# Lab 9: Orchestrate SOAP and REST Endpoints

[Lab 9: Orchestrate SOAP and REST Endpoints](#_86maahp7gk5r)

[Overview](#_1hulvwn4ret)

[Step 1: Configure the Database Connector](#_hekv486au28q)

[Step 2: Configure the SOAP Web Services Consumer](#_r8erzm9us1p1)

[Step 3: Create DataMapper transformations](#_7p0upyjbgcnf)

[Step 4: Add an Error Handler](#_nfocr42gt6vh)

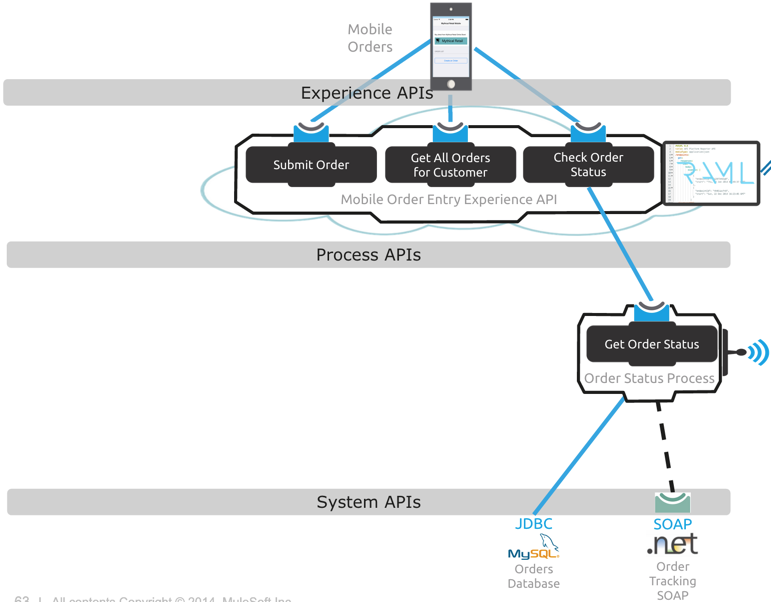
[Step 5: Run and Test the API](#_subkunlz4qxi)

[Step 6: Test from a Mobile Application](#_akdm01u2eh4t)

[Summary](#_u0r5ms2bvgno)

## Overview

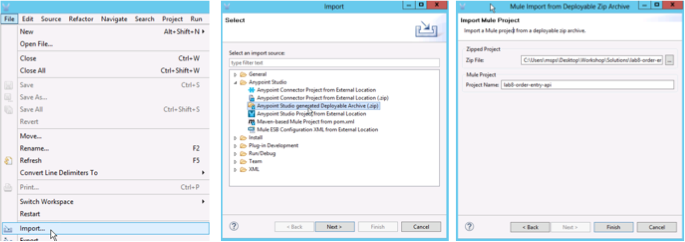
At this point, you should already have the order retrieval and submission implemented in the **get:/orders** and **post:/orders** flows of our API project created in Anypoint Studio. Let’s now proceed with implementing the method for checking order status.



In this lab, we will start implementing the remaining **get:/orders/{id}/status** operation to check the status of an order. We will do the following:

1. Configure the Database Connector to query the order price and status from a MySQL database.
2. Configure the SOAP Web Services Consumer to retrieve order tracking information from the Order Tracking SOAP Web service we proxied in our API Gateway in Lab 1.
3. Create DataMapper Transformations

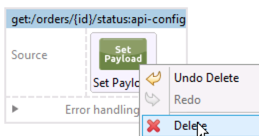
|  |
| --- |
| **NOTE:** If you wish to skip this lab or need to catch up, you can open the completed solution by Importing the following project:  **C:\Users\msps\Desktop\Workshop\Solutions\lab9-order-entry-api.zip** |



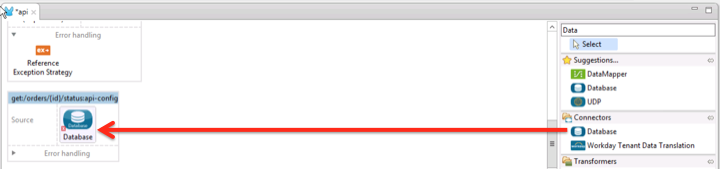
## Step 1: Configure the Database Connector

Lets implement the **get:/orders/{id}/status** back-end flow to check the status of orders by querying our order status from a **MySQL** database**.** We’ll also retrieve order tracking information from the **SOAP Web service** we proxied into our API Gateway earlier onin the first lab.

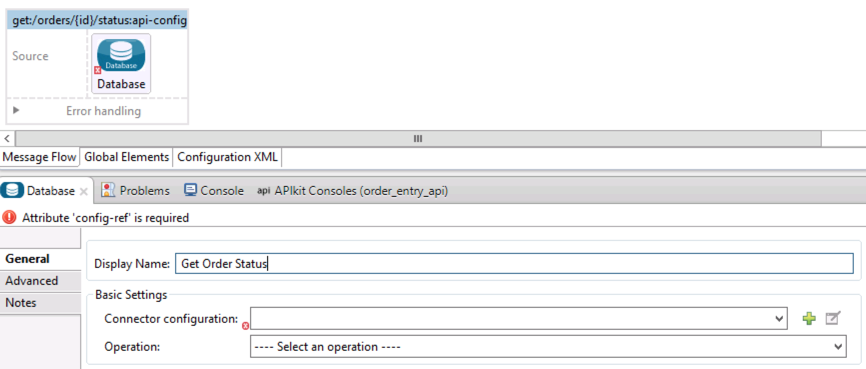
|  |
| --- |
| **NOTE**: Make sure you are in the **get:/orders/{id}/status:api-config** flow |



1. Delete the Set Payload processor from the flow named **get:/orders/{id}/status:api-config**.



1. Search the **Database** component in the toolbox and drag and drop it to the **get:/orders/{id}/status** flow



Configure the connector

1. First double-click on the Database Connector to display its properties
2. Change it's name to **Get Order Status.**

A best practice in Mule is to externalize connectivity information. To configure the database connector we will leverage this best practice.

1. Open **mule-app.properties** in **src/main/app**.
2. Insert the following into **mule-app.properties**

db.host=localhost

db.port=3306

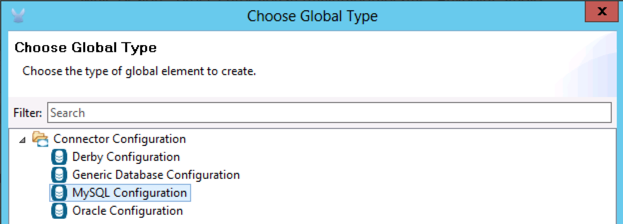
db.user=orderentry

db.password=Mule1379

db.database=orderentry

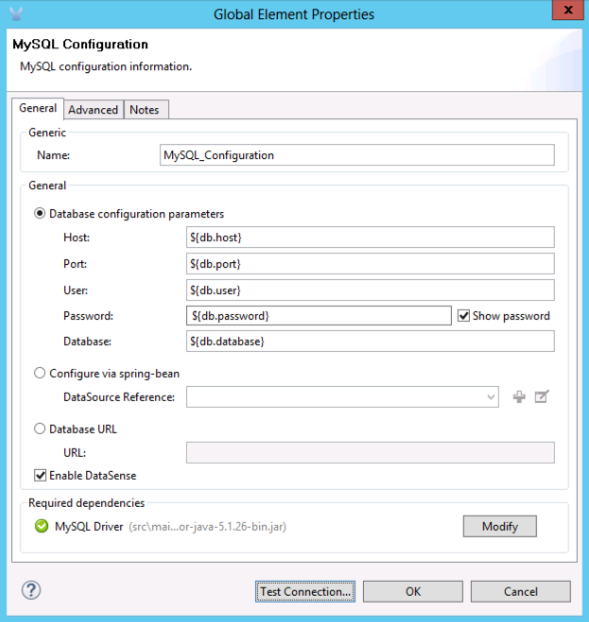
1. Save mule-app.properties

When you deploy and/or start a Mule instance you can override these properties with different values.

