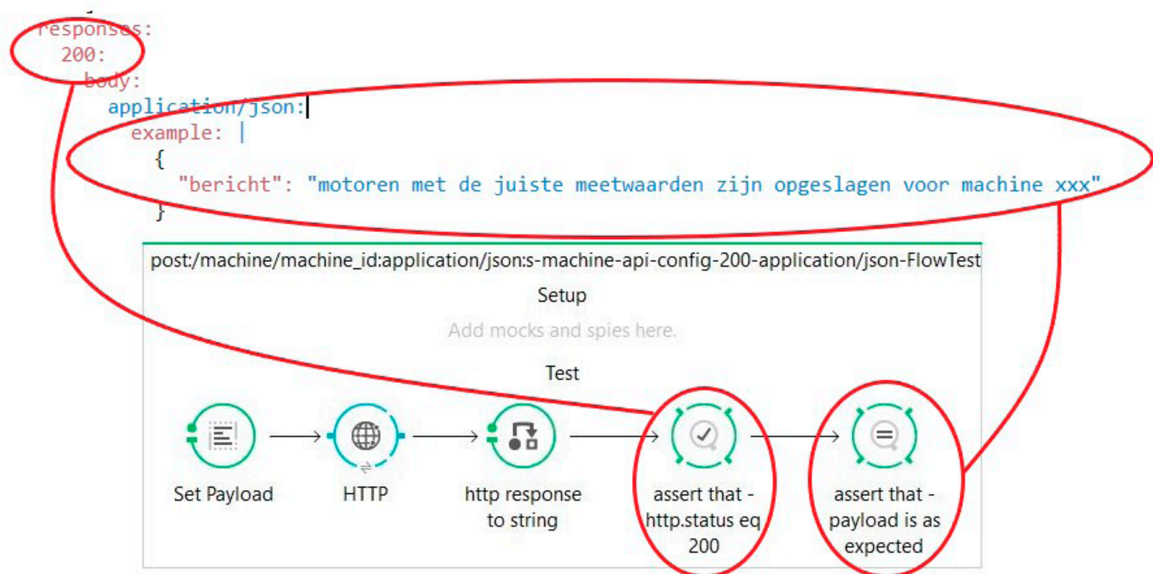
Het maken van een Test Suite is vrij eenvoudig. Rechtsklik op de APIKit Router in Anypoint Studio en selecteer *Create s-machine-api.xml Suite for RAML*:



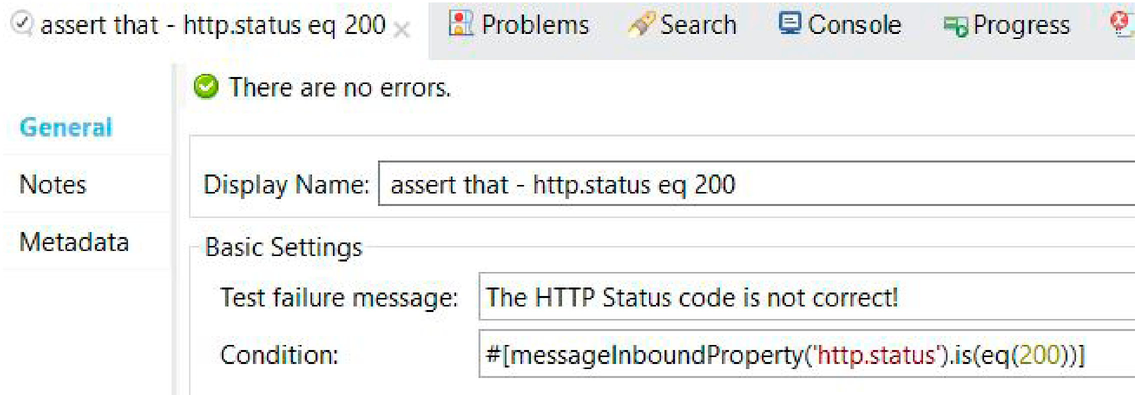
Er wordt nu een Test Suite gegenereerd met unit tests voor elke resource/actie/response combinatie die staat gedefinieerd in de RAML. In ons voorbeeld is er 1 resource, te weten / machine/{machine_id}, met 1 POST-actie en 2 responses, 200 en 500. Er worden dan twee unit tests gegenereerd (machine/post/200 en machine/post/500).

Assertions

Bij het genereren van de testgevallen worden er ook meteen een aantal assertions gegeneerd binnen elke flow. Hierin wordt de geretourneerde HTTP status en de geretourneerde payload gevalideerd.



