AMIS SIG/Oracle Developer MeetUp Workshop Creating Oracle JET Composite Components with OpenLayers GIS library

23rd January 2018

Oracle JET is a toolkit for the creation of rich web applications. Many applications will have location-related aspects. Such applications can benefit from advanced map capabilities – for presenting data in maps, allowing users to interact with maps in order to formulate queries, navigate to relevant details or manipulate data. OpenLayers is one of the most prominent open source JavaScript libraries for working with maps in web applications. It provides an API for building rich web-based geographic applications similar to Google Maps and Bing Maps. One of the geographic data providers that OpenLayers works well with is Open Street Map (OSM) – also fully open source.

In this workshop, you will get introduced to OpenLayers in general and more specifically with the first steps to integrate OpenLayers and OSM in Oracle JET.

Once you have become a little acquainted with the combination of OpenLayers and JET, we will go to the next level: the creation of JET Composite Component based on OpenLayers. Composite Components are a crucial mechanism in Oracle JET. Aligned with the W3C Web Components standard, composite components are standardized, encapsulated, easily reusable building blocks that will help developers be more productive. Additionally, composite components can be plugged into the Oracle Visual Builder Cloud Service development framework to add tailor made functionality to the low code component palette for use in declaratively developed applications that can be stand-alone or extensions of Oracle SaaS applications.

You will learn in a step by step approach how to create Composite Components in general – with properties, methods, events, embedded resources – and how to create one for OpenLayers in particular.: the input-country component.

This component allows users to select a country from a map, displayed in a popup. The map can be panned and zoomed in and out across many levels.

The component supports two-way data binding, callback functions and publication of events. It can be packaged, published and shipped and reused in any JET 4 application as well as in Visual Builder Cloud applications.

1. Prepare your local environment

We will work with Oracle JET, release 4. You can work in a Virtual Machine, a Docker Container, your bare operating system. You will need a text editor – Visual Studio Code is a very popular one (https://code.visualstudio.com/) .

If you do not already have Oracle JET 4.x running in your environment, then execute the first step in http://www.oracle.com/webfolder/technetwork/jet/globalGetStarted.html to get started with Oracle JET 4 on your machine.

Download the OpenLayers distribution – a zip-file with the combined JavaScript and CSS files for OpenLayers (4.x) from https://github.com/openlayers/openlayers/releases/ .

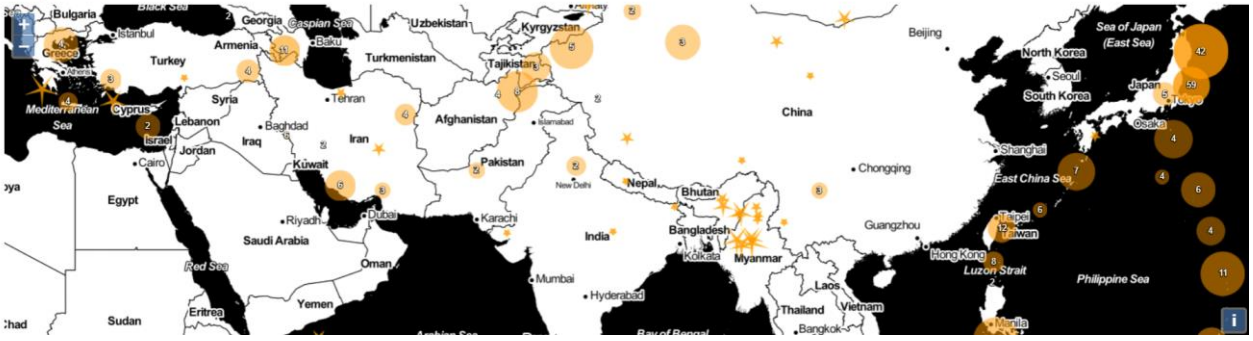
2. Explore OpenLayers

OpenLayers provides many features for presenting and manipulating geo data in many different ways and contexts. At http://openlayers.org/en/latest/examples/ - you will find a number of interesting examples that give you some idea of what can be done – and how it can be done.