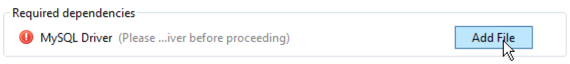
****

Now that you have your properties defined, lets add a new connector configuration.

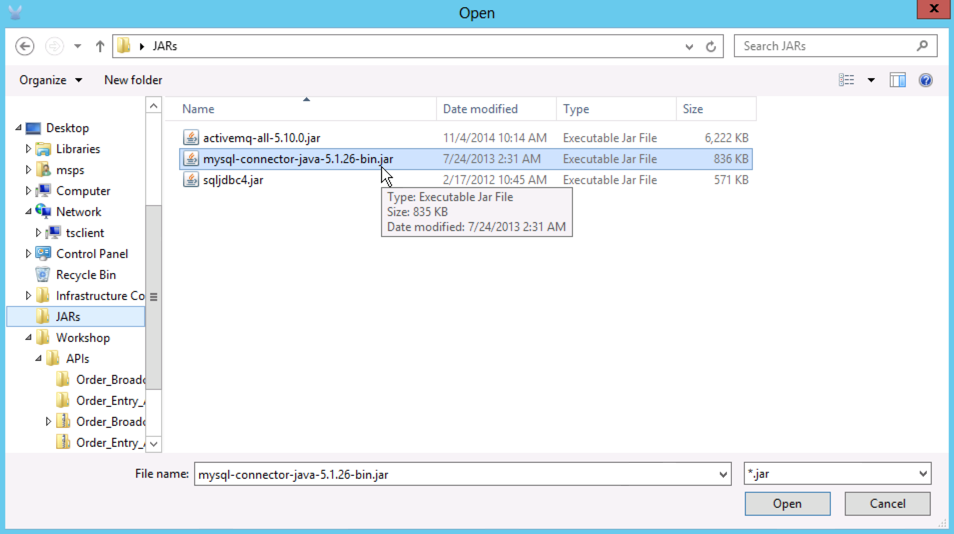
1. Click on  button on the Connector Configuration.
2. Select **MySQL Configuration**
3. Click OK.



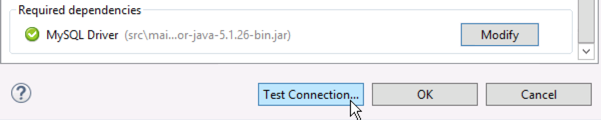
1. For our database configuration specify the following values for the properties:
   * Host: **${db.host}**
   * Port: **${db.port}**
   * User: **${db.user}**
   * Password: **${db.password}**
   * Database: **${db.database}**



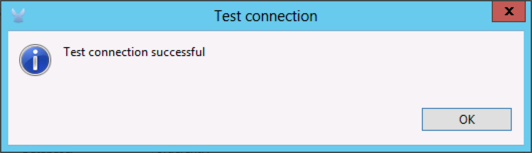
1. Scroll to the bottom of the MySQL Configuration under Required dependencies and click **Add File** to add the MySQL Driver.



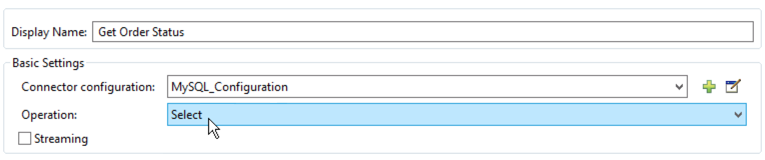
1. Locate the **mysql-connector-java-5.1.26-bin.jar** from the **JARs** directory on the desktop - **C:\Users\msps\Desktop\JARs**.



1. You should see a green check mark for the MySQL Driver.
2. Click **Test Connection** to verify the connectivity.

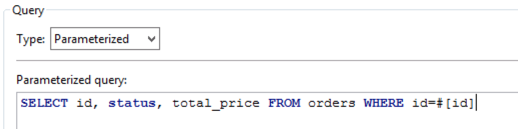
****

1. Click **OK**.



Once we have the connector configuration created, let's configure our operation.

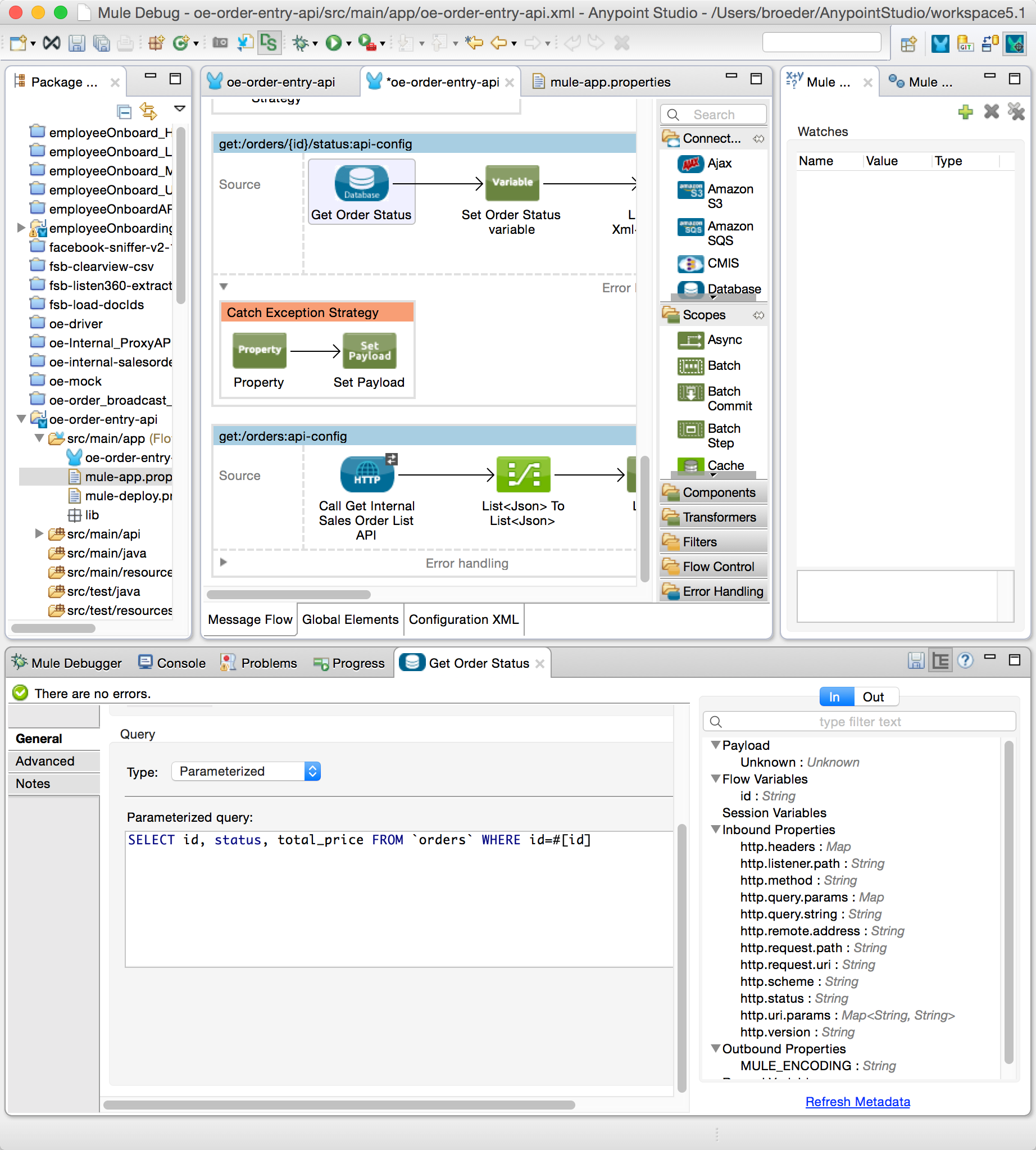
1. Choose the **Select** operation.