De status die de API teruggeeft wordt vergeleken met een verwachte response status. In bovenstaand geval wordt dus gekeken of de status die de HTTP teruggeeft daadwerkelijk 200 is:



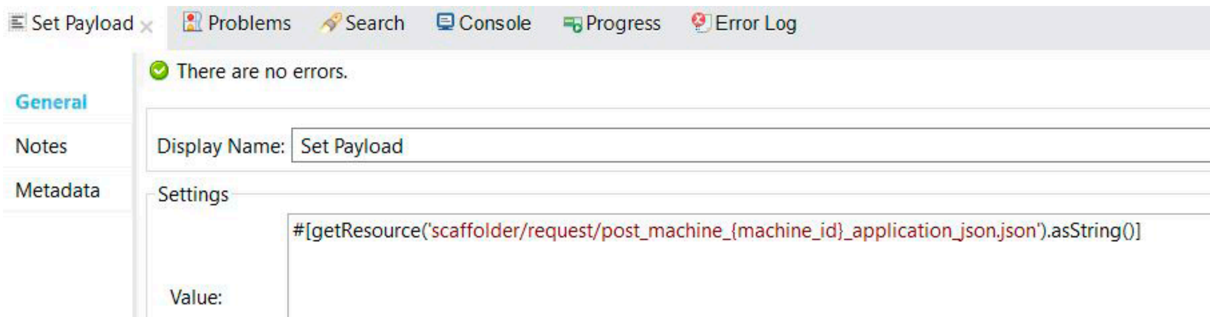
Hiervoor dienen het request en de data die geretourneerd wordt door externe services een status 200 tot gevolg te hebben. Request- en response-berichten worden automatisch gegenereerd bij het aanmaken van een Test Suite met gebruikmaking van de RAML. Ze worden geplaatst in de map src/test/resources binnen het project:

```
src/test/munit
    s-machine-api-apikit-test.xml
src/test/resources
    log4j2-test.xml
    scaffolder.request
        { } post_machine_{machine_id}_application_json.json
        { } post_machine_{machine_id}_motor_{motor_number}_application_json.json
    scaffolder.response
        { } post_200_machine_{machine_id}_application_json.json
        { } post_500_machine_{machine_id}_application_json.json
```

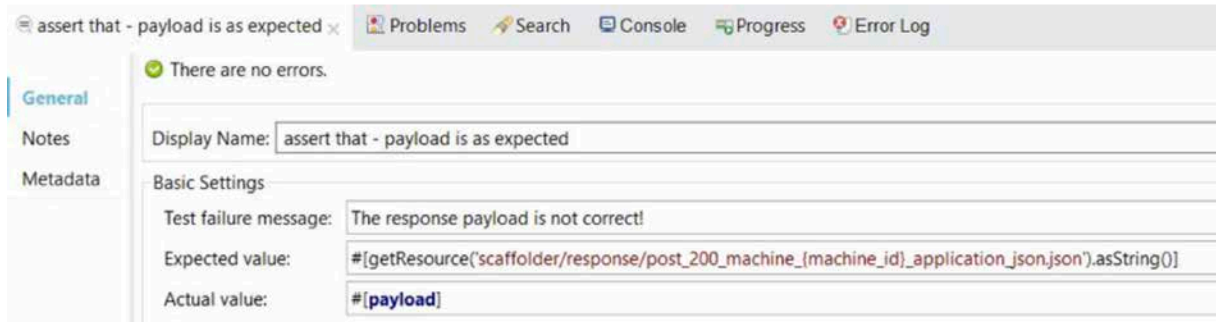
Het request bestand *post_machine_{machine_id}_application_json.json* ziet er als volgt uit en is dus rechtstreeks uit de RAML overgenomen:


```
[
  {
    "motor nummer": "1",
    "meetwaarde motor" : 1533
  },
  {
    "motor nummer": "2",
    "meetwaarde motor" : 2612
  },
  {
    "motor nummer": "3",
    "meetwaarde motor" : 1211
  }
]
```


Aangezien alle meetwaarden boven de 1000 zijn zal dit een response met status 200 geven. In de *Set Payload* stap van de test wordt het request-bericht gezet. Deze stap is ook automatisch gegenereerd:



Naast een assertion op de status wordt er ook automatisch een assertion op de payload aangemaakt. Deze vergelijkt de daadwerkelijke output die terugkomt van de API met een verwachte output. De assertion-stap op de payload ziet er als volgt uit:



