

API Testing with BDD Framework

Hands-on Exercises

Exercises Overview	0
Objectives	1
Reading Materials	1
Resources	1
Basics	1
Test your exposed application API's	2
Exercise #1 - Creating testing application for exposed API (expected duration 30m)	2
Test CompanyCreateOrUpdate API	5
Exercise #2 - Testing Parameters, Output, Interoperability and Errors (expected duration 4h 30m)	5
#2.1 - Setting up our test scenario base structure	8
#2.2 - Updating our API under test	9
#2.3 - Building our test logic	11
#2.4 - Setup & Teardown steps	13
#2.5 - Error Handling and Status Codes	15
Summary	21
Exercise #3 - Security (expected duration 1h 30m)	21
#3.1 - Create Demo Auth API	21
#3.2 - Test the Security of the API	22
Summary	27
Exercise #4 - Data (expected duration 1h 30m)	27
Exercise #4.1 - Check Data if data persisted (expected duration 30m)	27
Exercise #4.2 (Optional) - Following the Testing Pyramid Thinking Model (expected duration 1h)	28
Overall Summary	29

Exercises Overview

There are multiple types of API's you can build with OutSystems. You can build REST or SOAP API's, but you also build service actions, which are actually REST API endpoints that are only available within the platform.

On top of that, applications built within your OutSystems factory can also consume external API's, both REST or SOAP.

Ensuring API's are working as expected and specially complying to their defined contracts is a crucial part of any testing strategy. To accelerate the way you and your teams can perform these kinds of tests, you can build automation.

In the application portfolio provided, the CRM Services application already exposes service actions and API endpoints, which we will be building automated tests for during our next exercises.

Objectives

To provide exercises that demonstrate how to test our own APIs, using the BDD Framework to build automated tests for APIs and Service actions, to address some common anti-patterns for testability and how we can use a data driven approach.

Note: to use this approach to test external APIs make sure that setup and teardown methods are available otherwise the tests might not have the correct results.

Reading Materials

- [API Testing \(Becoming a tester in OutSystems - Guided path\)](#)
- [Throw a Custom Error in an Exposed REST API](#)
- [How to Handle HTTP Status Codes When Consuming a REST API in OutSystems](#)

Resources

- [BDD Framework](#)

Basics

Our goal is to test the API's following the POISED strategy:

- **Parameters**
- **Output**
- **Interoperability**
- **Security**
- **Errors**
- **Data**

To recap, while testing Outsystems API's we should focus on the following:

- **Parameters**
 - Mandatory/optional inputs
- **Output**
 - API Response
- **Interoperability**
 - Input and Output formats according to the specification
- **Security**
 - Authorization
- **Errors**
 - Error message meaningfulness and status code
- **Data**
 - Is data persisted as expected

- Verify functional and non-functional Requirements

So, for these exercises we will be using the BDD Framework.

Test your exposed application API's

When testing your exposed APIs you have the advantage of having access to reference your actions and database to enhance your tests.

Exercise #1 - Creating testing application for exposed API (expected duration 30m)

In this exercise, the main goal is to prepare our test application for using POISED strategy.

First, let's capture our API test objects that we'll be using for this exercise

1. Open service center for your personal area and get the documentation swagger file URL for the API exposed by the module Customers, from CRM Services App:

The screenshot shows the OutSystems development console for the 'Module Customers'. The 'Integrations' tab is active, displaying the configuration for exposed REST APIs. The table below shows the details for the 'CRMServicesAPI' entry.

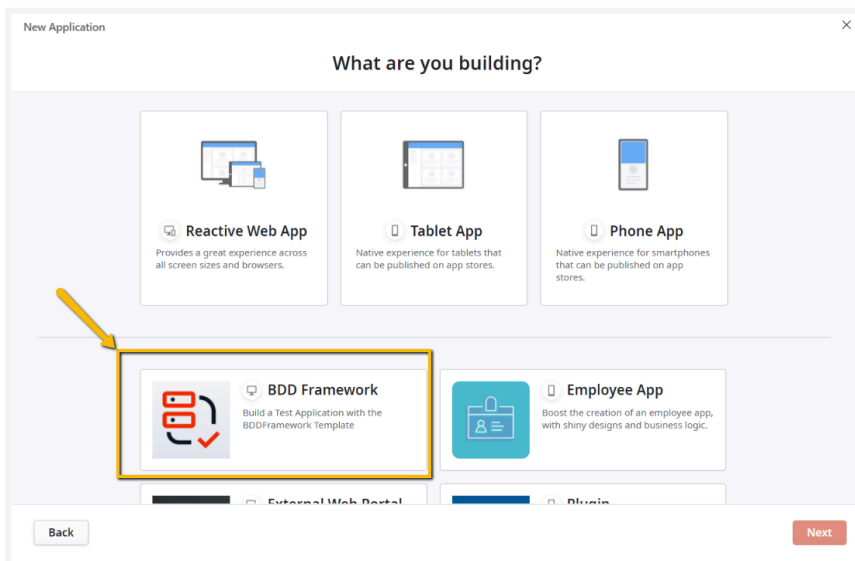
Name	URL	Internal Access	HTTP Security	Documentation	Logging Level
CRMServicesAPI	/Customers/rest/CRMServicesAPI	Internal Access	SSL/TLS	(Open)	Default

CRMServicesAPI

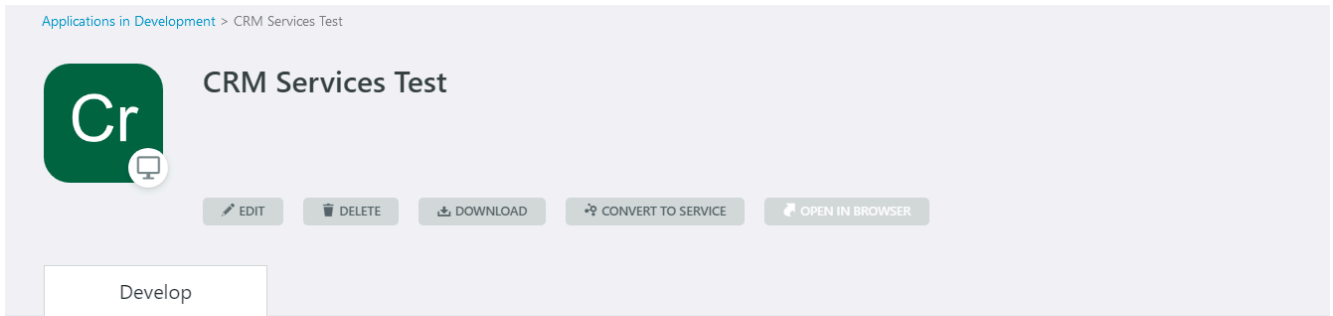
[Base URL: /Customers/rest/CRMServicesAPI]
swagger.json

POST	/CompanyCreate
PUT	/CompanyCreateOrUpdate
DELETE	/CompanyDelete
POST	/ContactCreateHistory
PUT	/ContactCreateOrUpdate
PUT	/ContactCreateOrUpdatePicture
DELETE	/ContactDelete
GET	/GetCompanyById
GET	/GetCompanyByName
GET	/GetContactById
GET	/GetContactByName
GET	/GetCountryByName

2. Now that we have the API's, let's create a new app that will test the app "CRM Services".
3. Open Service Studio and create a new app from scratch and select the BDD Framework as the template and call it "CRM Services Test"

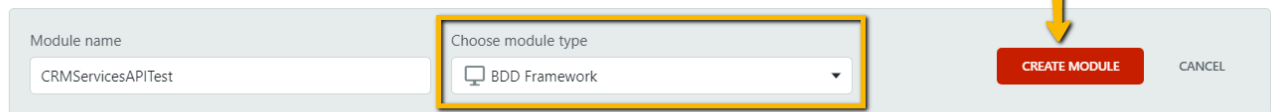


4. After the above step you have to create a module inside that app, and for that you should create it based on the BDD framework module type.

