WEEK 4 UNIT 3
WRITING UNIT TESTS WITH QUNIT

Please perform the exercises below in your app project as shown in the video.

Table of Contents

Preview

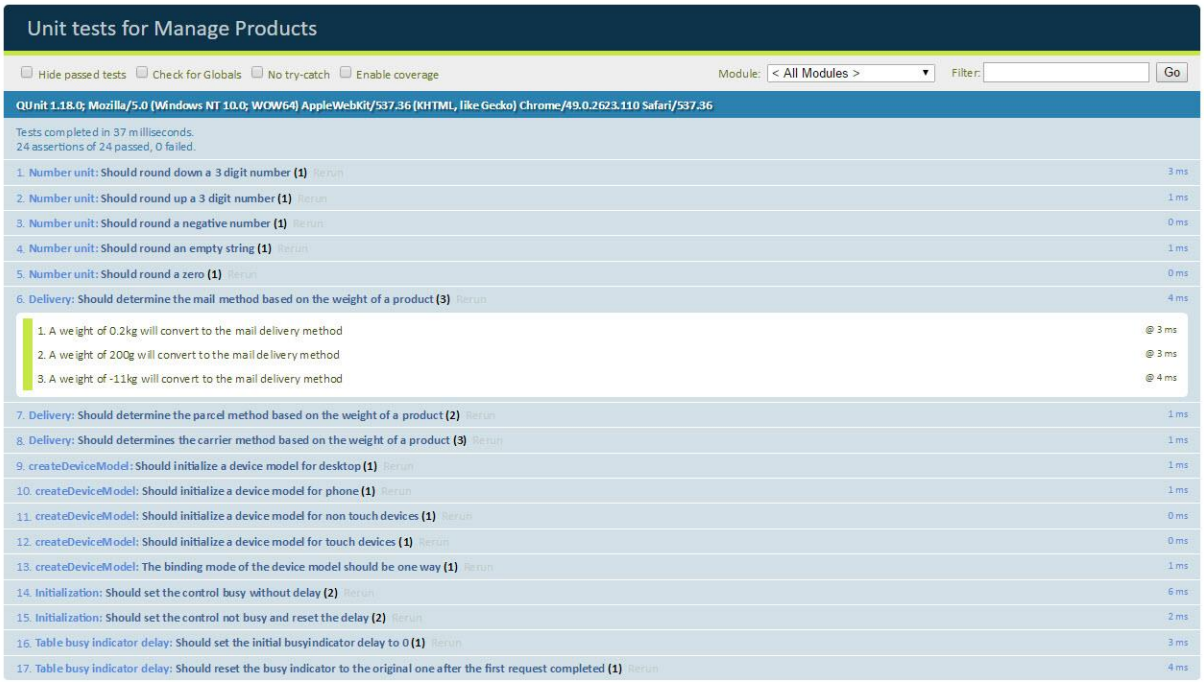


Figure 1 – Unit test page for the app

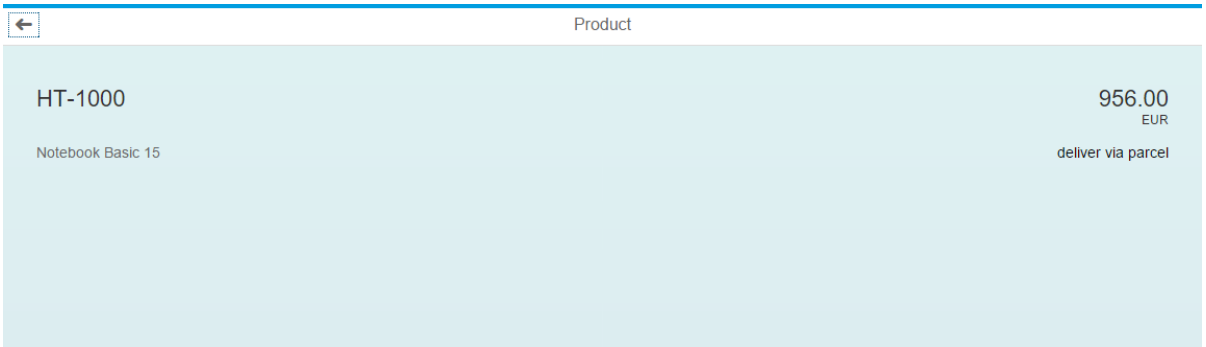


Figure 1 - The formatter added to the manage products object page

1 ADD THE FORMATTER TO THE PROJECT

webapp/model/formatter.js

```
sap.ui.define([] , function () {
    "use strict";

    return {

      numberUnit : function (sValue) {
        …
      },

      /**
       * @public
       * Determines a delivery method based on the weight of a product
       * @param {string} sMeasure the measure of the weight to be
formatted
       * @param {integer} iWeight the weight to be formatted
       * @returns {string} sValue the delivery method
       */
      delivery: function(sMeasure, iWeight) {
        var oResourceBundle = this.getModel("i18n").getResourceBundle(),
          sResult = "";

        if(sMeasure === "G") {
          iWeight = iWeight / 1000;
        }
        if (iWeight < 0.5) {
          sResult = oResourceBundle.getText("formatterMailDelivery");
        } else if (iWeight < 5) {
          sResult = oResourceBundle.getText("formatterParcelDelivery");
        } else {
          sResult = oResourceBundle.getText("formatterCarrierDelivery");
        }

        return sResult;
      },

    …
    };

  }
);
```

Copy the formatter from week 2 unit 2 or the code above to add the delivery formatter to the manage products app project. We have added JSDoc documentation and removed the `this.getView()` call as there is a shortcut in the `BaseController.js` file that we can use in the template to make our code even shorter.

Note: Remove the .getView() call!
Do not forget to remove this call if you copy the code from week 2 unit 2 or the tests that we write later on will not work!

You should be familiar with this code already. The formatter determines a delivery method (mail, parcel, or carrier) based on the weight and measure of the product from the model. This is the formatter that we want to write a unit test for in this exercise – our unit under test.