1. Chose **Parametrized** as the type and enter the query for getting the order status and price:

**SELECT id, status, total\_price FROM orders WHERE id=#[id]**

|  |
| --- |
| **NOTE**: The **#[id]** is a variable from the payload that is derived from the **{id}** parameter defined in the RAML for this method. |



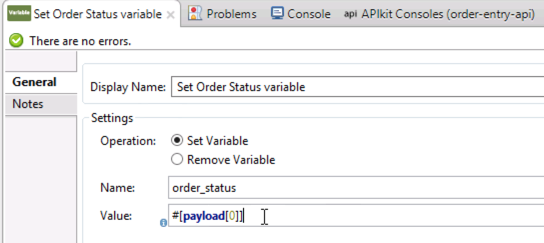
1. Make sure to Click *Refresh Metadata* in the bottom right corner of the database configuration.



1. If you get an error click *Edit connector* and test your database connection.
2. If that is OK and you still get an error, check your SQL syntax.



1. Search the **Variable** component in the toolbox
2. Drag and drop it after **Get Order Status.** We will store the **status** and **price** for a particular order ID in a variable.



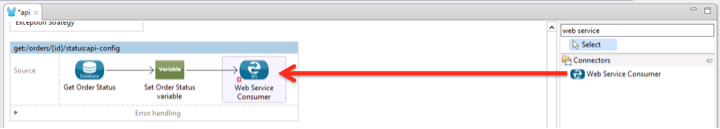
1. Double-click on the Variable to edit the configuration.
2. Change the display name to **Set Order Status** **variable**.
3. Select **Set variable** option
4. Set the name to **order\_status**
5. Set value #[payload[0]]

|  |
| --- |
| **NOTE:** The **payload** contains the results that the database query returns. This is returned as a collection of records. Since our expected response from the database will only be a single record containing the status and price for a particular order ID, we will only need to get the first record from the collection. The **payload[0]** expression returns the first record in the collection. |

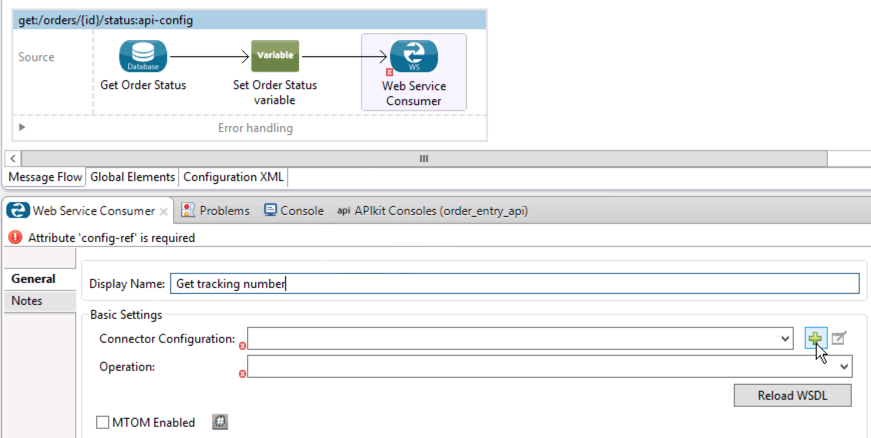
1. Click **Save All**. 

## Step 2: Configure the SOAP Web Services Consumer

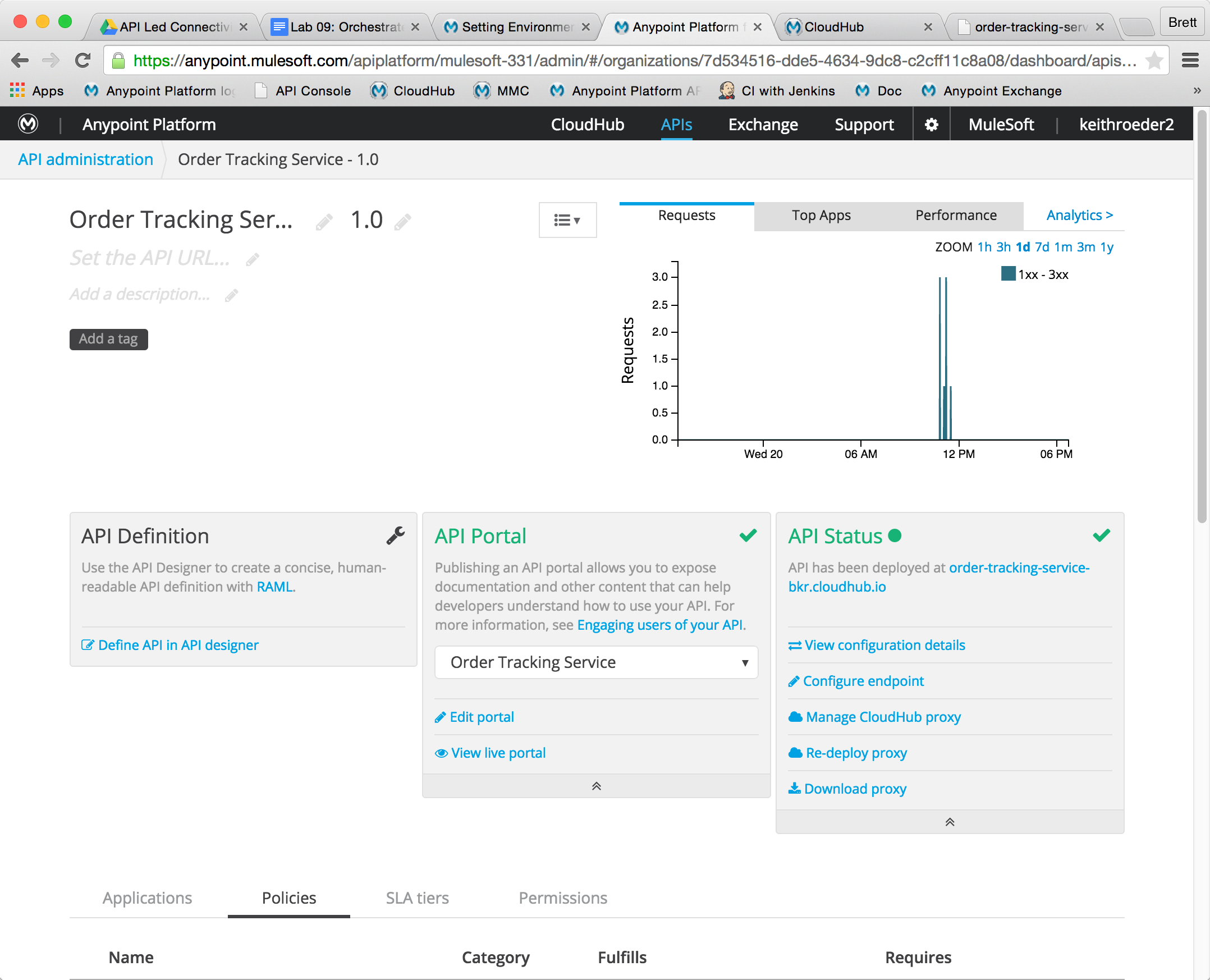
We will call a SOAP Web service to get the **tracking number** associated to the given order. This will be the SOAP web service we proxied in the first lab that is now running in our API Gateway.