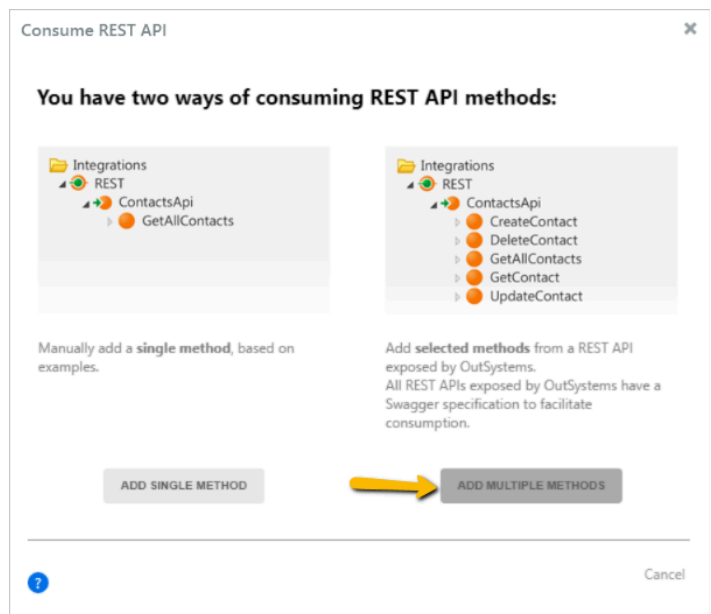
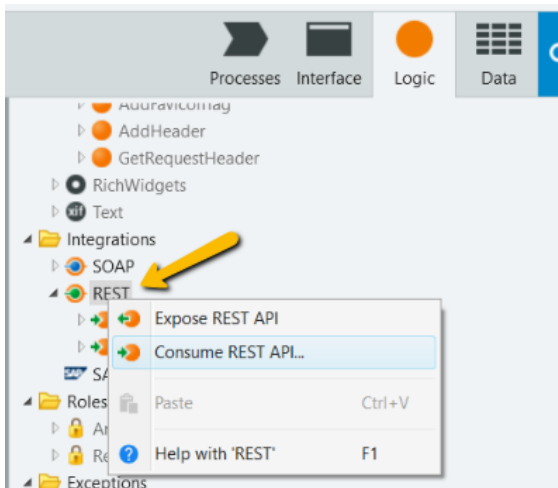


Modules

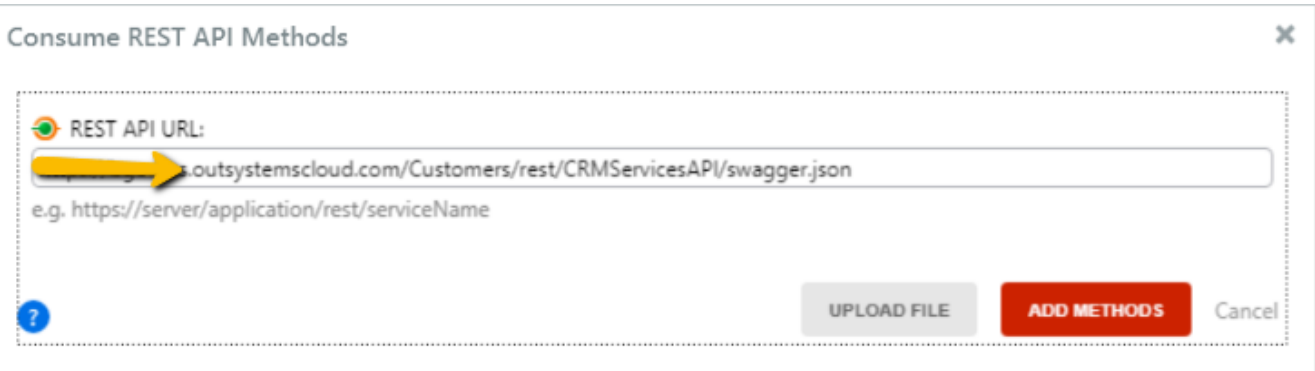
Modules allow you to structure your application into several pieces, each piece implementing a specific purpose.



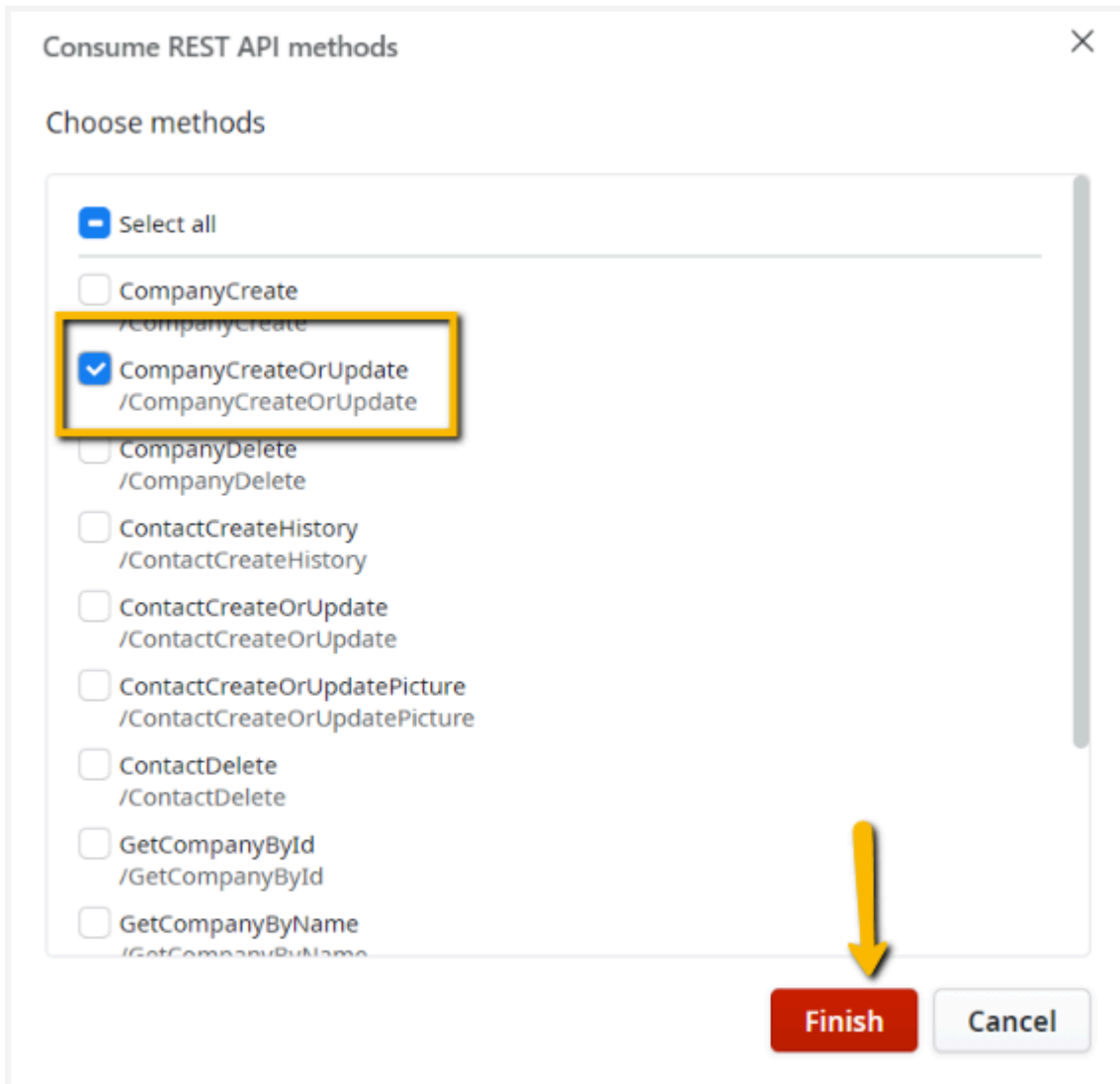
5. Now that we have the application and the module created, we need to consume the swagger url of the CRM Services API module and select only the "CompanyCreateOrUpdate" and "GetCompanyByName" methods.



Now put the swagger url that you have on the API definition documentation:



Click on “Add Methods” and then select the following methods:



Now that we have the methods added it's time to test those APIs.

For that purpose, our next exercises will be focused in testing the **CompanyCreateOrUpdate** API endpoint, applying the POISED strategy.

Test CompanyCreateOrUpdate API

Exercise #2 - Testing Parameters, Output, Interoperability and Errors

(expected duration 4h 30m)

Since these vectors of the POISED testing strategy are very connected we will simulate the testing of all them in this exercise.

Our goal for the whole exercise is to test the following :

- Parameters - Mandatory/optional inputs
- Parameters - Request Headers
- Output - API Response
- Interoperability - Input and Output formats, according to specification
- Errors - Error message meaningfulness and Status code

Based on the previous vectors of the POISED strategy, since they are very related, we are going to use a data driven approach. The idea is to cover all the tests we'll build with a single reusable scenario.

As our strategy for data driven testing, we will use a static entity instead of a spreadsheet. Now you ask...why? The reason is the maintenance of that information. With the spreadsheet as a resource for every single change you end up downloading and uploading the file and publishing, with the static entity it becomes easier this task. Also, this enables you to read and change your test data, directly within the service studio, which is not possible with the spreadsheet approach.

Note: Avoid having static entities with more than 300/400 records. If you have a scenario that covers way more than this number of tests, use a spreadsheet for that particular case.