Wanneer je de unit test gaat runnen zal dit niet goed gaan (Zie de laatste paragraaf hoe je een Test Suite kunt runnen). De volgende foutmelding wordt gegooid:

```
Message               : The response payload is not correct! expected:<{
"bericht": "motoren met de juiste meetwaarden zijn opgeslagen voor machine
16"  }> but was:<{   "bericht": "motoren met de juiste meetwaarden zijn
opgeslagen voor machine {machine_id}"  }> (java.lang.AssertionError).
```

Het gaat fout omdat de flow geen waarde binnenkrijgt voor de variabele machine_id. Dit is de uri-parameter van de flow die we aan het testen zijn. De aanroep van de API binnen de unit test gaat via een HTTP-connector. In deze HTTP-connector staat de uri waarmee de flow wordt aangeroepen. Hier moet deze variabele dus nog gezet worden:

Display Name: HTTP
General Settings
Connector Configuration: HTTP_Request_Configuration
URL Settings
Path: /machine/{machine_id}
Method: POST
Parameters
Click in the button below to add a parameter
header Name: Accept Value: application/json
header Name: Content-Type Value: application/json
uri-param Name: machine_id Value: 16
Add Parameter

De test zal nu goed verlopen. Van belang is hierbij op te merken dat de test op dit moment nog geen echte unit test is aangezien de externe services in de flow nog daadwerkelijk worden aangeroepen en niet gemockt worden. Dit kan problemen opleveren bij het testen: externe services zijn bijvoorbeeld niet altijd bereikbaar vanaf elke machine. Denk hierbij aan locale (ontwikkel) machines, maar ook buildservers zoals Jenkins, Hudson en Bamboo. Wanneer deze services vanaf die betreffende machine niet beschikbaar zijn, zal de flow dus altijd een error terugkrijgen van de service, en zal de unit test dus ook nooit succesvol kunnen lopen.

Mocking

MUnit beschikt over mogelijkheden tot het mocken van request- en response-berichten. Hierdoor wordt de test geïsoleerd van externe systemen waardoor veranderingen hierin ook geen impact hebben. Dit wordt bereikt door mocks te maken voor endpoints of connectors die gedefinieerd zijn binnen de flow.

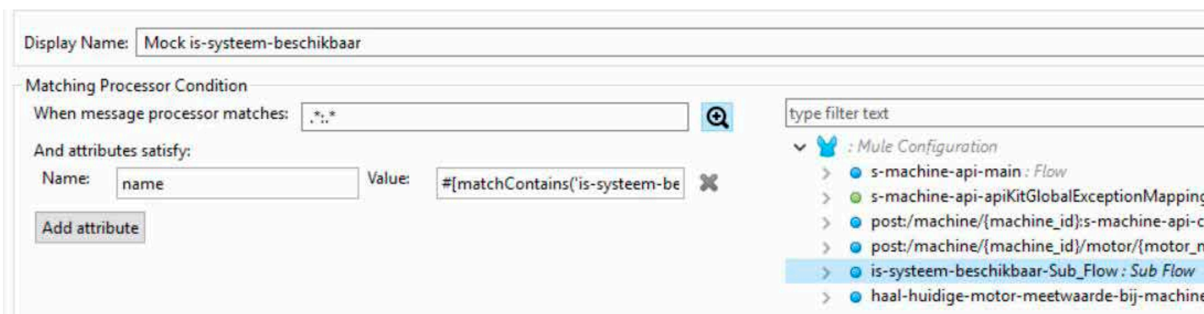
Subflows en private flows mocken

Men kan een mock aan de unit test toevoegen door een mock component naar de test flow te slepen. Vervolgens kan men een message processor kiezen die gemockt moet worden. Dit kan alleen wanneer de betreffende flow die gemockt moet worden zich binnen het huidige project bevindt.

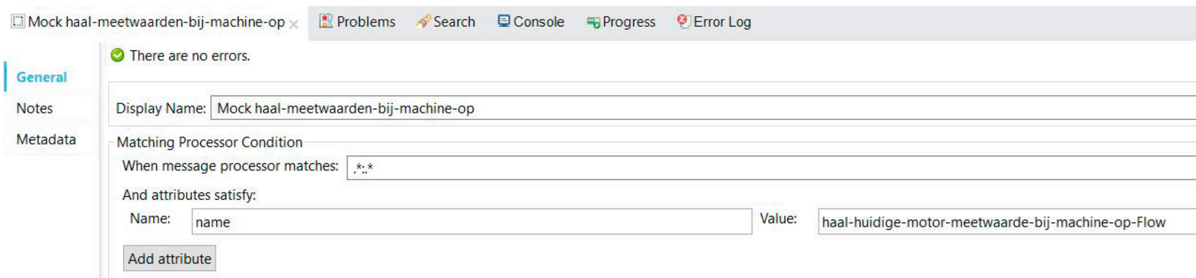
In ons voorbeeldproject wordt een subflow aangeroepen die teruggeeft of een systeem beschikbaar is. Deze kan men kiezen wanneer men op het vergrootglas in de Matching Processor Condition klikt. Het attribuut dat aangeeft wanneer de mock moet worden gebruikt wordt nu automatisch gevuld met de waarde:

```
#[matchContains('is-systeem-beschikbaar-Sub_Flow')]
```