We will write the tests using a pattern called “Make it work, make it nice”. In a first step we get the test to work and run successfully and in a second step we worry about writing elegant and maintainable test code and covering all the paths of the formatter. This helps you structuring your code and writing minimal test cases.

2 MAKE IT WORK

webapp/test/unit/model/formatter.js

```
sap.ui.define([
    "opensap/manageproducts/model/formatter",
    "test/unit/helper/FakeI18nModel",
    "sap/ui/thirdparty/sinon",
    "sap/ui/thirdparty/sinon-qunit"
 ], function (formatter, FakeI18n) {
    "use strict";

    QUnit.module("Number unit");

    …
    QUnit.test("Should round a zero", function (assert) {
      numberUnitValueTestCase.call(this, assert, "0", "0.00");
    });

    QUnit.module("Delivery");

    QUnit.test("Should determine a delivery method based on the weight of
a product", function (assert) {
      var oControllerStub = {
        getModel: sinon.stub().withArgs("i18n").returns(new FakeI18n({
          formatterMailDelivery : "mail"
        }))
      };
      var fnIsolatedFormatter = formatter.delivery.bind(oControllerStub);

      assert.strictEqual(fnIsolatedFormatter("KG", 0.2), "mail");
      assert.strictEqual(fnIsolatedFormatter("G", 200), "mail");
    });
 }
);
```

Unit tests are running the “unit under test” in an isolated environment. Our formatter is already loaded as a dependency by the template code. But it does not have access to the view, its controller, or the models set on the view. That is why we also load a `FakeI18nModel` and the `SinonJS` dependencies.

We do not want to test the controller, the view, or the model functionality. So we first remove the dependencies by replacing these calls with empty hulls with the help of `SinonJS` and its stub method. The `FakeI18nModel` is part of the template and simply mocks a resource bundle so that we do not have to worry about translation texts while testing. You can call the constructor with a configuration object that contains any key value pair.