For the first part of our exercise we will build the following scenario:

Scenario: <ScenarioName>

Given:

There is no company called <CompanyName>

And

I want to create the company with name <CompanyName>, Company Type <CompanyType>, email <Email>, country with code <CountryCode>

When:

I call CompanyCreateOrUpdate endpoint with those attributes

Then:

if(<ExpectedSuccess> = true) -> if condition added inside expression or another step

The response returned a valid Company Id

else

The output message received is <OutputMessage>

DATA STEERING TABLE

Scenario Name	Company Name	Company Type	Email	Country Code	Expect Success	Output Message	Ex. Order
Company Success Creation	Covid Free	Customer	covidfre e@exampl e.com	US	true		0
Mandatory Fields Missing	Covid Free				false	Company type, Email and Country are mandatory	1
Invalid Email Format	Covid Free	Customer	covidfre e.exempl e.com	US	False	The email format is not valid	2

This scenario has some dynamic values <example> and a data steering table that basically represents our test data.

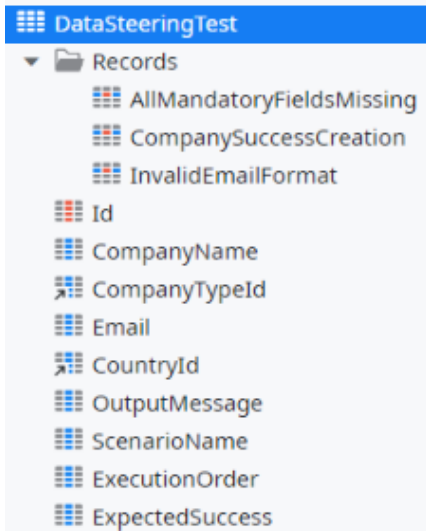
As you can see from the above scenario we will perform a couple of validations, but the API "CompanyCreateOrUpdate" (from the [first exercise](#)) that we are going to use to make the tests, does not have those validations developed yet. So our goal not only is to test the API but also to make sure that is developed properly.

For the sake of the exercise let's assume that you have forgotten to put those validations on the API when you coded it.

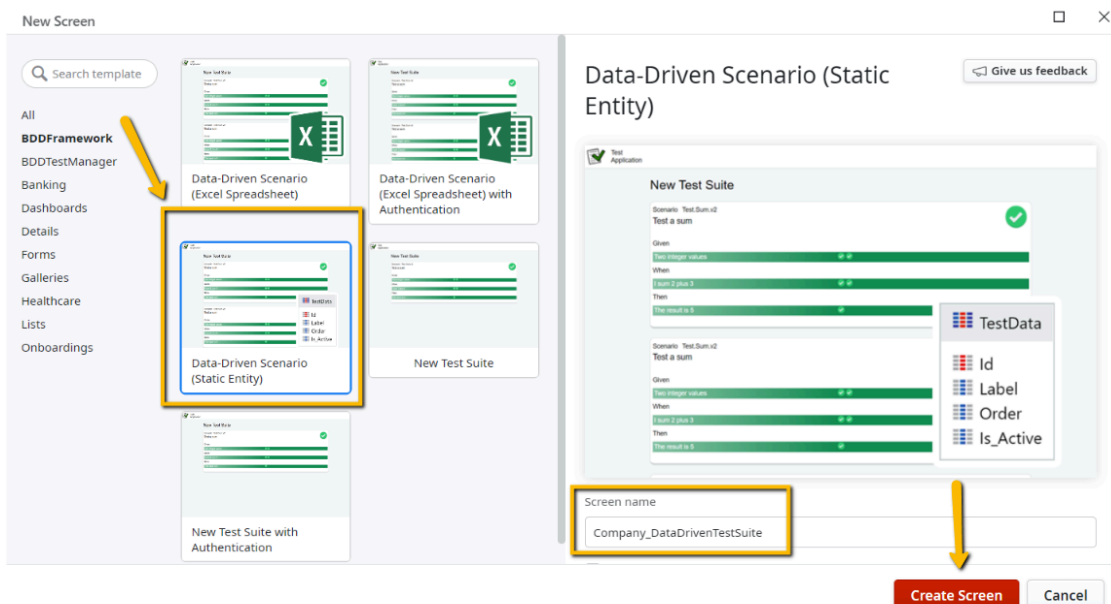
So the first thing that we need to do is to fix it in order to have the validations stated on the scenario. Having this in mind let's start our exercise.

#2.1 - Setting up our test scenario base structure

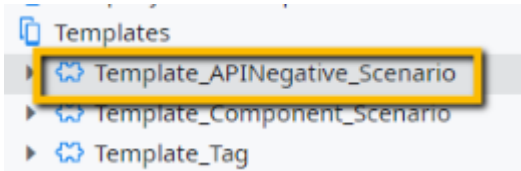
- Now, let's create a static entity called "DataSteeringTest" with the [DATA STEERING TABLE](#) column fields and one extra field called ScenarioName (the data for you to put in the fields and what records to add are the ones from that same table). This last field should hold a meaningful name in order for the scenario name to change based on the test running. For the CompanyType and Country you need to reference those tables from the module Customer. The static entity should be something similar to this:



- Once you finish the static table, add a new screen to the TestFlow, naming it "Company_DataDrivenTestSuite" and select the Data-Driven Scenario (Static Entity) template:



8. Still, inside the module created create a new UI flow called “CompanyCreateOrUpdateTestScenarios”. This will be the flow to hold the test scenario web blocks.
9. Now, let’s start to build our test scenario, to accelerate the development we will use the template webblock called “Template_APINegative_Scenario” (from the Templates flow) as the starting point.

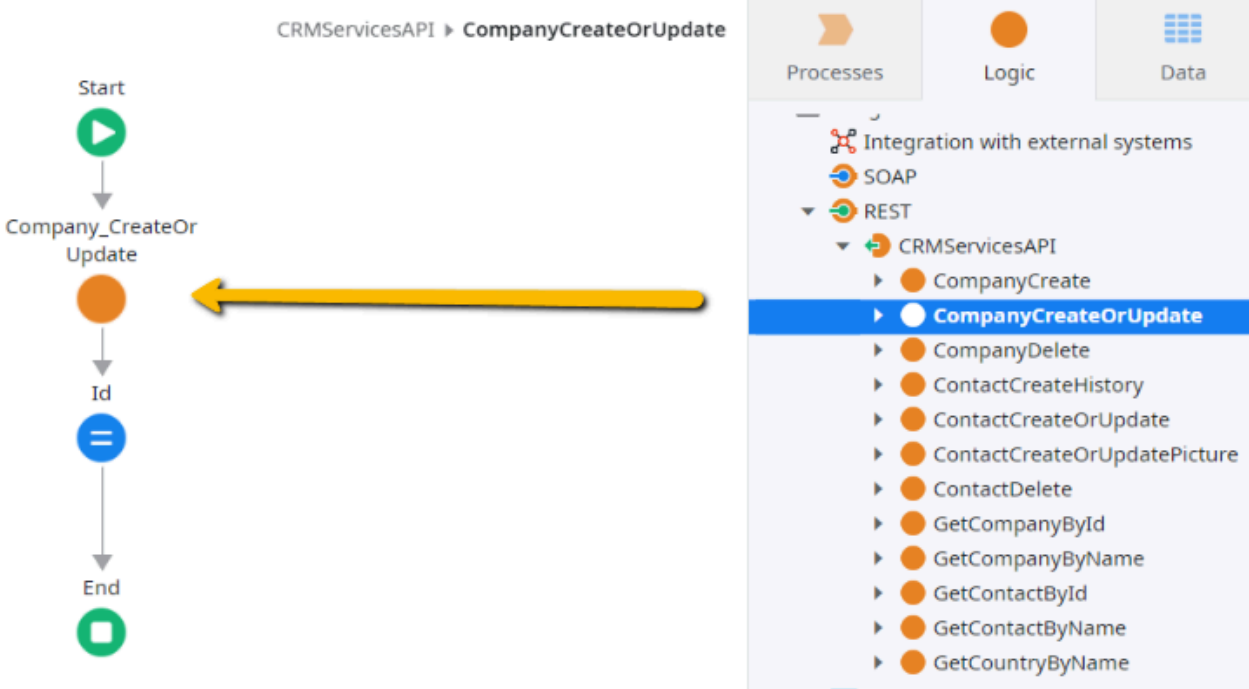


10. Rename the webblock to “Company_DataDrivenTests”, drag it to the flow “CreateCompanyOrUpdateTestScenarios” and now let’s add to the webblock an input parameter with the static entity record data type:
 - a. InDataSteeringTestRec [record]
11. Add 2 extra input parameters to have the Company Type and Country Code for filling in on the descriptions of our tests since in our static entity we have only the IDs. In order to fetch that information, on the aggregate to iterate the static entity just do a join with the entities “**CompanyType**” and “**Country**” from the **Customer module**
12. For now we are just adjusting the test steps descriptions, the text on them should change according to the test that is running, so adjust them to be dynamic (use [expressions](#) and [IFs](#)).
13. Before we implement the logic for our test to run, we need to make sure that our API has the proper validations in place.