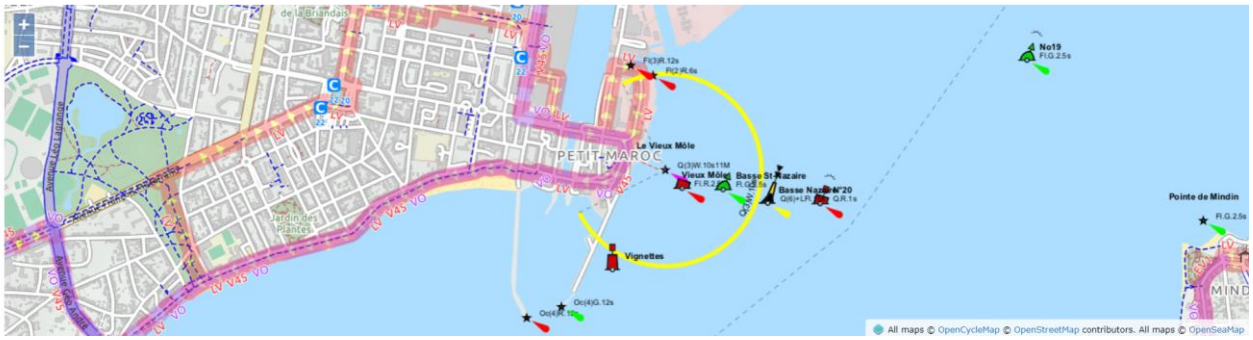
Before actually coding with OpenLayers yourself, explore these examples to get a little taste.

Stamen Tiles

Earthquake Clusters

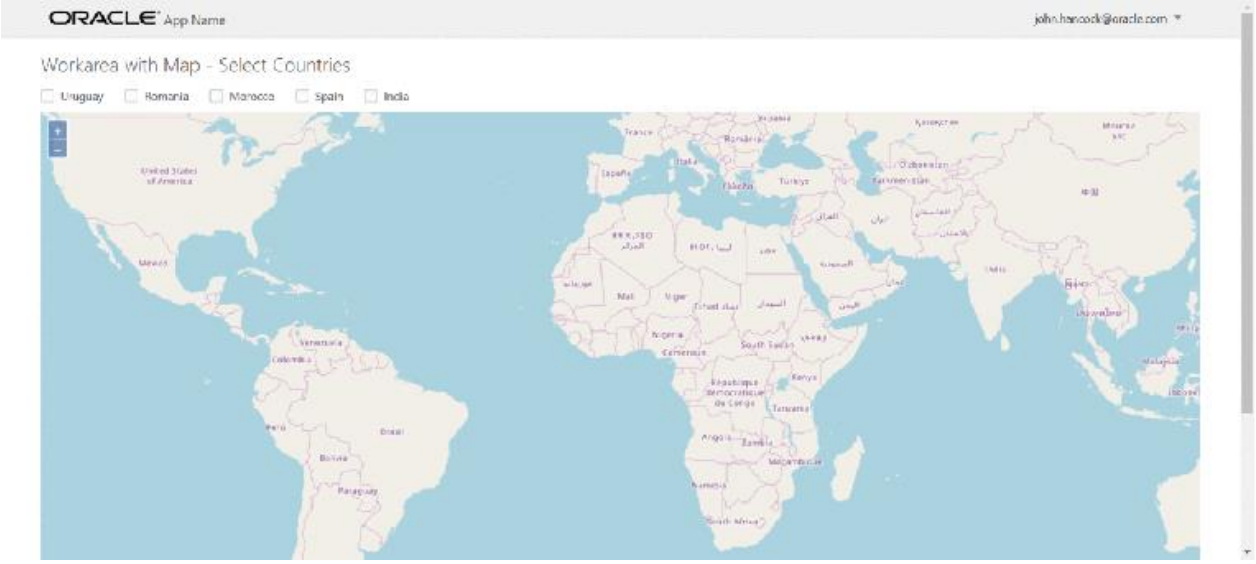


Localized OpenStreetMap



3. Embedding OpenLayers in Oracle JET for Advanced Maps and GIS style User Interfaces

In a few simple steps, you will create the JET application illustrated below –a mix of a JET Checkbox Set where countries can be selected and an OpenLayers map that is manipulated from JavaScript to show (and hide) markers for the countries that are selected (and deselected).