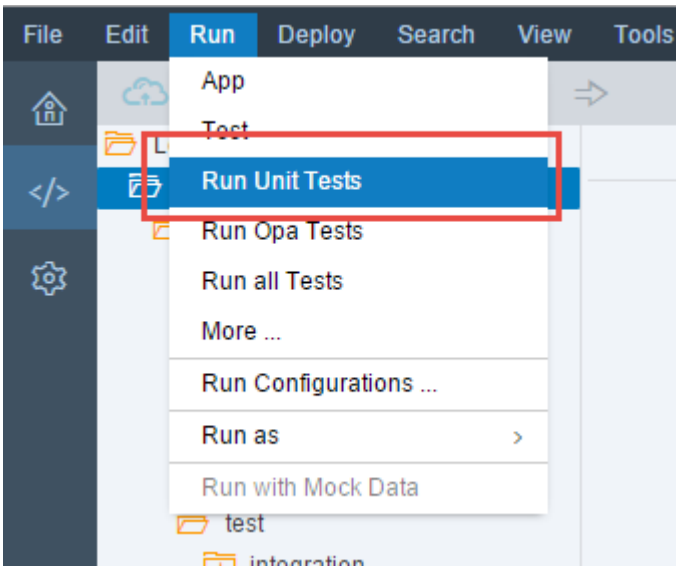
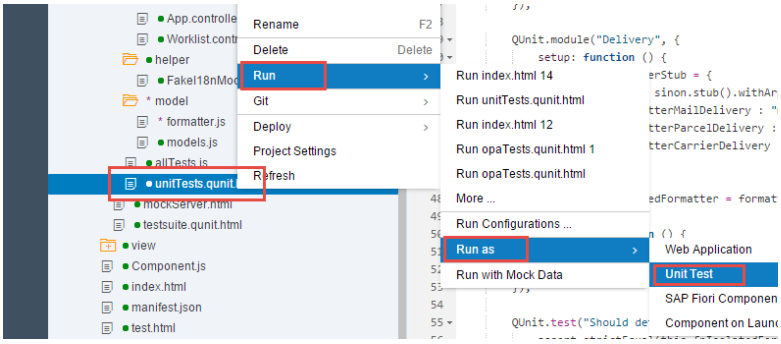
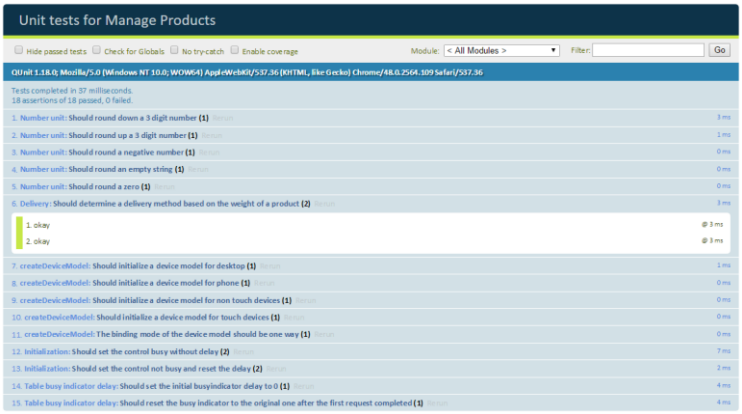
Then we bind our stub to the `delivery` formatter by calling the `bind` function of JavaScript.
The `this` keyword inside the `delivery` function is now bound to our controller stub when the function is invoked using variable `fnIsolatedFormatter`, so calls to `this.getModel()` actually call the stub. We can still pass arguments as required.