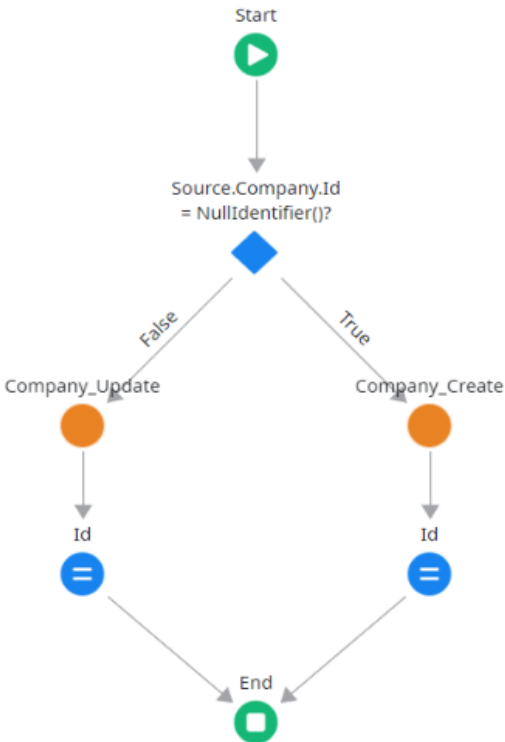
#2.2 - Updating our API under test

The goal of this part is to validate if the “CompanyCreateOrUpdate” API handles the validations of the **mandatory Parameters** and their **format (Interoperability)**, returning the correct messages, based on the requirements of the test scenarios.

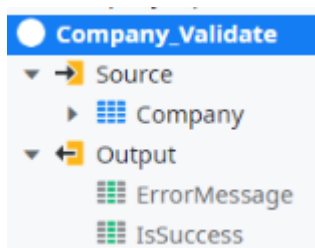
14. Go the “**Customers**” module, and if you open the API that we are exposing we are using the action “Company_CreateOrUpdate”:



15. Inside server action called “Company_CreateOrUpdate” you will see that there’s no validation being done on the flow:



16. So, let's create a server action called "Company_Validate", with an input parameter with structure of the Company record and with an output structure having the fields "IsSuccess" and "OutputMessage". So, something like this:

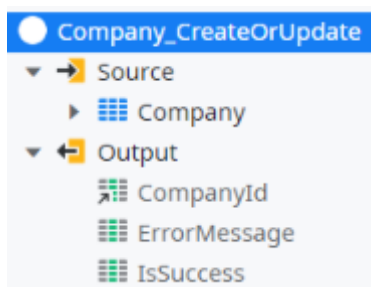


17. On that action created, the expected validations (based on the scenario) are the following:

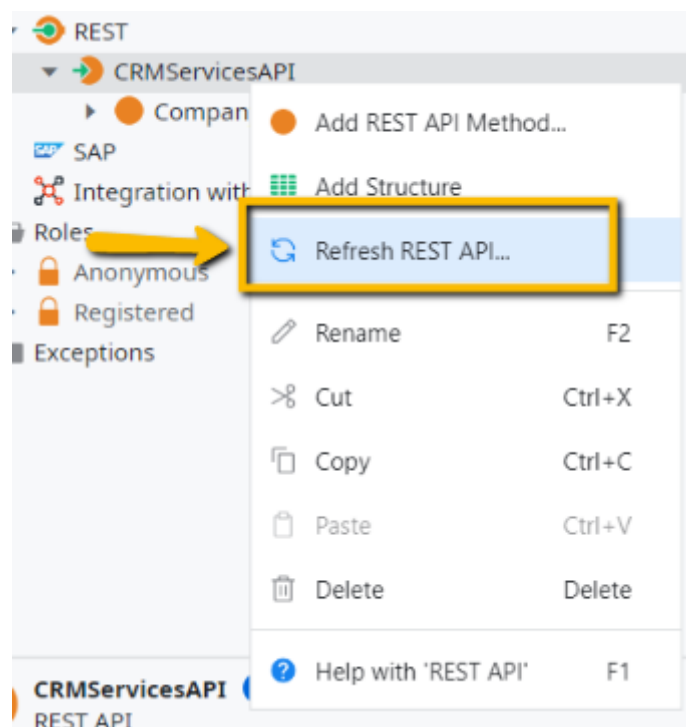
- **Mandatory fields check:**
 - Company name
 - Company type
 - Company email
 - Company country
- **Format validations:**
 - Company email must be a valid email address
 - Suggestion - use the platform built-in function "EmailAddressValidate"

Attention: when building this validation logic, ensure you build all the correct validations with the expected output message in order to match with the output message of the [DATA STEERING TABLE](#)

18. Now that you have created the validation action, you need to adjust the server action "Company_CreateOrUpdate" and also the REST API that we are exposing. So, let's start by changing the server action to have an output structure with the fields "Id", "IsSuccess" and "OutputMessage".



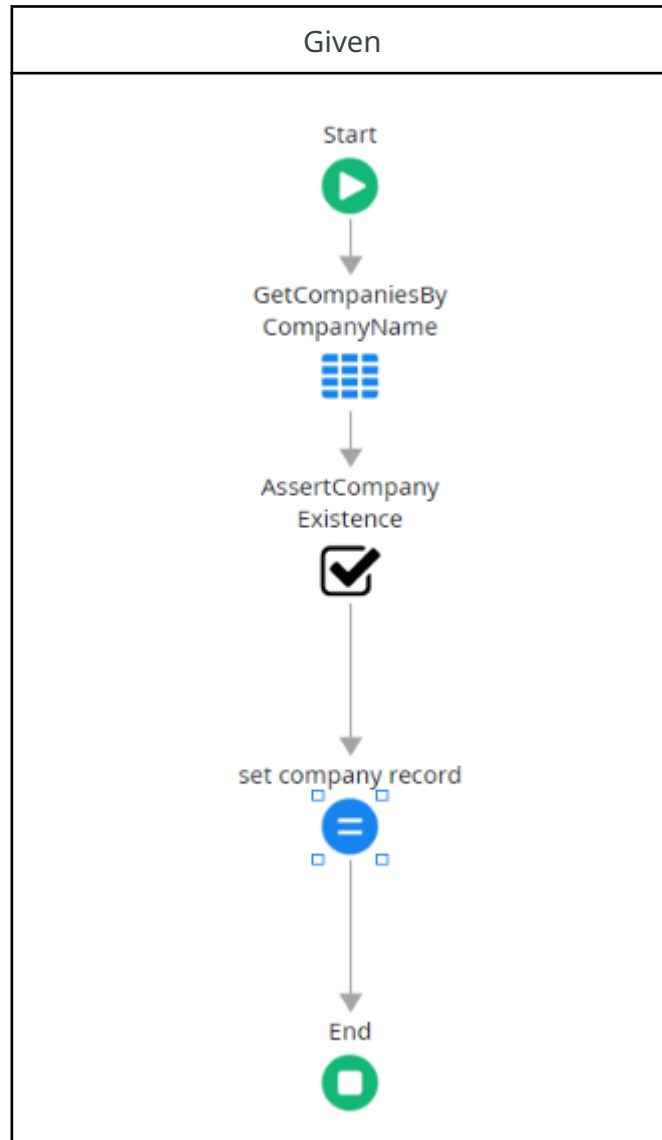
19. Inside the action “Company_CreateOrUpdate”, add the validation action “Company_Validate” before the server actions “Company_create” and “Company_update”. Now, after that action add IF condition to check the response “IsSuccess”, if the response is “False” (false branch) set the output with the Company_Validate response fields IsSuccess and OutputMessage.
20. Now that you have fixed the server action with the validations, it’s time to fix the API to also send the new output structure. First put the output structure of the Alike this:
- Id [CompanyId]
 - IsSuccess [boolean]
 - OutputMessage [text]
21. Now, based on the response of the “Company_CreateOrUpdate” action, set the REST API output structure with the specific values from that response.
22. Since the response of the REST API has changed, you need to readjust also the consumer modules. Go to the module “CRMServicesAPI_Tests” and refresh the REST API in order for the output structure to be updated and fix any errors/warnings due to that change.



#2.3 - Building our test logic

23. After the previous changes are published, it's time to start building our test scenario steps logic, so, inside the module "CRMServicesAPITest" and let's start by creating the GIVEN step to verify if there's already a company with that name.