1. Search the **Web service consumer** component in the toolbox
2. Drag & drop it to the end of the **get:/orders/{id}/status** flow.



1. Click on it to edit the configuration.
2. Change it's display name to **Get tracking number**
3. Click the icon to add a new configuration.
4. Name it **Order\_Tracking\_SOAP\_Service**.



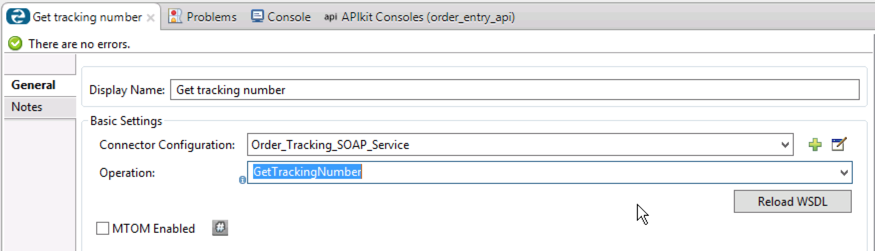
You are now going to call the .net Order Tracking SOAP service you proxied earlier. To make it easier, we are going to remove the policies.

1. Flip over to the browser
2. Remove the Rate Limiting policy that is applied. Now we don’t have to worry about providing credentials!
3. Flip back to Studio
4. For the **WSDL Location**, put the URL for the proxy you created in the proxy lab. Ex. [http://order-tracking-service-XXX.cloudhub.io/OrderTracking.svc?wsdl](http://order-tracking-service-1.cloudhub.io/?wsdl)

The Service, Port and Address fields will be automatically filled from the WSDL.

To do this successfully, you should have completed the proxy lab and you need to **disable all the policies** you previously set in the API Platform. Otherwise, you can just use the localhost WSDL directly.

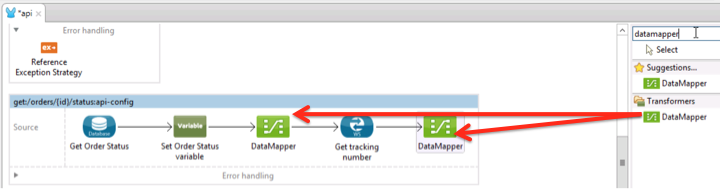
<http://localhost:86/OrderTracking.svc?wsdl>



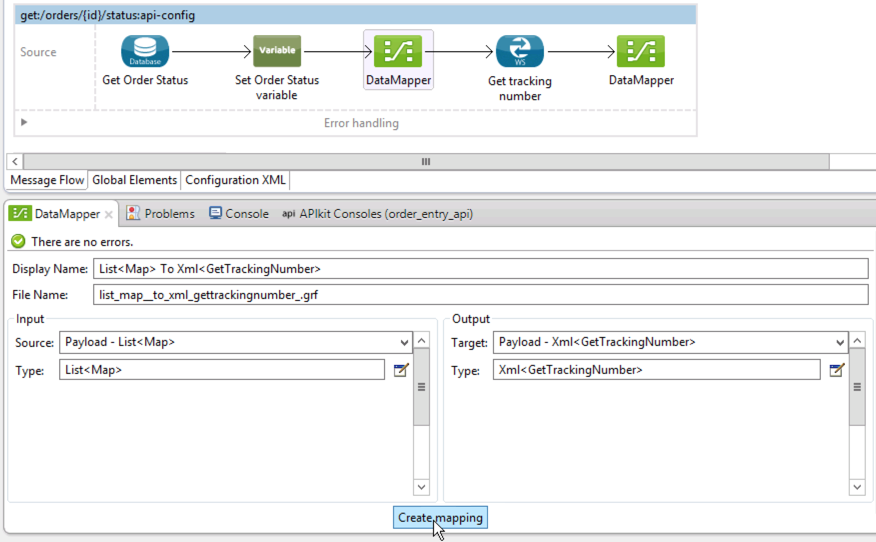
1. Click OK.
2. Choose the **GetTrackingNumber** operation.
3. Click **Save All**. 

## Step 3: Create DataMapper transformations

We will transform the information returned by the database connector to what the web service consumer needs to call the SOAP Web Service, which is XML. Then we’ll transform the SOAP Web Service XML response to what our REST API is supposed to return, which is JSON.



1. Search **DataMapper** component in the toolbox.
2. Drag & drop a Data Mapper component before the **Get tracking number Web Service Consumer.**
3. Drag & drop a Data Mapper component after the **Get tracking number Web Service Consumer.**

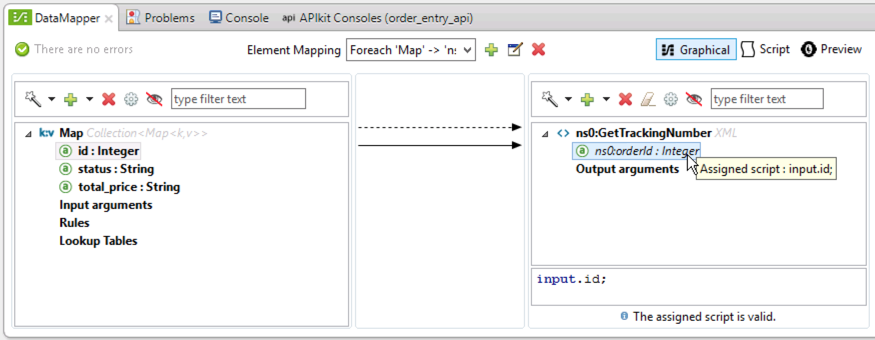
****

Notice that DataMapper used Data Sense to detect the inbound payload is JSON and the web service input is XML.