Opvallend is dat bij het mocken van subflows er een MEL-expressie wordt gebruikt met de functie matchContains:



Bij het mocken van private flows is dit niet het geval. In onze voorbeeld-API wordt ook een private flow aangeroepen waarbij de meetwaarden van de motoren in een machine worden opgehaald. Wanneer we daarvoor een mock toevoegen ziet dat er als volgt uit:



Zoals te zien wordt de waarde van de naam van de private flow in tegensteling tot een subflow zonder MEL-expressie neergezet.

De payload die gezet moet worden, moet nu worden toegevoegd voor beide mocks. Dit kan voor de private flow met de volgende MEL-expressie:

```
#[getResource('scaffolder/response/haalOpmeetwaardeBijMachine.json').
asString()]
```

“Externe” endpoints mocken

Voor dit voorbeeld wordt gebruik gemaakt van de al eerdergenoemde stap 4 waarin via een HTTP POST-request een meterwaarde wordt opgeslagen. Voor het gemak is deze flow ook in ons voorbeeldproject gezet, maar voor dit voorbeeld doen we alsof dit niet het geval is. Wanneer een endpoint dat gemockt moet worden niet in het hetzelfde project zit kan het ook niet gekozen worden met het al eerdergenoemde vergrootglas. De *when message processor matches* optie in de mock moet dan handmatig ingevuld worden evenals een attribuut dat moet worden ingevuld. Standaard wordt de functie op wildcards ingesteld zoals we in de vorige paragraaf konden zien. Soms willen we echter heel specifiek een stap binnen een flow benoemen. We halen deze gegevens uit de Configuration XML van de flow:

```
<http:request config-ref="HTTP_Request_Configuration_post_motor"
                   path="/api/machine/{machine_id}/motor/{motor_number}"
method="POST"
       doc:name="HTTP - Verzend meetwaarde van motor">
<http:request-builder>
          <http:uri-param paramName="machine_id" value="#[flowVars.machine_
id]"/>
          <http:uri-param paramName="motor_number" value="#[flowVars.motor_
number]"/>
  </http:request-builder>
</http:request>
```

De *when message processor matches* optie krijgt de waarde *http:request*. Een nieuw toegevoegd attribuut krijgt bij *Name* de waarde *doc:name* en als *Value* de waarde *HTTP – Verzend meetwaarde* van motor. Hierdoor kan dus heel specifiek een bepaald type van een flow-stap worden gemockt. In dit geval wordt dus een HTTP-request gemockt. Het is ook mogelijk om bijvoorbeeld een Database te mocken. In dat geval gebruik je *db:insert* voor het mocken van database-inserts of *db:update* voor het mocken van updates op de database.

De payload in de mock moet natuurlijk ook gezet worden. Dit gaat op dezelfde manier als bij een private flow.

De Mock ziet er dan als volgt uit:

Display Name: Mock verzend-meetwaarde-van-motor

Matching Processor Condition

When message processor matches: http:request

And attributes satisfy:

Name: doc:name Value: HTTP - Verzend meetwaarde van motor

Add attribute

Then return message with payload: #[getResource('scaffolder/response/verzendMeetwaardeVanMotor.json').asString()]

Encoding:

MIME Type:

Properties

No properties

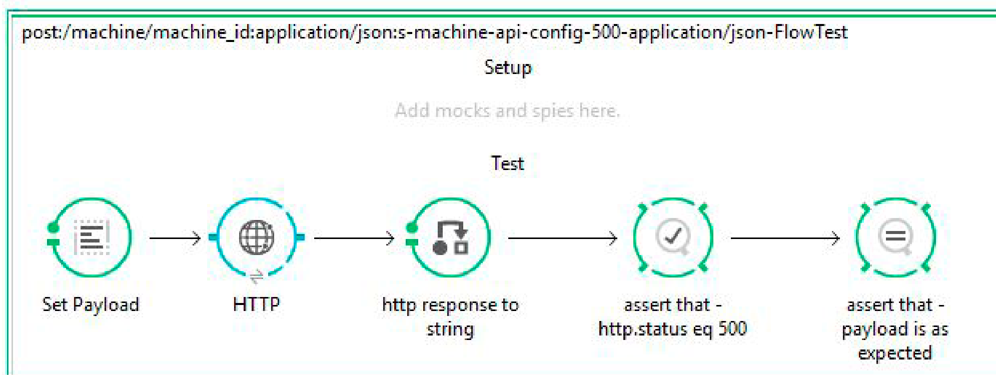
Add property

Een tekortkoming in de standaard MUnit mock functionaliteit is dat je slechts één mocked response in kunt stellen. Wanneer een service dus vaker wordt aangeroepen (bijv. in een for-each loop) waarbij je wilt simuleren dat één call goed gaat, en één call een (functionele) fout teruggeeft, is dit niet mogelijk via de standaard MUnit mock functionaliteit.

Spy Event Processor

De Spy Processor in MUnit maakt het mogelijk te bekijken wat er gebeurt in de stap voor en de stap na de aanroep van een event processor zoals bijvoorbeeld een HTTP-request. Je kunt dan een aantal instructies uitvoeren voor en/of nadat de event processor is uitgevoerd. Dit kunnen bijvoorbeeld assertions zijn, maar je zou ook bijvoorbeeld de payload van een request kunnen opslaan zodat je die later in de unit test kunt bekijken. Dit gaan we in het volgende voorbeeld doen.

In ons voorbeeldproject worden in stap 4 alleen de juiste meterwaarden van een motor doorgestuurd via een service en de foute meterwaarden niet. We gaan in de tweede gegenereerde unit test (machine/post/500) een test maken waarbij drie meterwaarden worden verwerkt waarvan er 1 niet correct is:



We moeten weer een mock-request maken net zoals bij de eerste unit test.
Het request-bestand *post_500_machine_{machine_id}_application_json.json* moeten we zelf aanmaken en is niet gegenereerd toen de Test Suite werd aangemaakt. Het ziet er als volgt uit:

```
[
  {
    "motor nummer": "1",
    "meetwaarde motor" : 1533
  },
  {
    "motor nummer": "2",
    "meetwaarde motor" : 899
  },
  {
    "motor nummer": "3",
    "meetwaarde motor" : 1211
  }
]
```

Aangezien de tweede meetwaarde onder de 1000 is zal dit een response met status 500 geven.
In de *Set Payload* stap van de unit test moeten we de locatie van dit request-bericht nog goed zetten.

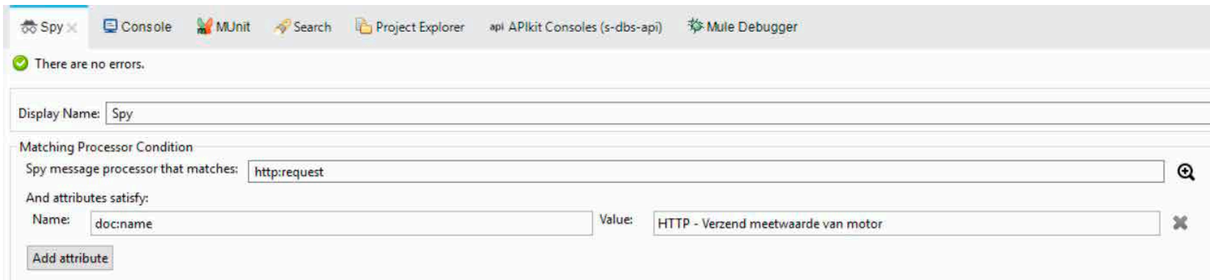
We verwachten dat de service die de meterwaarden verzendt (stap 4) maar twee keer wordt aangeroepen en dat dit gebeurt met twee verschillende payloads. De eerste call gaat namelijk over motor 1 en de tweede over motor 3. Binnen de Spy is het alleen maar mogelijk om één assertion op te nemen en is het niet mogelijk om twee verschillende payloads te checken. We zullen dus voor het begin van de aanroep van onze te testen service de verwachte uitkomst moeten vastleggen om die na afloop te vergelijken met de requests die daadwerkelijk zijn gepost. Dit doen we in een *flowVar* genaamd *verwachte_requests*. Tevens moeten we een *flowVar* aanmaken met een lege array waarin we de requests die in de flow worden gepost gaan opslaan. Dit is de *flowVar* genaamd *daadwerkelijke_requests*. Deze flowVars worden gezet voordat de te testen service wordt aangeroepen. Helaas worden ze niet doorgegeven door de HTTP-connector heen. Daarom kunnen we in deze unit test geen gebruik maken van een aanroep via de HTTP-connector maar zullen we een Flow Reference moeten gebruiken naar *post:/machine/{machine_id}:s-machine-api-config*.