24. Below we have the bdd step block logic that can be used:



25. Now that you finished the GIVEN, let's create the WHEN step logic, to call the REST API using the record that was set on the previous step (GIVEN) and store the response in our local variables to check on the THEN steps. It should be something like this:



26. Finally let's create the THEN step based on the previous stored response:

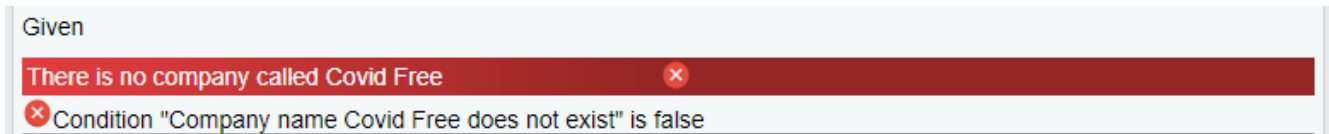
THEN (if IsSuccess = True) Verify if a valid id was generated for the record.	THEN (if IsSuccess = False) Verify if the output message is the expected for the error
↓ AssertResponse  ↓	↓ AssertOutput Message  ↓

27. Once you have finished the last step, you need to create a new UIFlow called "CRMServicesTestSuites", this will hold the test suite screens.

28. Now, inside the above UIFlow add a new screen, using BDDFramework Data-Driven Scenario (Static Entity) screen template and then call it "CompanyCreateOrUpdateTestSuite" that basically will have all the tests for the CompanyCreateOrUpdate API.

29. On the test suite page "CompanyCreateOrUpdateTestSuite" (created on the previous point) inside the preparation change the aggregate in there to fetch data from the static entity "DataSteeringTest" created earlier. To ensure the expected outcome of the exercise, order the records by the "ExecutionOrder" attribute, ascending.

30. Finally, add the web block "Company_DataDrivenTests" to the list record already created replacing the template webblock.
31. Run the page and you will see that only the first have passed. All other tests should be failing on the first GIVEN step. And if you re-run your test suite, all will fail due to the below condition.



If this is not happening to you it means that you are already taking into account the correct setup and teardown steps. If yes, the next part of our exercise is a confirmation of what you have done, otherwise you should correct where you feel necessary, following these next steps.

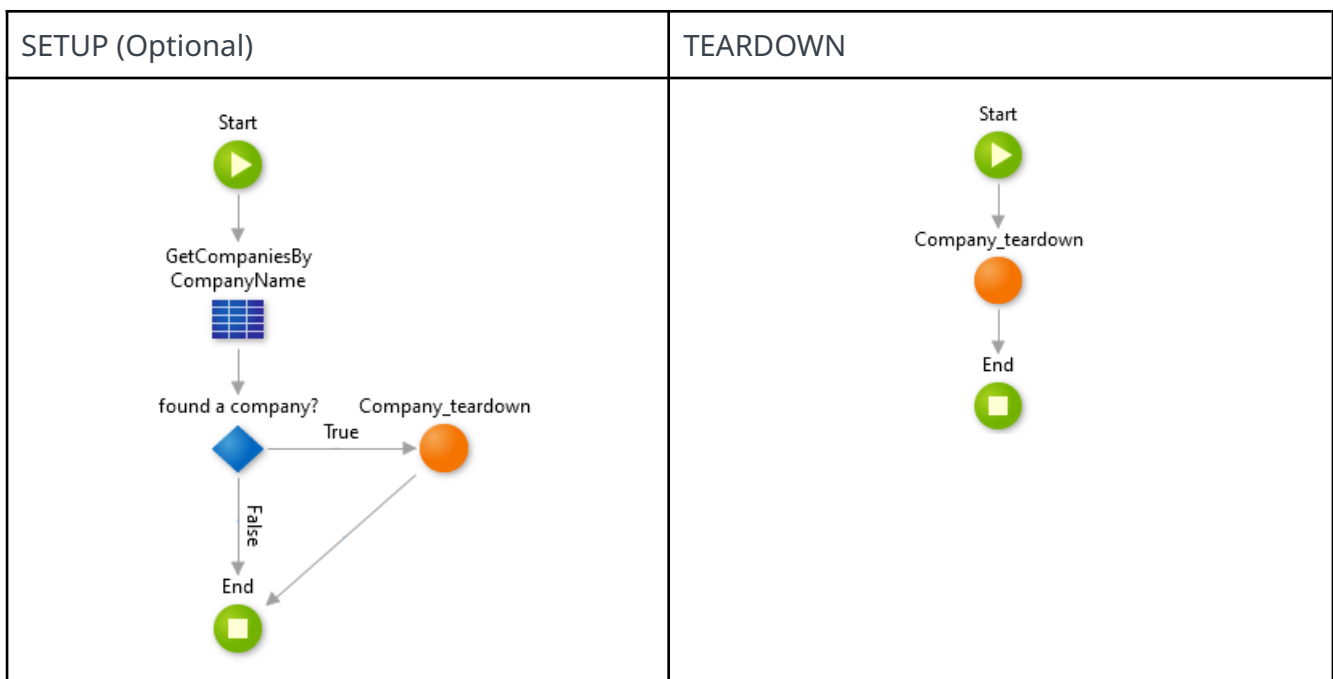
#2.4 - Setup & Teardown steps

32. The reason for these failures is because we have an assertion validating the non-existence of the company, and our first test is actually creating one with that name for a success scenario. So, what we have to do to solve this, is to create a teardown step that resets the system to its initial state whenever a company is actually created. Following best practices, we should make use of our **SETUP** and **TEARDOWN** steps in order to keep the isolation of the tests, and ensure repeatability without human intervention.

When testing APIs, just like we do with component testing, you need to take into account the data that is required for the test.

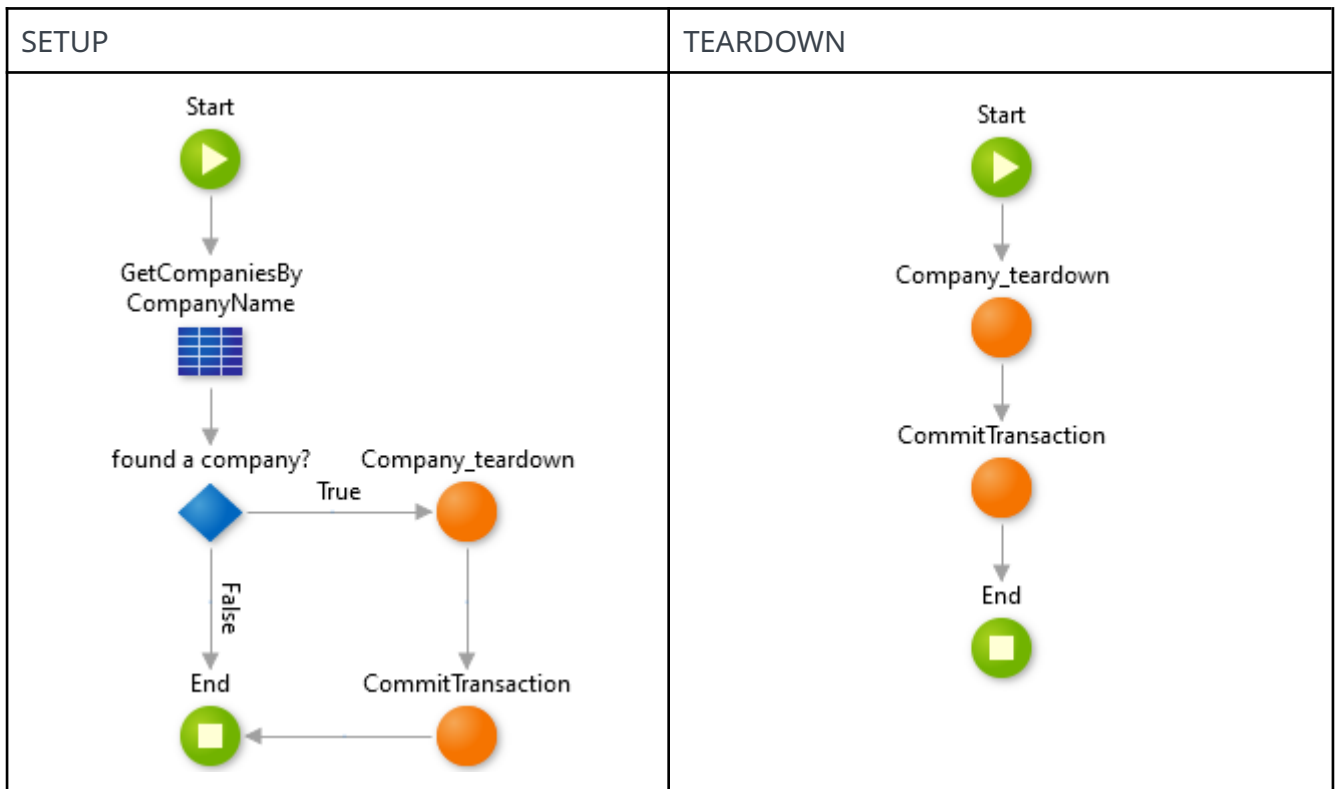
- **SETUP**: use it when you need to ensure pre-existing data that is required for your test execution.
- **TEARDOWN** : use it to ensure you clean-up any system changes that are a result of your setup and its test execution. It should ensure that the system is reset into its original state.