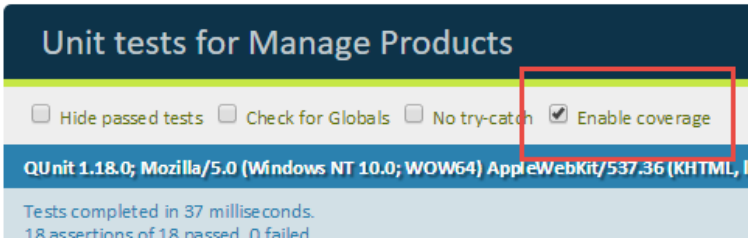
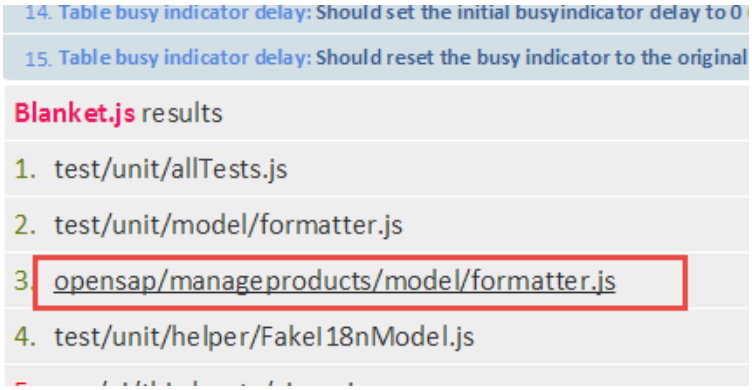
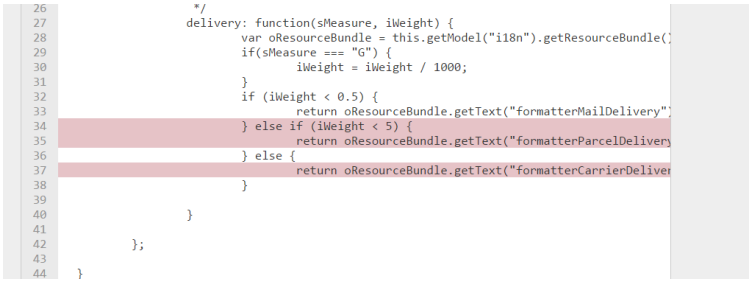
Finally we perform our first assertions. We first check the “mail” branch of the formatter logic by invoking the isolated formatter function with the values that we expect in the data model (`KG, 0.2`). Then we add an assertion for the conversion special case (1000g → 1kg).

We strictly compare the result of the formatter function with the hard-coded strings that we expect from the fake i18n model. This way we do not have to test the real i18n texts and can be sure that the logic of the formatter is correct.

3 RUN THE UNIT TESTS

Now run the unit tests by following the steps below

Explanation	Screenshot
1. Select the project folder “ManageProducts” and choose the run configuration “Run > Unit Tests”.	
2. **Alternative:** Select the file “unitTests.qunit.html”, right click on it and choose the “Run > Run as > Unit Test” option.	
3. Check the QUnit result page that opens in a new window and expand the Delivery formatter test case.	

Explanation	Screenshot
4. Tick the checkbox “Enable coverage” to show the code coverage report of your tests.	
5. Click on the formatter file in the result list.	
6. Note that there are still paths that we did not cover yet in our “Make it work” section. We will now “make it nice”.	

4 MAKE IT NICE

webapp/test/unit/model/formatter.js

```
sap.ui.define([
    "opensap/manageproducts/model/formatter",
    "test/unit/helper/FakeI18nModel",
    "sap/ui/thirdparty/sinon",
    "sap/ui/thirdparty/sinon-qunit"
 ], function (formatter, FakeI18n) {
    "use strict";

    QUnit.module("Number unit");

    …

    QUnit.module("Delivery", {
       setup: function () {
          var oControllerStub = {
             getModel: sinon.stub().withArgs("i18n").returns(new FakeI18n({
                formatterMailDelivery : "mail",
                formatterParcelDelivery : "parcel",
                formatterCarrierDelivery : "carrier"
             }))
          };
          this.fnIsolatedFormatter =
formatter.delivery.bind(oControllerStub);
       },
       teardown: function () {
          this.fnIsolatedFormatter = null;
       }
    });

    QUnit.test("Should determine the mail method based on the weight of a
product", function (assert) {
       var oControllerStub = {
          getModel: sinon.stub().withArgs("i18n").returns(new FakeI18n({
             formatterMailDelivery : "mail"
          }))
       };
       var fnIsolatedFormatter = formatter.delivery.bind(oControllerStub);

       assert.strictEqual(this.fnIsolatedFormatter("KG", 0.2), "mail", "A
weight of 0.2kg will convert to the mail delivery method");
       assert.strictEqual(this.fnIsolatedFormatter("G", 200), "mail", "A
weight of 200g will convert to the mail delivery method");
       assert.strictEqual(this.fnIsolatedFormatter("G", -11), "mail", "A
weight of -11kg will convert to the mail delivery method");
    });

    QUnit.test("Should determine the parcel method based on the weight of
a product", function (assert) {
       assert.strictEqual(this.fnIsolatedFormatter("G", 500), "parcel", "A
weight of 500g will convert to the parcel delivery method");
       assert.strictEqual(this.fnIsolatedFormatter("KG", 3), "parcel", "A
weight of 3kg will convert to the parcel delivery method");
    });
```

```
    QUnit.test("Should determines the carrier method based on the weight
of a product", function (assert) {
      assert.strictEqual(this.fnIsolatedFormatter("KG", 23), "carrier", "A
weight of 23kg will convert to the carrier delivery method");
      assert.strictEqual(this.fnIsolatedFormatter("KG", 5), "carrier", "A
weight of 5kg will convert to the carrier delivery method");
      assert.strictEqual(this.fnIsolatedFormatter("foo", "bar"),
"carrier", "Invalid values will convert to the carrier delivery method");
    });

  }
);
```