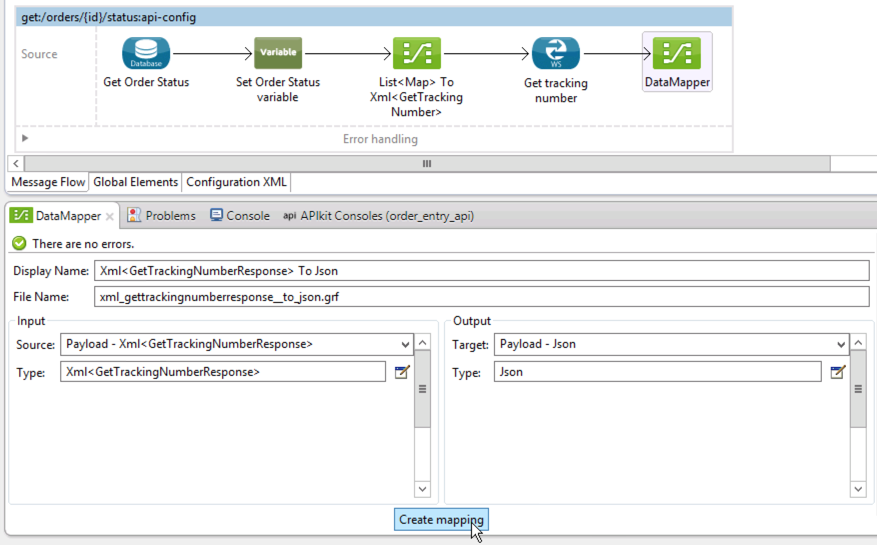
1. Click the first Data Mapper and then click **Create Mapping.**

****

1. Drag & drop the Map collection to the **GetTrackingNumber** root XML to define the element mapping.



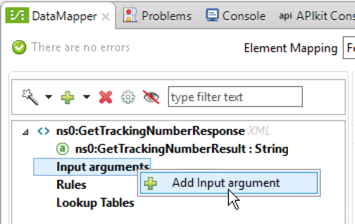
1. Map the **id** from database result to **ns0:orderId** on the right side.



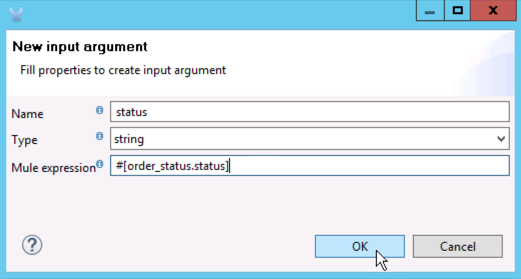
1. Click the second Data Mapper and then click **Create Mapping.**



1. Map **ns0:GetTrackingNumberResult** to **tracking\_number.**



1. Right click on **Input arguments**
2. Select **Add Input Argument** to add a new one for getting **status** field from the previously stored **order\_status** variable.

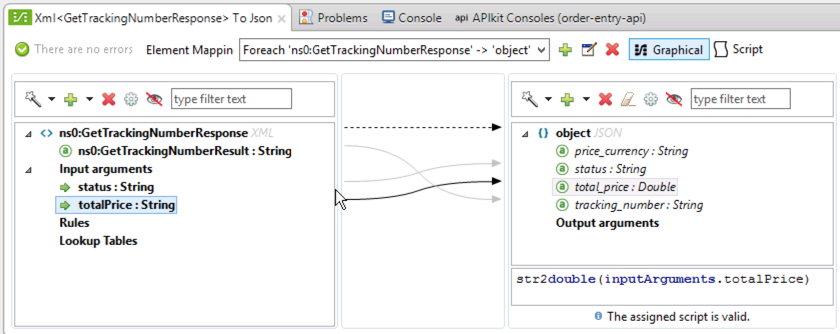
****

1. Set the following:
   * Name: **status**
   * Type: **string**
   * Mule expression **#[order\_status.status]**
2. Right click on **Input arguments** and add another input argument for getting the **totalPrice** field from the previously stored **order\_status** variable.



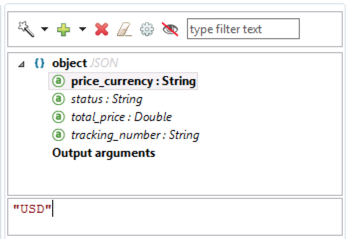
1. Set the following:
   * Name: **totalPrice**
   * Type: **string**
   * Mule expression **#[order\_status.total\_price]**

|  |
| --- |
| **NOTE**: we’ll use the type of String for totalPrice to accommodate the source which is storing the price as a string. Later on, you’ll see that our mapping will automatically convert this to a Double. |



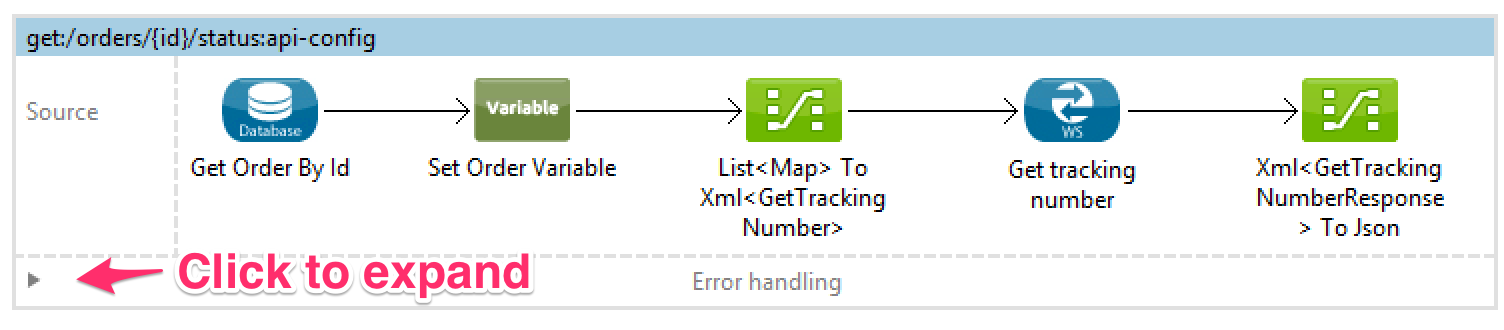
1. Map the **status** field from Web service response to the **status** field in the JSON response.
2. Map the **totalPrice** field from Web service response to the **total\_price** field in the JSON response.

|  |
| --- |
| **NOTE**: Notice how the mapping made use of the str2double function to convert the totalPrice to a Double. That’s because the outbound object is defined as Double. |



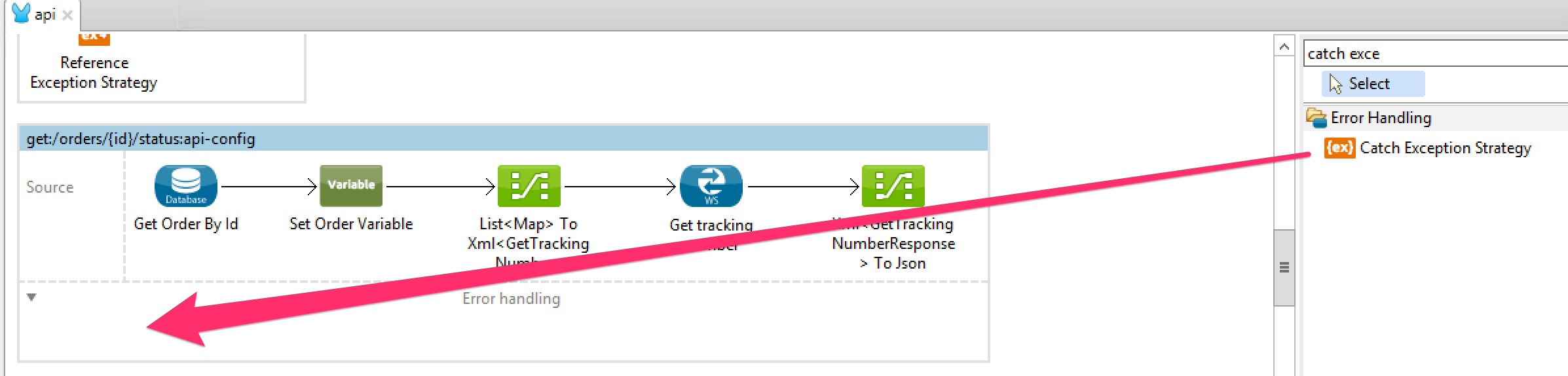
1. On the output on the right, click on the **price\_currency** field
2. Set value to **"USD"**