33. Since our test execution can create data, we will need a teardown stage at the end of our test scenario. So let's start by creating a new service module inside your test app called "CRMServicesAPITest_DataSetup"
34. Now consume the server action "Company_Delete" and the "Company" entity from the "Customers" module.
35. Create a folder on the server actions called InternalAPIs
36. Inside that folder create a public server action called "Company_Teardown" and in this action use the referenced action. Given a company id, if it isn't a null identifier, it should call the delete action.
37. Now, go back to your test scenario and add the teardown in order to delete the company created.
38. Because we already have an existing record in our database, we can also add a Setup step to our scenario. It should check if the input company name exists, and if it does, we can reuse our "Company_Teardown" action to remove it, this way ensuring the required pre-conditions for our test to successfully execute.



1. Our test suite will execute multiple tests, where **the action under test is an API call**. While our **Test suite steps** has one **single transactional scope**, each **API call has its own transactional scope**. Our Setup and Teardown steps are using **server side logic** to

delete the record, so it **executes within the transactional scope of the flow**, which is only **committed at the end**. In this exercises you might not have concurrency problems but if you want to make sure it's avoided, just add the commit transaction (action needs to be imported from the "System" dependency) on your Setup & Teardown code to look like this:



39. Now publish your test application and re-run your test suite. and all the expected success tests should **PASS**. If at this stage you still have any **FAILING** scenario, you probably have an error either in your test code, or on the code under test.

#2.5 - Error Handling and Status Codes

One of the important best practices when doing APIs is to handle errors with status code in order for the consumers to receive a proper response. So, the API should have an error handling and should raise HTTP error codes for giving meaningful information.

The below scenario is basically the same set of tests used in the [first scenario](#), but it adds error handling specific test data (we added a new column called "Status Code" and slightly changed the scenario itself).

Scenario: [<ScenarioName>](#). Test w/ error handling

Given:

There is no company named [<Name>](#)

And

I want to create the company with name [<Name>](#), Company Type [<Type>](#), email [<Email>](#), country with code [<Country>](#)

When:

I call CompanyCreateOrUpdate endpoint with those attributes

Then:

```
if (expected success = true)
{
    The request was successful and response returns Id a number <> 0
}
else
{
    The endpoint call fails with status code = <StatusCode>
    and the error message = <OutputMessage>
}
```

DATA STEERING TABLE (New column in green background)

Name	Type	Email	Country Code	Expect Success	Status Code	Output Message
Covid Free	Customer	covidfree@example.com	US	true	200	
Covid Free				false	400	Company type, Email and Country are mandatory
Covid Free	Customer	covidfree.example.com	US	False	400	The email format is not valid

40. Let's start by fixing our API, to include these error handlings. You can either change our original test scenario, or creating a new one by copy the original and making the required changes. It's advisable to copy the original and change it in order for you to have both tests, the first one just for reference.

41. Now that you have the new webblock ready, add a new attribute to the test data static entity, which will hold the expected status code, and then update the static records with the expected status code value based on the [above test scenario](#).

42. As a second step, adjust the webblock based on the test scenario. If you noticed the THEN step of this scenario is different from the [first exercise](#). The text on the steps

description should change according to the test that is running, so use expressions with the proper input parameters.

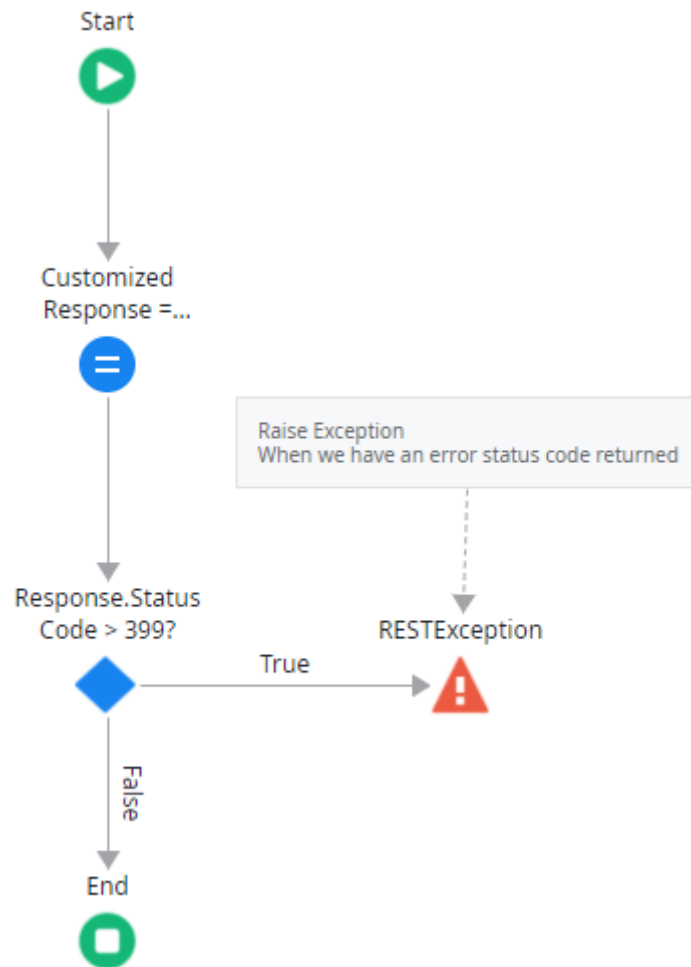
43. Now that we have both the scenario updated, and the static entity data updated, we need to adjust our "CompanyCreateOrUpdate" REST API. Following the best practices, the APIs should have an error handling and should raise HTTP error codes for giving meaningful information. Based on the above table if a validation fails it should raise the error and set the proper status code. Go to the "Costumers" module and add the dependency action "SetStatusCode" from the "HTTPRequestHandler" extension
44. After you reference the action go to the REST API and after the action "CompanyCreateOrUpdate" if the IsSuccess output field is equal to "false" you will set the status code and raise the exception. Use a custom user exception created by you to raise the error.



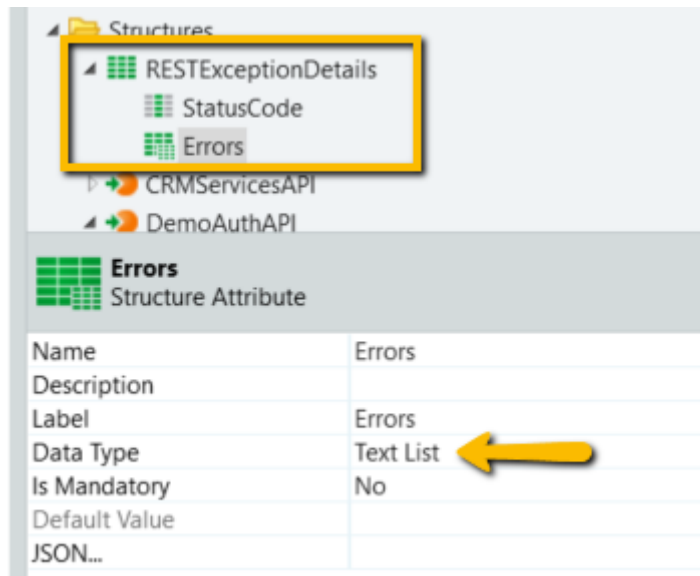
45. Now, it's important to understand where to put that error handling. If somehow you were thinking that the validation action should have this error handling is wrong because if the action is going to be used in other places, you don't want to raise unexpected errors. That's the reason to put this handling directly on the API method.

Now that we have the API raising the HTTP error, we need to handle these errors on the consumer side. So go back to the "CRMServicesAPITest" module and let's edit the "OnAfterResponse" handler on the consumed API.

Inside the "OnAfterResponse", if the response sends a status code > 399 (which based on our documentation is the minimum code that does not correspond to an error), we will raise an exception created by us on this consumer side. Create a user exception called "RESTException" and use it to raise the error based on the code and send the "ResponseText" field in the exception message:



46. What we did on the previous point, is to fetch the error of the response and raise a custom exception to send the error structure of the REST JSON response which is `StatusCode[text] + Errors [Text list]`. For us to complete our updated test scenario, we need to handle this custom exception inside our test steps. So still inside the “CRMServicesAPITest” module, we need to create a structure like this to hold the response:



47. Now on the WHEN step of the scenario that makes the call to the API we need to handle the custom exception and store the values to be validated on the THEN steps. In order to do that you will need to deserialize the exception message using the data structure created before (StatusCode[text] + Errors[Text list]). It should be something like this:



48. The output message coming from the exception handle should be stored in the same variable already being used to store the output message from the REST API.
49. Finally, on the THEN steps use the stored values to do the proper asserts.
50. Now that we have our scenario complete, and re-run the test suite to see the results. if you created a new webblock make sure you add it to the Test Suite replacing the first exercise webblock before running again the test.
51. If you noticed there are some patterns that we have done more than once...which is whenever we assert if a company with a given name exists. So, to conclude this part of our exercise, let's make a server action for each reusable case with the proper assert condition in order to be reused. Replace the asserts of this scenario and the [first scenario](#) by these actions. Now remember to do this in similar cases (by the way, did you place it in the correct module? After all, these are shared patterns. Remember the architecture of the tests).