Now it is time to fine-tune the unit test that we have written before and to cover all paths of the formatter.

We want to create a QUnit test for each of the delivery methods and do not want to duplicate the code to isolate the formatter function. Therefore the QUnit module gets a configuration object as a second parameter with a setup and teardown function. These functions will be executed before and after each test.

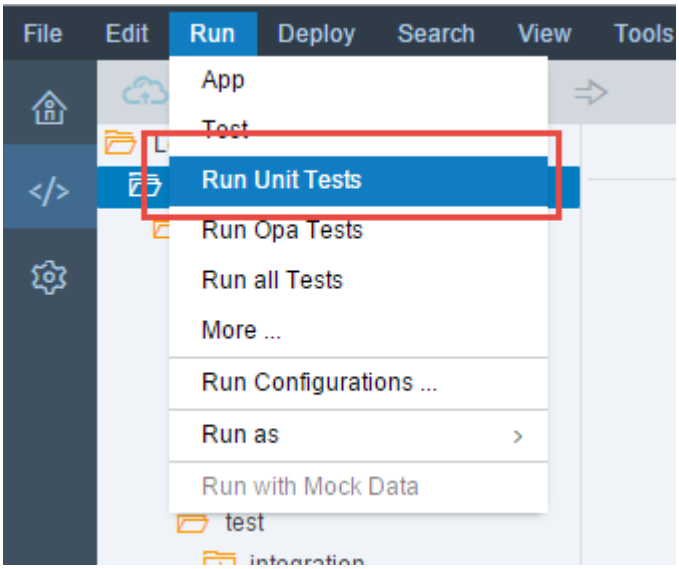
Here we can put the code from the previous step and replace the `fnIsolatedFormatter` with `this.fnIsolatedFormatter`. We also replace the calls to the isolated formatter with this new syntax and add a third test case for an invalid negative value.