## Step 4: Add an Error Handler

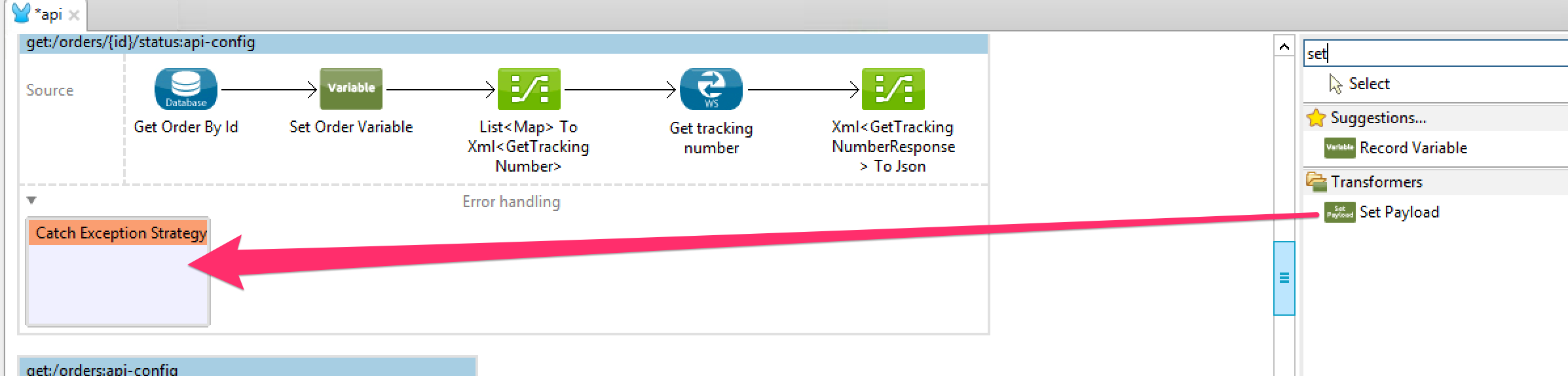


Mule makes it easy to add Error Handling to any flow.

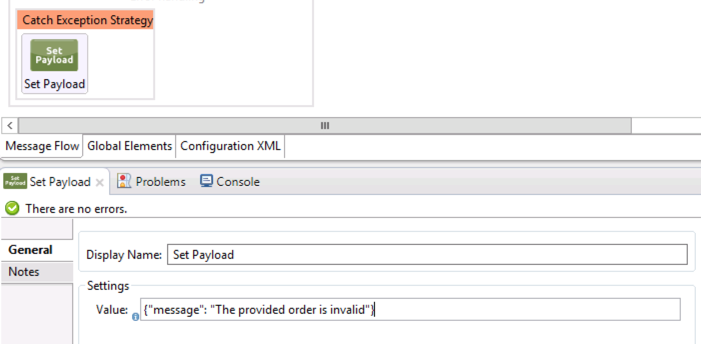
1. Click on arrow to expand the Error Handling container for the flow.



1. Search for **Catch Exception Strategy** in the component palette
2. Drag it to the Error Handling container.

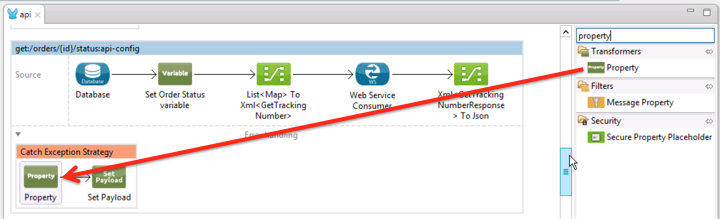


1. Search for **Set Payload** in the component palette
2. Drag it inside the **Catch Exception Strategy** previously added.

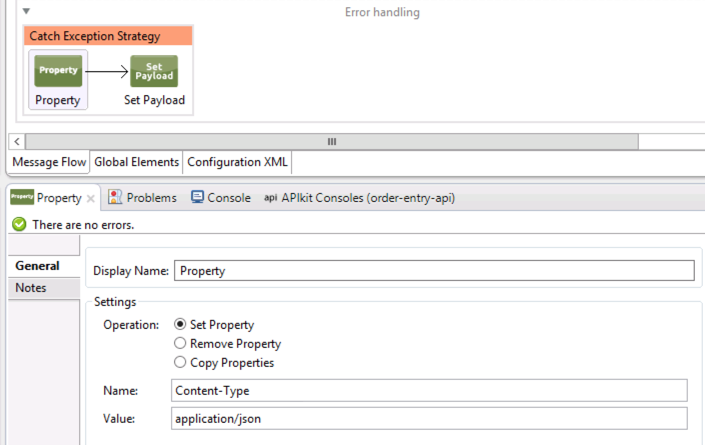


1. Click on it to edit the configuration.
   * Change the display name to **Set error message.**
   * Set value {"message": "The provided order is invalid"}

|  |
| --- |
| **NOTE**: If there are any exceptions, we will return this custom message. You can test it passing an invalid order id when test the order status in the API console. |

****

1. Lastly, add a **Property Transformer** before the **Set Payload** to indicate what type of message is being sent back.



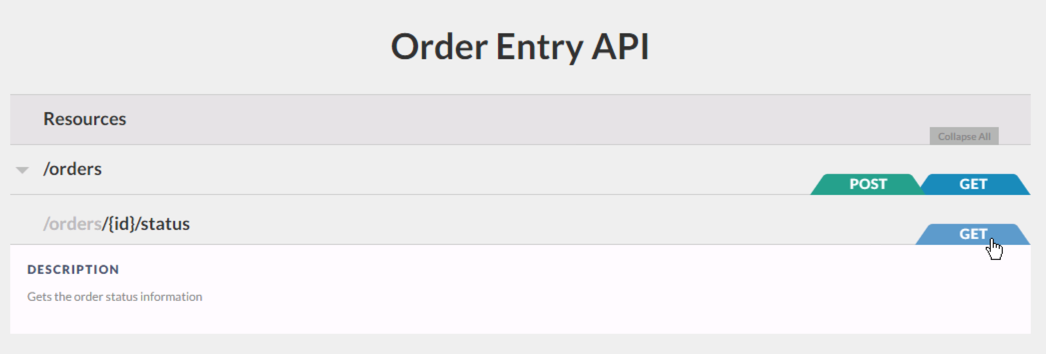
1. Select the **Set Property** operation and enter the following:
   * Name: **Content-Type**
   * Value: **application/json**

This configures the response to use JSON as its Content-Type.