Summary

With this exercise you learned how to check the parameters needed for the request and handle errors on the API and to send the proper output with meaningful information, including status code. It is a good practice to build your API's having this in mind.

You also learned how you can implement API exception handling, that you can then use in your test applications to verify your API expected behavior for error scenarios.

Exercise #3 - Security (expected duration 1h 30m)

On this exercise, as part of the security test we will check the following :

- Authorization

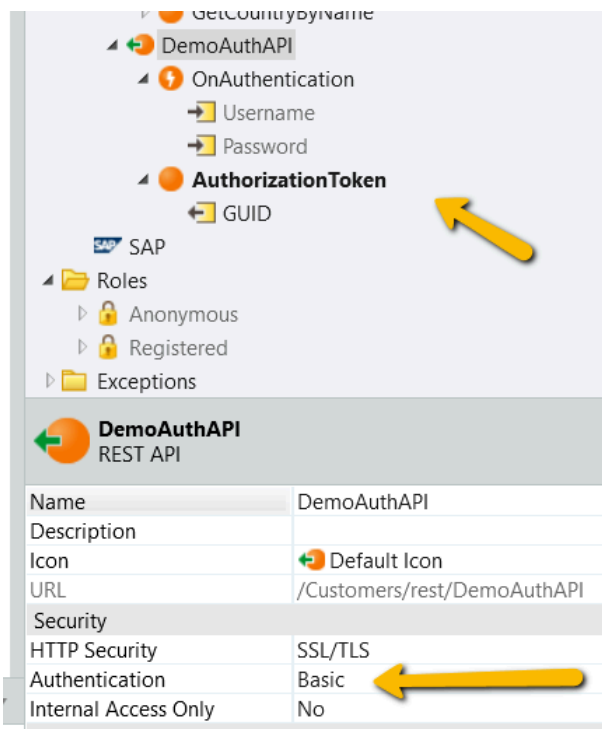
The goal of the exercise is to put in place the test in order to check the security of an API. The purpose is not only for you to learn how to test the security of an API, but also to learn some best practices when setting up the security of an API.

This API does not exist, so we are going to create that API to be tested.

#3.1 - Create Demo Auth API

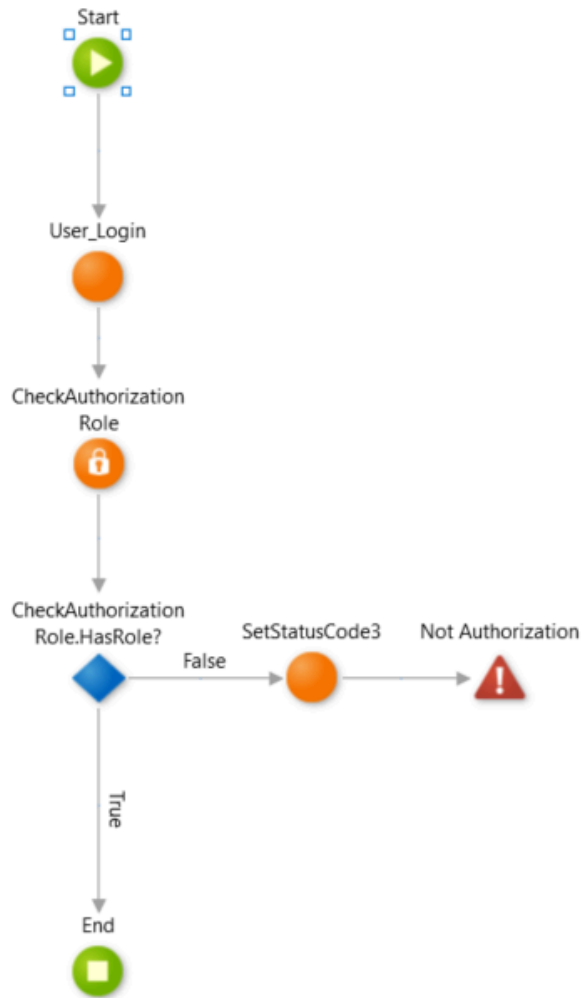
For the sake of the exercise we are going to add basic authentication to the API but we are going to restrain the ability to use it only to users with a certain role. The next steps will guide you on this process, and the first step is to create the REST API itself:

1. The next test we want to mimic the assessment of the API security with basic authorization and in this case we are not going to do a successful test login but an invalid login. So, go to the module "Customers" and create a new REST called "DemoAuthAPI"
2. Then, add a new method called AuthorizationToken with "basic" authentication and returning a GUID (use the system action generate guid) as a dummy output, for the sake of this exercise you just need to send back something on the response:



3. Since you have set the "basic" authentication there's a system handler that was added to the API called "OnAuthentication". In order to introduce a restricted access to the to the REST API, create a role called "Authorization"

4. Now inside the "OnAuthentication" handler, add the check of the role created on the previous step and if the user doesn't have this role set the response code to 403, using the "SetStatusCode" action from "HttpRequestHandler" extension. It should be something like this:



#3.2 - Test the Security of the API

Now that we have the API created it's time to test it. We have the following test scenario and for the sake of the exercise we are only using one user, but the correct implementation would be a data driven scenario having seed data for 2 users, one with a role and another without role:

Scenario: Verify API Security

Given:

I have the user with username "aaron.sullivan" and password <password from aaron on your personal environment>

When:

I call the AuthorizationToken method from DemoAuthAPI endpoint with the given details

Then:

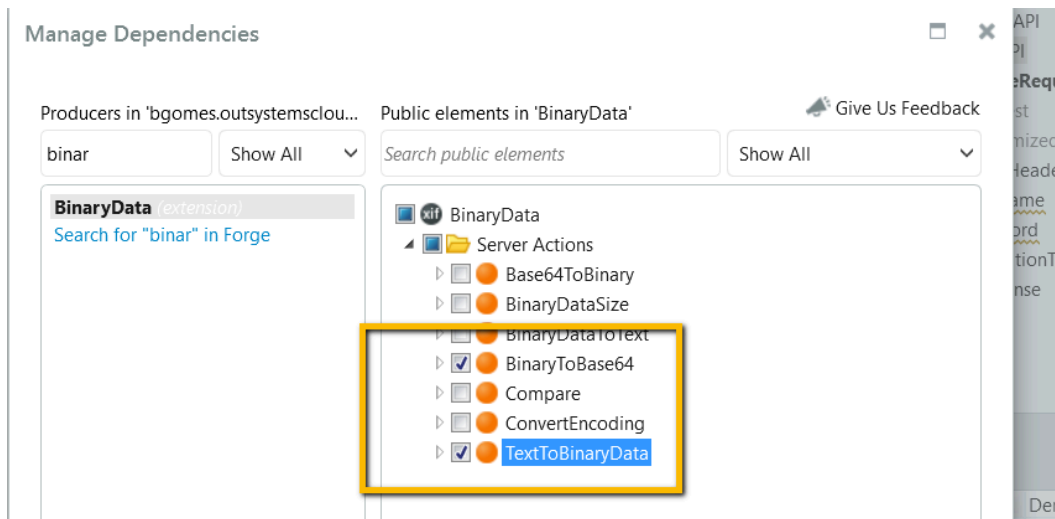
IF (user has no role)

The call returns the code 403 (forbidden)

Else

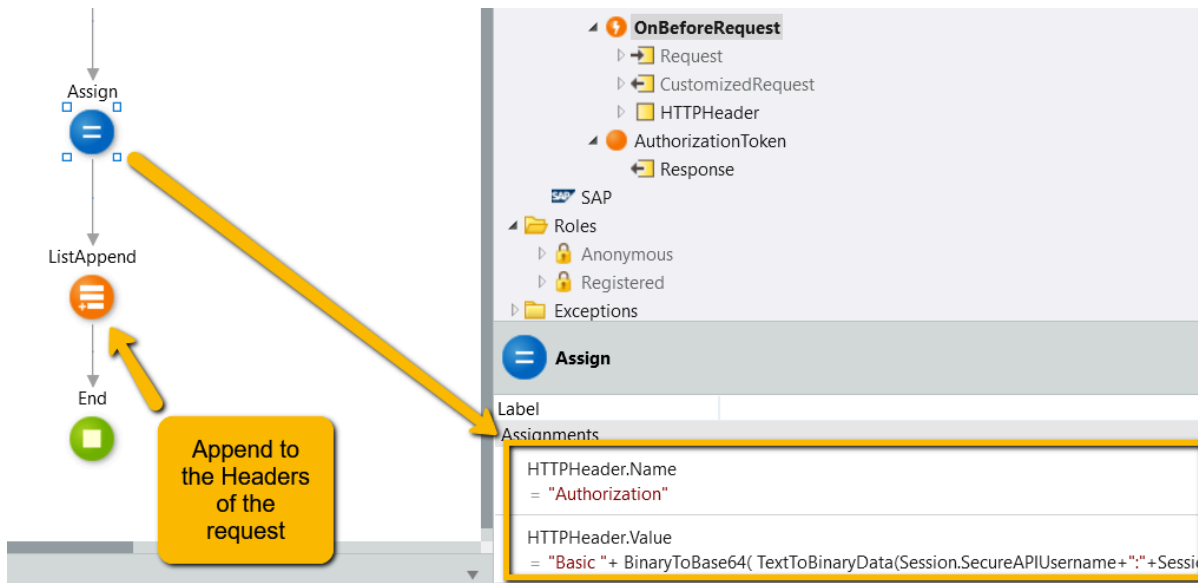
The call returns a GUID not empty

5. Let's start the development of our test. First, go back to the "CRMServicesAPITest" and consume the above API.
6. Now set on the advanced options of the consumed API to have a "OnBeforeRequest".
7. Click on "Manage Dependencies" and consume the actions "TextToBinaryData" and "BinaryToBase64" from the BinaryData extensio.