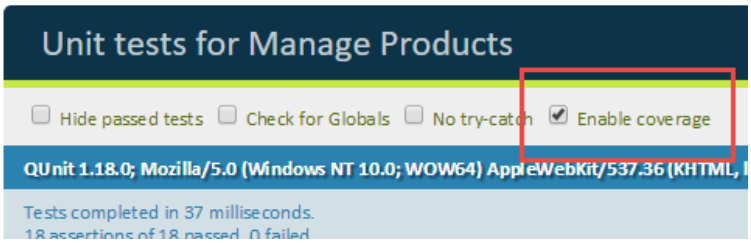
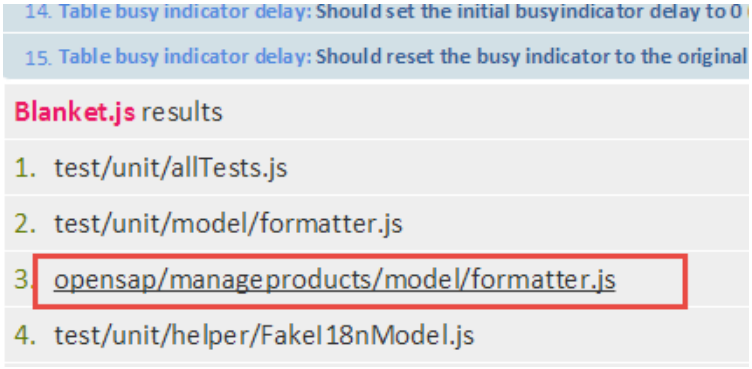
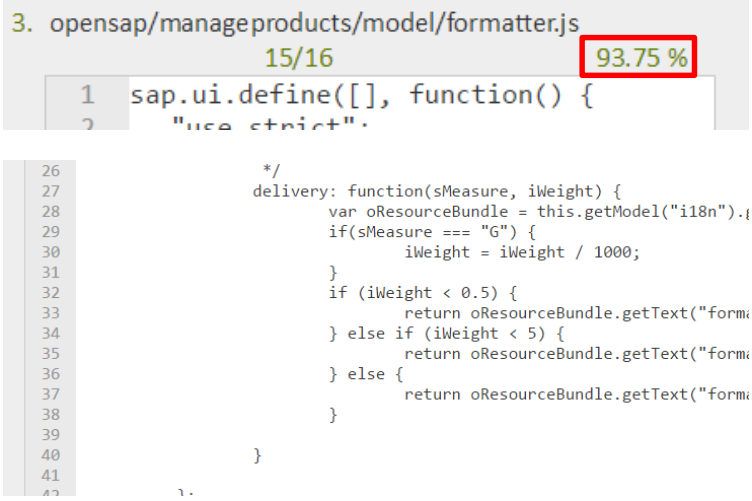
The `QUnit.test` functions now only contain assertions and are very simple. But the assertions lack context information so we add a meaningful message as the third parameter of each assertion.

To cover all logical paths of the formatter, we add two additional QUnit test cases and check the parcel and carrier methods there. Edge cases and invalid arguments are very likely to cause logical errors, so we make sure to also add assertions for those.

5 RUN THE UNIT TESTS AGAIN

Now run the unit tests again by following the steps below

Explanation	Screenshot
1. Select the project folder “ManageProducts” and choose the run configuration “Run > Unit Tests”.	
2. Check the QUnit result page that opens in a new window. There are three “Delivery” test cases now, and each assertion has a meaningful message	

Explanation	Screenshot
3. Tick the checkbox “Enable coverage” to show the code coverage report of your tests	
4. Click on the formatter file in the result list	
5. Note that we have now achieved 100% test coverage in our `delivery` formatter (93,75% for the formatter file overall). There are no more red lines.	

6 USE THE FORMATTER IN THE VIEW

webapp/view/Object.view.xml

```
<mvc:View …>

 …
      <ObjectHeader …>
         <attributes>
            <ObjectAttribute text="{Name}"/>
         </attributes>
         <statuses>
            <ObjectStatus text="{
               parts: [
                  {path: 'WeightUnit'},
                  {path: 'WeightMeasure'}
               ],
               formatter : '.formatter.delivery'
            }"/>
         </statuses>
      </ObjectHeader>
    …

</mvc:View>
```

Add a new aggregation `statuses` right after the `attributes` aggregation of the `ObjectHeader` in the object view. Copy the code from week 2 unit 2 or add the `ObjectStatus` control above to display the delivery formatter on the object page.

webapp/i18n/i18n.properties

```
#~~~ Worklist View ~~~~~~~~~~~~~~~~~~~~~~~~~~
…

#XTIT: Mail delivery formatter text
formatterMailDelivery=deliver via mail

#XTIT: Parcel delivery formatter text
formatterParcelDelivery=deliver via parcel

#XTIT: Carrier delivery formatter text
formatterCarrierDelivery=deliver via carrier
…
```

Copy the i18n texts from the project of week 2 unit 2 or add the three texts above

Now run the app, navigate to the object page and verify that the formatter is displayed correctly.

Related Information

QUnit Testing Fundamentals
QUnit Home Page
Sinon.JS Home Page