Dit betekent ook dat we URI-parameter op een andere manier moeten zetten dan in de HTTP-connector. Dit gebeurt nu ook via een *flowVar* omdat URI-parameters standaard om worden gezet naar flowVars in een Mule flow:

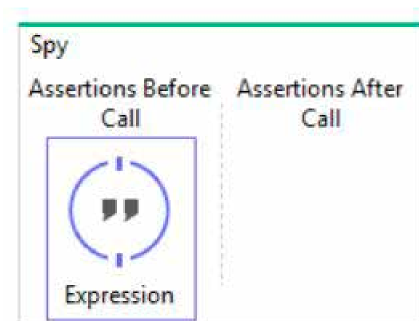
Output FlowVar - machine_id

```
1 %dw 1.0
2 %output application/java
3 ---
4 '16'
```

Nu kan de *Spy* toegevoegd worden aan de unit test. De *spy* dient natuurlijk te weten welke processor geïnspecteerd moet worden. Dit kan op eenzelfde manier aangeven worden als in een mock:



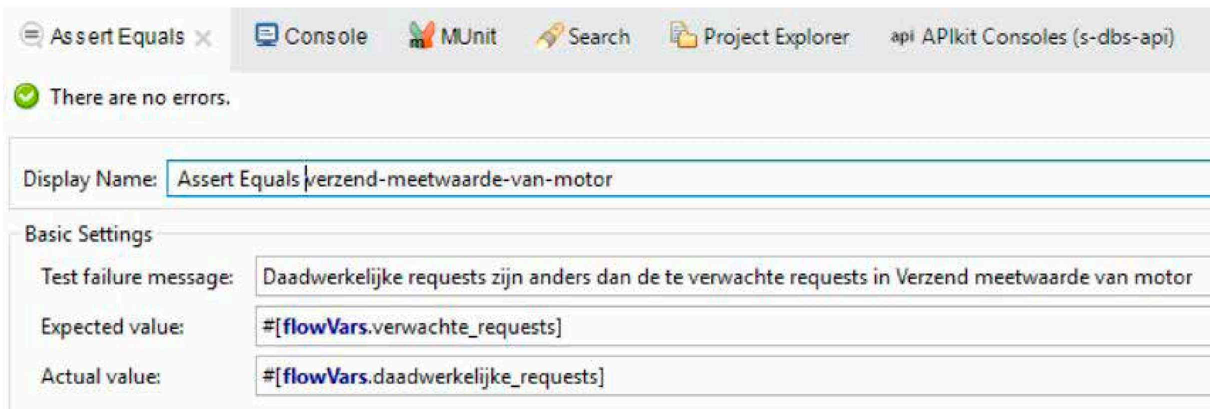
In de *Spy* gaan we een expressie toevoegen die iedere request opslaat in de eerder aangemaakte *daadwerkelijke_requests flowVar*. De *Spy* inclusief expressie ziet er dan als volgt uit:



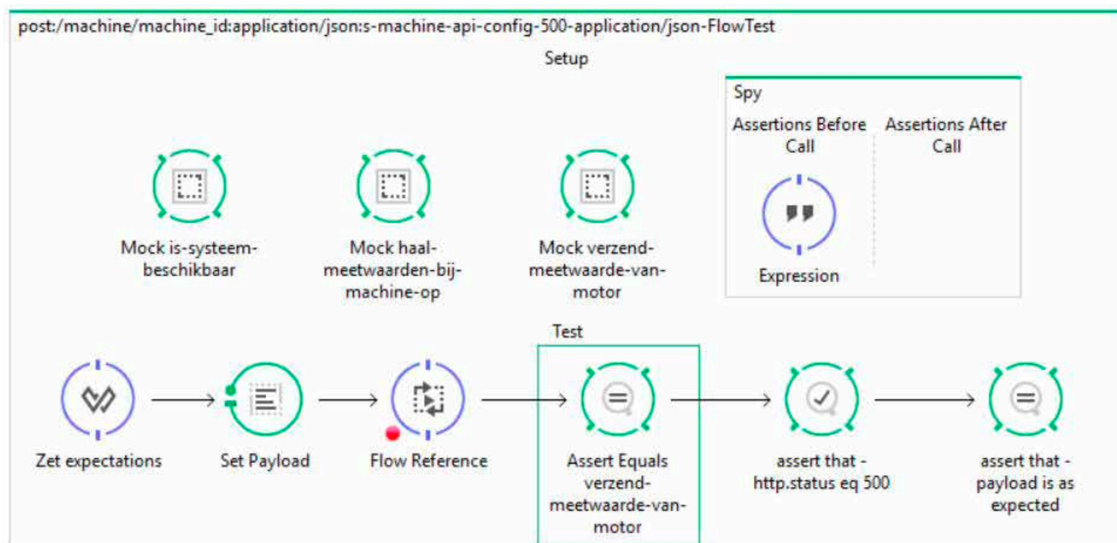
We maken hier gebruik van message.payloadAs(java.lang.String) en niet direct van de Payload om de waarde toe te voegen aan de *verwachte_requests* variabele. Een application/json object wordt namelijk geretourneerd als een stream. Een response in plain/text of application/java kan dus wel direct als Payload worden toegevoegd. Je zou kunnen denken dat het omzetten van de responsen via bijvoorbeeld een *Transform Message of JSON to Object* binnen de Spy een oplossing voor dit probleem zou kunnen zijn. Helaas is dit niet het geval. De Spy werkt dan in zijn geheel niet meer.

Een tweede probleem bij het gebruik maken van message.payloadAs(java.lang.String) wordt gevormd door de indents die in een JSON of XML bericht zitten. Wanneer we de te verwachten requests gaan vergelijken met de daadwerkelijke requests moet elke spatie binnen de berichten gelijk zijn omdat we data als String aan het vergelijken zijn. Hierdoor kost het maken van een te verwachten request bericht behoorlijk wat tijd.

Na het zetten van de Spy moet nu nog de assertion toegevoegd worden die de te verwachten requests vergelijkt met de request die daadwerkelijk zijn gepost. Daartoe slepen we een Assert Equals in de unit test op de plaats naast de Flow Reference en zetten we de gegevens die vergeleken moeten worden en optioneel de melding die wordt gegeven wanneer de test faalt:



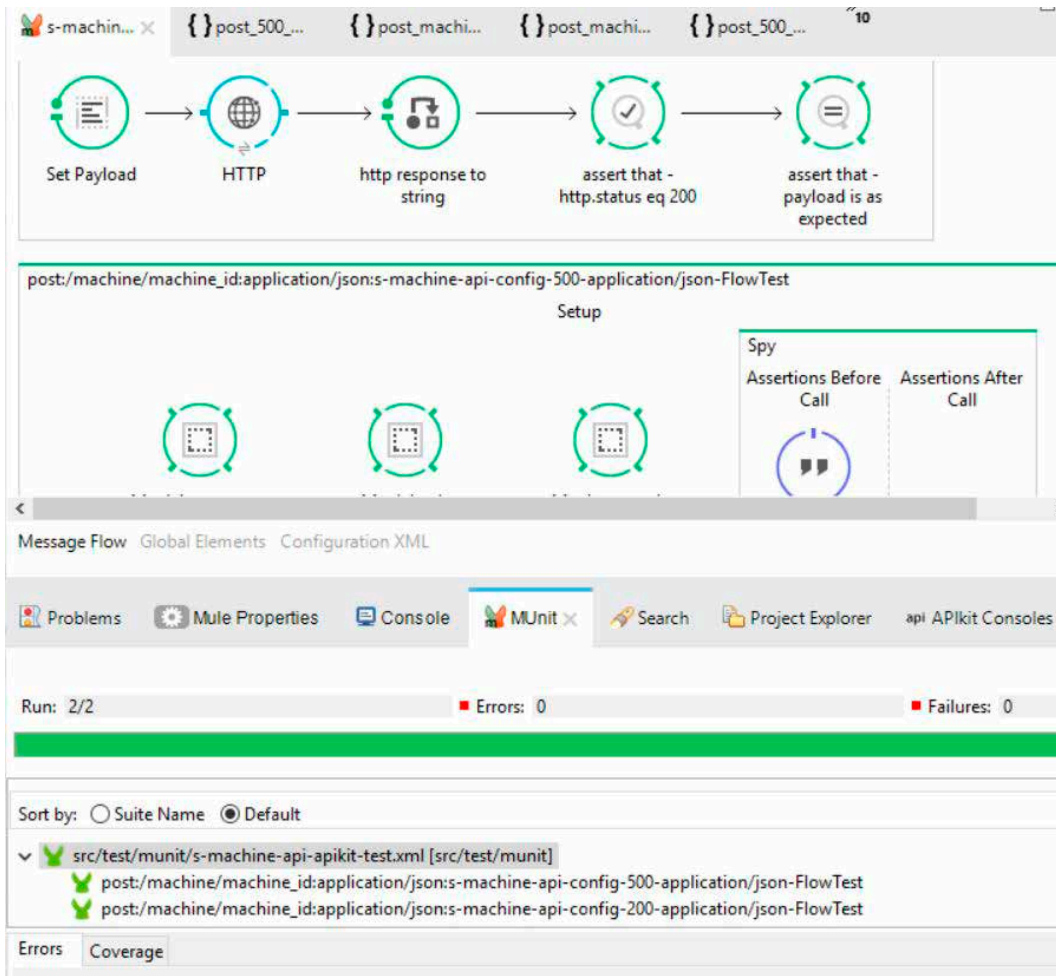
De unittest ziet er dan als volgt uit:



We kunnen nu bijna de Unit test gaan draaien. Een paar dingen moeten nog worden toegevoegd. De verschillende Mocks uit de eerste unit test zijn gekopieerd in deze unit test. Tevens moet de assert op status 500 worden aangepast omdat we geen HTTP-connector meer gebruiken maar een Flow Reference. De expressie `#[messageInboundProperty('http.status').is(eq(500))]` dient vervangen te worden door `#[message.outboundProperties['http.status']==500]`.

Het draaien van een Test Suite is heel eenvoudig. Men rechter-klikt op het MUnit-scherm en kiest Run MUnit Suite. Het is tevens mogelijk een los testgeval te draaien. Hiertoe selecteert men eerst het betreffende testgeval en kiest Run MUnit test. Het draaien van een Test Suite kan ook via Maven worden uitgevoerd via de standaard test *phase* (mvn test).

In het tabblad *MUnit* wordt het resultaat van de tests getoond:



Naast het overzicht van welke tests succesvol gelopen hebben, heb je ook nog het *Coverage* tabblad. Hierin zie je de dekkingsgraad van de tests en is een hulpmiddel om ervoor te zorgen dat de unit tests zo dekkend mogelijk worden:

Errors | Coverage

Check covered message processors

Generate Report

Overall coverage:61,11

s-machine-api.xml
- s-machine-api-apiKitGlobalExceptionMapping(,00%)
- post:/machine/{machine_id}:s-machine-api-config(100,00%)
- haal-huidige-motor-meetwaarde-bij-machine-op-Flow(,00%)
- s-machine-api-main(100,00%)
- is-systeem-beschikbaar-Sub_Flow(75,00%)
- post:/machine/{machine_id}/motor/{motor_number}:s-machine-api-config(,00%)

Conclusie

In dit Whitebook is uitgelegd hoe flows getest kunnen worden met het MuleSoft framework MUnit. Een aantal opties zoals mocks, assertions en spies zijn behandeld. Helaas is de ruimte van een Whitebook te beperkt om alle mogelijkheden van MUnit te bespreken.

Concluderend kunnen we stellen dat MUnit als tool zeker nuttig is. Er zijn echter nog wel een aantal zaken niet helemaal goed uitgewerkt en ontbreekt er functionaliteit zoals we hebben kunnen zien:

- Conditionele mocked responses: meerdere verschillende resultaten van een service binnen een flow kunnen mocken. Hiermee kan bijvoorbeeld zowel een goedsituatie als een foutsituatie binnen een for-each getest worden.
- Meerdere assertions binnen een Spy kunnen vastleggen. Hierdoor moet men gebruik maken van een mechanisme om requests op te slaan om ze later te vergelijken met een variabele met daarin een reeks verwachte requests. Dit levert problemen op met JSON en XML berichten die moeilijk te vergelijken zijn.

* Voor het gebruik van MUnit is een extra plugin nodig voor Anypoint Studio, de MUnit Anypoint Studio Plugin.

** De naamgeving, beginnend met de letter s, volgt het system-process-experience design principe van MuleSoft waarin wordt gesteld dat een API-systeemarchitectuur uit deze drie lagen bestaat. In ons voorbeeld is er sprake van een API voor een systeem laag die direct met onderliggende systemen communiceert, zoals bijvoorbeeld een Database.

*** Het verschil tussen een ”normale” flow, een subflow en een private flow wordt hier uitgelegd: https://forums.mulesoft.com/questions/57742/what-is-the-difference-between-flow-sub-flow-and-p.html