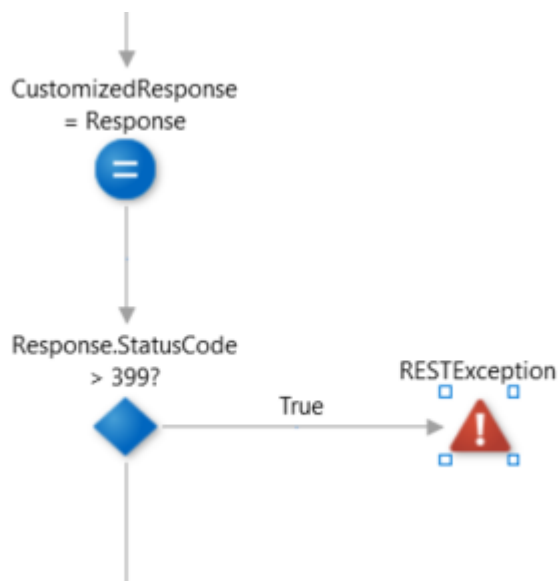


8. Now let's create two session variables **SecureAPIUsername** and **SecureAPIPassword** to hold the username and password to test the security of the API.

9. On the action **“OnBeforeRequest”** let’s add the basic authorization with the session variables that hold the test username and password like this:



10. Now add the handler **“OnAfterResponse”**, and let’s raise an error if the status code > 399 (which based on our documentation is the minimum code that does not correspond to an error). You already have something similar on the exercise before for the error handling:



11. On the message of the raised exception send the **“ResponseText”** field of the Response structure.

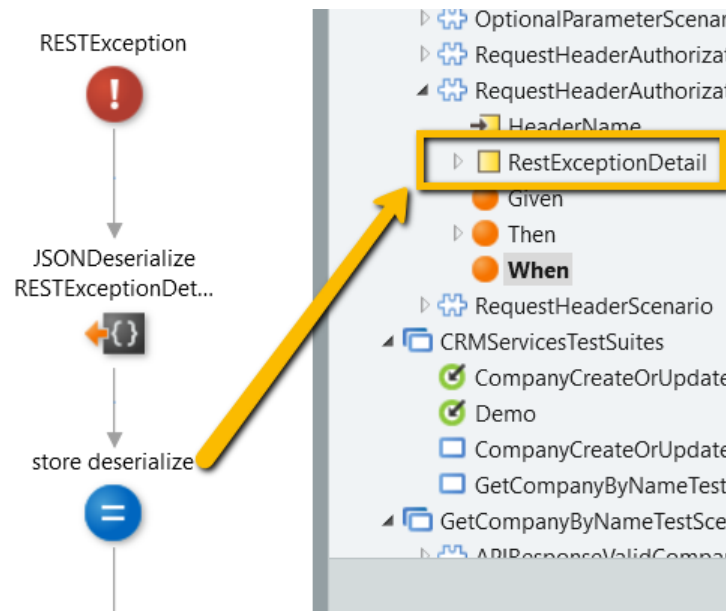
12. Now we need to set our seed data for the test, so, create a static entity called AuthenticationUser with 2 attributes "Username" and "Password" need to get the user credentials. If you don't know the password of user *aaron.sullivan* go to the Users app <https://<yourpersonaldomain>/users>, access it by using the application user "admin" and change the password of the user "aaron.sullivan" to use it on the scenario [above](#)
13. Since we already have the user in place, it's time to create a new web block to test this scenario. So, create a new Webflow to hold this new webblock and create a new test suite screen to add the test scenario.
14. Now, that you have already the test suite in place let's begin to work on the test scenario. On the GIVEN step of your test scenario make sure change the session variables to set up the credentials before making the call to the API



15. Now let's reuse the structure we created to hold the errors raised by an API.

<ul style="list-style-type: none"> RESTExceptionDetails <ul style="list-style-type: none"> StatusCode ErrorMessage CRMServicesAPI 	
ErrorMessage Structure Attribute	
Name	ErrorMessage
Description	
Label	Error Message
Data Type	Text List
Is Mandatory	No
Default Value	
JSON...	

16. After making the call the response might be an error, so, add an exception handler to grab the specific error handler that we have created on the "OnAfterResponse" and store that information to be used in the next steps, in a variable with the structure built on the above point



17. Now that you have set the when step it's time to do the THEN step. In this step you will check the status code stored on previous steps
18. After finishing the scenario above, add the web block to the test suit and run it, this test should **PASS**
19. One test is done, so it's time to do the second test, go to the Users and add the role to aaron.sullivan and run again the test, it should also **PASS** but showing the description of the **Else** part of the THEN step. This step was manually because for the sake of the exercise we did not created our seed data, so, create our users, one with role and another without the role to make the proper implementation,

Summary

With this exercise you learned how to test security of your API. This also will help you to understand the concerns that you have to have in order to make a correct API for other consumers

Exercise #4 - Data (Optional, expected duration 1h 30m)

On this part of the testing strategy, the goal is to test the following :

- Is data persisted as expected
- Verify functional and non-functional requirements

For this section we are going to check if the data created matches what is saved on the database, and for that we are going to use the API GetCompanyById to compare if the values are correct.

Exercise #4.1 -Check Data if data persisted (expected duration 30m)

1. Create a new web block on the scenarios UIflow to hold the following scenario (use the simple scenario template as starting point to accelerate):

Scenario: Verify Company Create Or Update Saved Data

Given:

There's no company named "Lost Chain"

And

I want to create the company with the name "Lost Chain", Company Type "Customer", email "lostchain@example.com" and country with code "US", Phone "(435) 341-1234"

When:

I call CompanyCreateOrUpdate endpoint with the given details

Then:

The endpoint returns successfully (200 ok)

and

The information created is correct

2. Now, on the THEN step use the GetCompanyById to retrieve the data saved and make the proper assert (tip: use the assertTrue to compare all value in just one condition). This will be done on the THEN step because if the API is slow the results from the DB might not be up to date.
3. To finalize our implementation, don't forget to do the Teardown step to clean the created data.
4. Add the web block to the test suites screen and run it. The result of this part should be **GREEN**

Exercise #4.2 (Optional) - Following the Testing Pyramid Thinking Model (expected duration 1h)

On the exercise #4 we are doing functional validation on our API endpoint to verify the correctness of the purpose of the endpoint on handling data. This is in fact a part of the POISED strategy, specifically for the Data part, but for this exercise now, we want to challenge you to think - is this test we just designed following the recommended practice of applying the Test Pyramid thinking model?

The fact is, we are testing an API that we control, right? This is an API developed within our outsystems factory and within our domain. On top of that, if we look at the implementation of our "endpoint under test", it is actually just executing a server action where all the functional behavior and data handling is done, and then grabbing its output and returning it.

So again, are we following the testing pyramid principle?

The answer is we're not.

The testing pyramid principle states that we should only perform our verifications on upper levels of the pyramid for those validations we cannot address in any way with lower level testing. And if we already have this validation done through lower level testing, it will then be redundant to repeat that same validation through upper level testing.

As such, the validation we are addressing in this specific scenario, should be done through component testing, and not API testing.

For this exercise, we want you to refactor the solution of the previous exercise to a component test. The solution will be pretty simple. The purpose of this exercise is just to highlight simple considerations one should make when building automated tests to ensure we follow the pyramid principle & avoiding duplication of effort/responsibilities across different test types.

Overall Summary

With these exercises not only you saw how to use the **POISED** testing strategy but also how to work with the SETUP and TEARDOWN data having to commit the information in order to avoid impacts on the other transactions when doing API testing.

So when you are building APIs make sure you follow these practices and test them, to be bulletproof as much as possible.