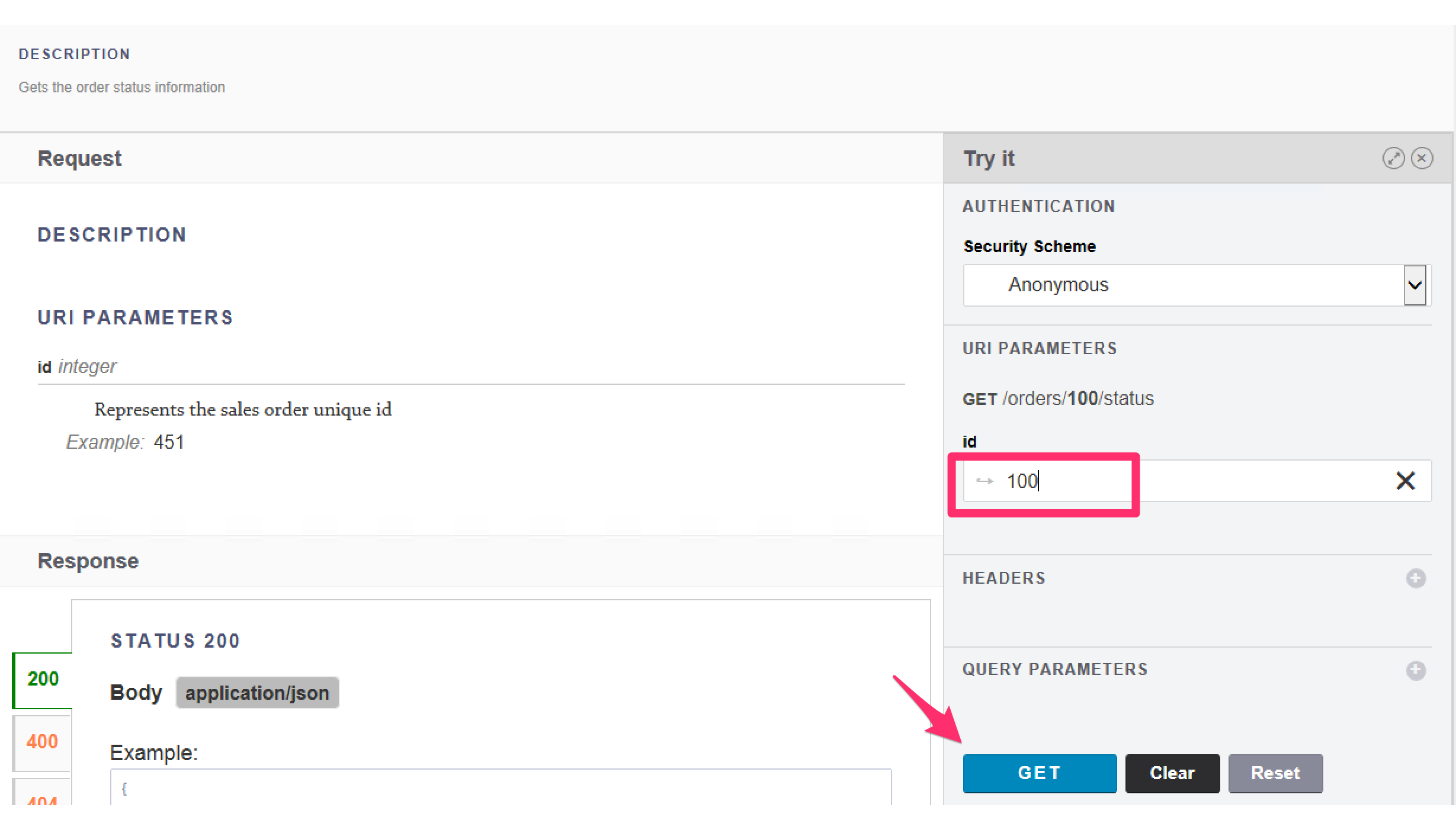
1. Click **Save All**. 

## Step 5: Run and Test the API

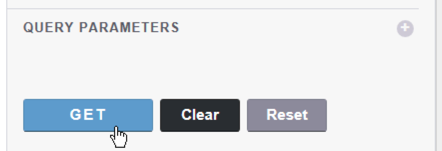
1. Now it's time to run the API in **Anypoint Studio** to see how it all works together. Start the application as done in the previous Lab. (Right-click the application,
2. Run as > Mule Application or simply click the Run icon and choose **order\_entry\_api**:



1. Let's try the GET operation for /orders/{id}/status.



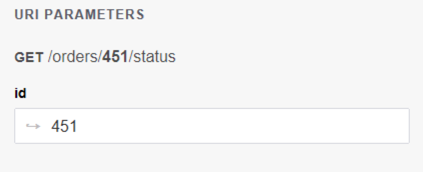
1. Under **Try it**, scroll down on the URI Parameters and put the number 100, which is the ID of an existing order.



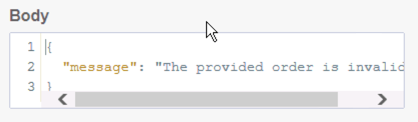
1. Scroll down further then click on the **GET** button.



