Tutorial - Unit Test for the Resource Controller GET Method

The following implementation example presents unit tests for the GET operation of the resource controller.

Go to the tab "Test Classes" and start with the definition of a test class. Tip: use the code template 'testClass' by typing 'test', pressing CTRL-SPACE, choosing the template and filling the variables while navigating with TAB.

Unit Test Class Definition

The unit test class defines the following member variables:

- *resource*: the resource implementation which is tested
- *request_stub*: a test double for the request
- *response_spy*: a test double for the response

The class has the following test methods:

- *test_get_exist_flight_success*: verifies that reading an existing flight record succeeds with a valid response
- *test_get_nonexist_flight_fail*: verifies that reading a non-existing flight record fails

Furthermore, it has a test *setup* method and the helper method *read_flight_from_db* which reads the first record of the *SFLIGHT* table.

```
class ltcl_res_flights_read definition final for testing
  duration short
  risk level harmless.

  private section.

    data:
        resource     type ref to cl_adt_rest_resource,
        request_stub type ref to cl_adt_rest_request_stub,
        response_spy type ref to cl_adt_rest_response_spy.

    methods:
        setup,
        test_get_exist_flight_success    for testing raising cx_static_check,
        test_get_nonexist_flight_fail    for testing raising cx_static_check,
        read_flight_from_db         returning value(result) type sflight.

endclass.
```

Method SETUP

The setup method creates two test double objects: the request stub and response spy, which are test substitutes for the real request and response objects, respectively.

The request stub supports manipulation of the input request data (like URI attributes, query parameters, and the request body), whereas the response spy allows the inspection of the response data returned by the resource controller.

The tested resource is instantiated as well - you have to modify the code and enter the name of your implementation class.

```
  method setup.

    create object request_stub.
    create object response_spy.
    create object me->resource type zXX_cl_sflight_adt_res_flights. " REPLACE XX with your
credentials!

  endmethod.
```

Method *test_get_existing_flight_success*

The first test method verifies that reading an existing flight record works correct. The part for reading an existing flight record is done in a separate helper method *read_flight_from_db*:

```
  method read_flight_from_db.

    data flights type standard table of sflight.

    select * from sflight into table flights up to 1 rows.
    if sy-subrc <> 0.
      cl_abap_unit_assert=>fail( msg = 'Precondition failed: no flight record found in database' ).
    endif.
    read table flights index 1 into result.

  endmethod.
```

The test uses the key components of the flight record read from the database to perform a GET request on the resource controller. To achieve that, the corresponding key values are set in the request stub as URI attributes. Then the request stub is passed to the resource implementation's *get* method. As second parameter, the *response_spy* instance is passed to the *get* method. This response implementation just records the result of the method call. It allows to retrieve the unserialized result data which is then compared with the expected result. The comparision on the structured data is much more robust and easier to analyze than comparison of the serialized binary data would be. The test verifies if the correct content handler has been set. The serialization of the data is a concern of the content handler and therfore not covered by this unit test.

The code for the test method looks as follows:

```
  method test_get_exist_flight_success.

    data:
        exp_flight type sflight,
        act_flight type sflight,
        connid_s type string,
        fldate_s type string,
        act_content_handler type ref to if_adt_rest_content_handler.

*   get an existing flight first
    exp_flight = me->read_flight_from_db( ).

*   prepare the request's URI attributes
    request_stub->set_uri_attribute( name = 'carrier_id' value = exp_flight-carrid ).
    request_stub->set_uri_attribute( name = 'connection_id' value = exp_flight-connid ).
    request_stub->set_uri_attribute( name = 'flight_date' value = exp_flight-fldate ).

*   call the resource controller method
    resource->get( request = request_stub response = response_spy ).

*   verify the results
    response_spy->get_body_data( importing data = act_flight ).

    cl_abap_unit_assert=>assert_equals( exp = exp_flight act = act_flight ).

    act_content_handler = response_spy->get_content_handler( ).
    cl_abap_unit_assert=>assert_bound( msg = 'No content handler set in response' act =
act_content_handler ).

  endmethod.
```

Method *test_get_nonexist_flight_fail*

The second test method covers the case that a client requests a flight which does not exist. It is expected that the resource controller sends a response with a "Resource not found" status. The creation of responses in error situation is implemented by the framework - the resource controller only has to raise the correct exception, in this case an exception of class *cx_adt_res_not_found*.

```
  method test_get_nonexist_flight_fail.
    data:
        response_status type i.

*   prepare the request's URI attributes with values for a probably not existing record
    request_stub->set_uri_attribute( name = 'carrier_id' value = 'YY' ).
    request_stub->set_uri_attribute( name = 'connection_id' value = '0000' ).
    request_stub->set_uri_attribute( name = 'flight_date' value = '18800101' ).

*   call the resource controller method - we expect that a cx_adt_res_not_found exception is raised
    try.
        resource->get( request = request_stub response = response_spy ).
        cl_abap_unit_assert=>fail( msg = 'Expected exception cx_adt_res_not_found not raised').
      catch cx_adt_res_not_found.  " OK, expected exception
    endtry.

  endmethod.
```

Testing the Content Handler

If the content handler is implemented locally, it makes sense to test it separately. Our example is just very simple but it gives you a guideline for how to implement a real test for a real content handler.

```
class ltcl_flight_content_handler definition final for testing
  duration short
  risk level harmless.

  private section.
    methods:
      serialize_successful for testing raising cx_static_check.
endclass.


class ltcl_flight_content_handler implementation.

  method serialize_successful.

    data:
      content_handler type ref to if_adt_rest_content_handler,
      flight type sflight,
      entity type ref to if_rest_entity,
      act_entity_data type string.

    flight-carrid = 'UT'.
    flight-connid = '1111'.
    flight-fldate = '20101206'.

    create object entity type cl_rest_entity.
    create object content_handler type lcl_flight_content_handler.

    content_handler->serialize( data = flight response_entity = entity ).

    act_entity_data = entity->get_string_data( ).

    cl_abap_unit_assert=>assert_not_initial( act = act_entity_data ).

  endmethod.

endclass.
```