1. There you will see the order information including the tracking number associated



1. Try passing an invalid order ID like order 451.

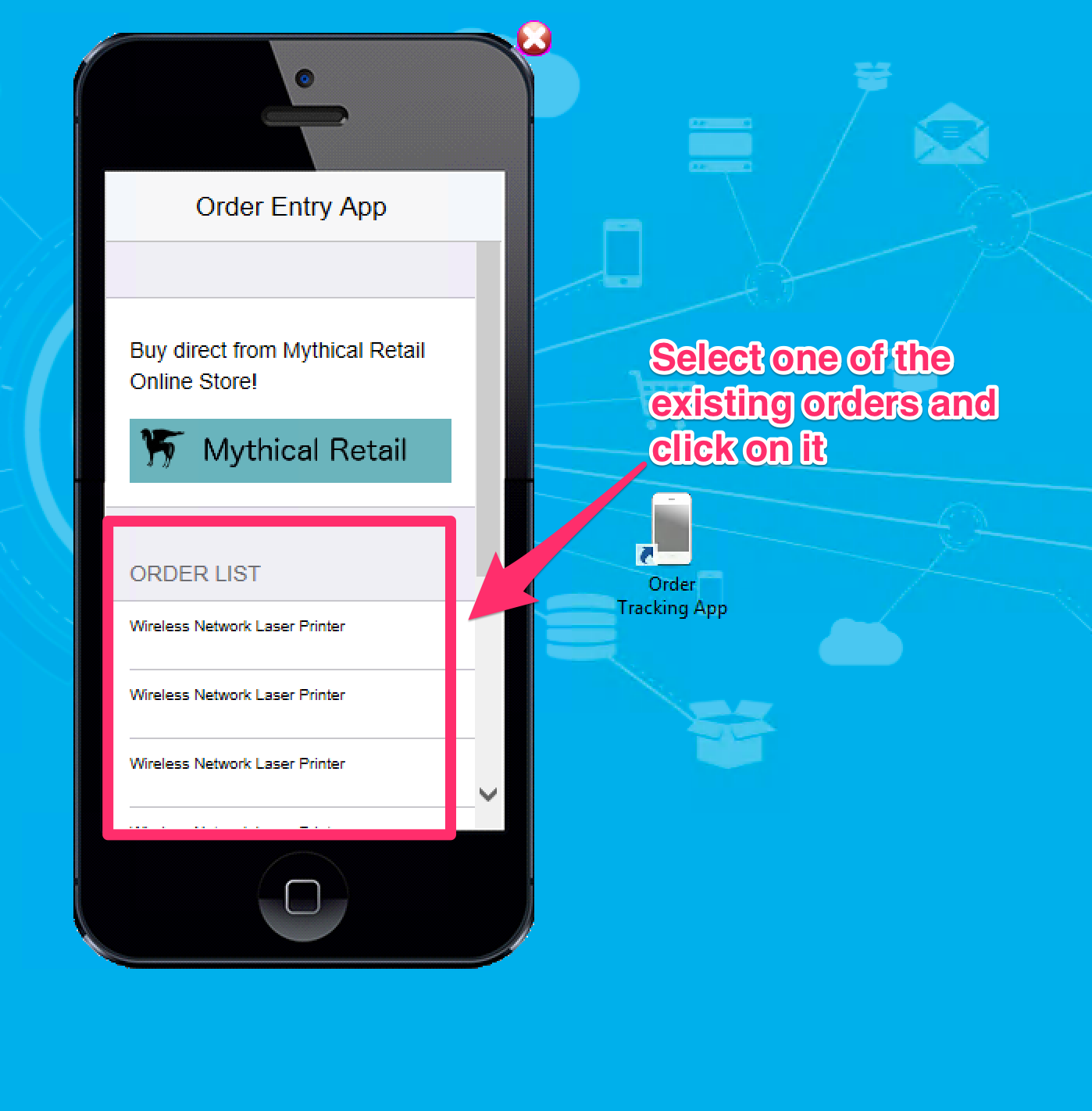


1. You should get a message saying your order is invalid.
2. You have successfully implemented and tested the Order Processing API.

## Step 6: Test from a Mobile Application

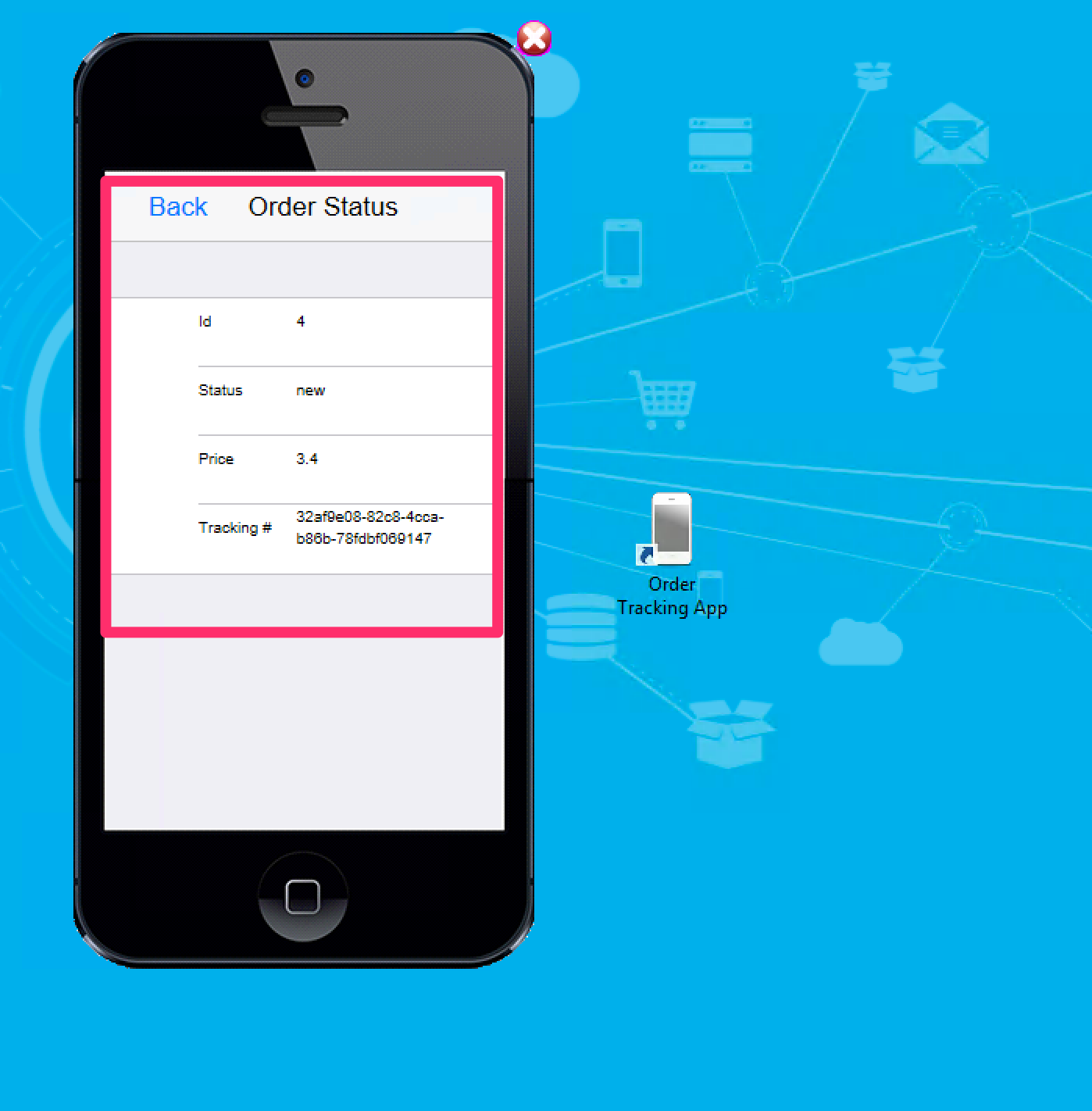
So far, we have designed, implemented the API and then tested it by using the RAML console. Now, we are going to test our API by a mobile application client.



1. Open the **Order Tracking App from the task bar.  
     
   **

Once you have opened the mobile application you will see the list of the orders that already exist and are returned by your API.

1. Then click on one of these orders to see the order status



1. Here you will see the the information of the order: Id, current status, total price and the tracking number with is retrieved from the .NET Web service
2. You have successfully tested your API from a mobile client, congratulations!

## Summary

In this lab, you completed the following steps:

[Step 1: Configure the Database Connector](#_hekv486au28q)

[Step 2: Configure the SOAP Web Services Consumer](#_r8erzm9us1p1)

[Step 3: Create DataMapper transformations](#_7p0upyjbgcnf)

[Step 4: Add an Error Handler](#_nfocr42gt6vh)

[Step 5: Run and Test the API](#_subkunlz4qxi)

[Step 6: Test from a Mobile Application](#_akdm01u2eh4t)

Through this lab, you saw how easy it is to consume legacy systems like databases and SOAP Web Services, allowing us to quickly create a connectivity layer on top of legacy systems. We saw how it queries data from a MySQL database, interact with a SOAP Web service endpoints and transform information between different structures of data. Additionally, we have tested the application from a mobile client in order to close the cycle from the design of the API up to the final consumer.

Go further. Explore the doc for more information on the

* [Database Connector](http://www.mulesoft.org/documentation/display/current/Database+Connector):
* [SOAP Web Service Consumer](http://www.mulesoft.org/documentation/display/current/Web+Service+Consumer)
* [Error Handling](http://www.mulesoft.org/documentation/display/current/Error+Handling)

Congratulations! You have finished Lab 9.

Please update the spreadsheet indicating you have completed Lab 9.