Source code for this lab can be downloaded from GitHub: [https://github.com/lucasjellema/jet-and-openlayers .](https://github.com/lucasjellema/jet-and-openlayers)

The steps are:

- create new JET application (for example with JET CLI)
- download OpenLayers distribution and add to JET application's folders
- configure the JET application for the OpenLayers library
- add a DIV as map container to the HTML file
- add the JavaScript code to initialize and manipulate the map to the ViewModel JS file

In more detail:

a. Create a new Oracle JET application

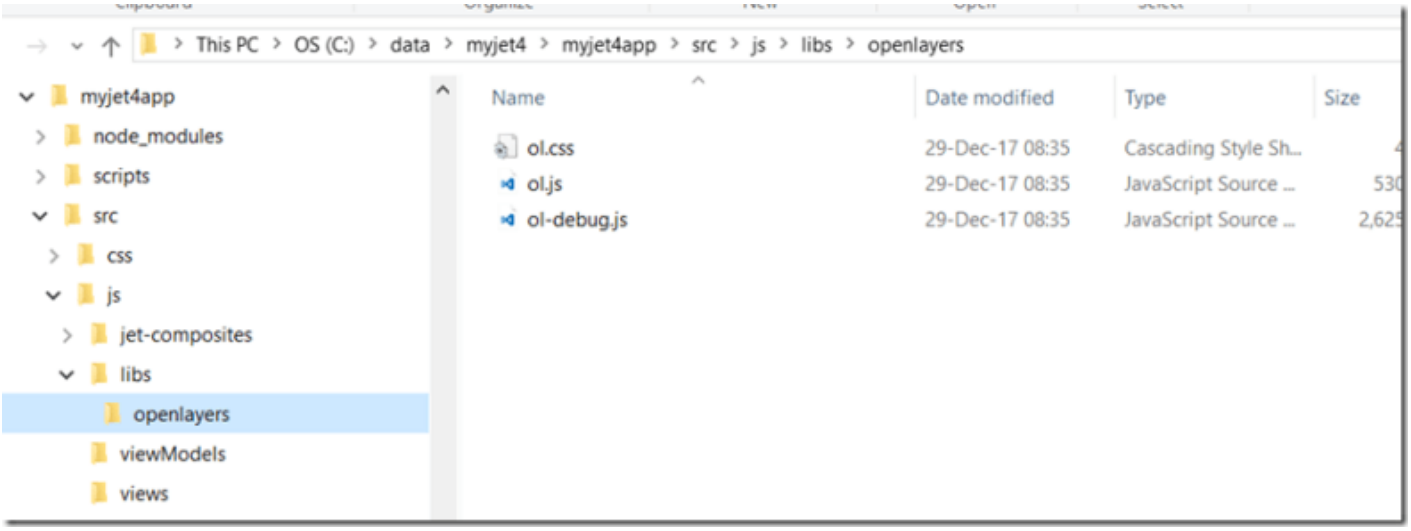
Create a new folder for your JET application. Use

```
ojet create projectname –template=basic
```

to create the new JET application

b. Download OpenLayers Distribution and Add to the JET Application

In the JET application's js/libs directory, create a new directory *openlayers* and add the resources extracted from the OpenLayers zip file to it.



c. Configure the JET application for the OpenLayers library

In the *js* directory update the *js/main-release-paths.json* file to include the new library.

```
  "ol": "libs/openlayers/ol-debug",

  "olcss": "libs/openlayers/ol.css"

}
```

In your RequireJS bootstrap file, typically *main.js*, add a link to the new file in the path mapping section and include the new library in the *require()* definition.

```
paths:

//injector:mainReleasePaths

{

  ...

  'ol': 'libs/openlayers/ol-debug'

}

//endinjector
```

In the same file add a Shim Configuration for OpenLayers

```
// Shim configurations for modules that do not expose AMD

  shim:

  {

    'jquery':
```

```
    {
      exports: ['jQuery', '$']
    }
    ,'ol':
    {
      exports: ['ol']
    }
  }
}
```

Finally, add module *'ol'* to the call to require ad as parameter in the callback function (if you want to perform special initialization on this module):

```
require(['ojs/ojcore', 'knockout', 'appController','ol', 'ojs/ojknockout', 'ojs/ojbutton',
'ojs/ojmenu','ojs/ojmodule'],
  function (oj, ko, app, ol) { // this callback gets executed when all required modules are
    ...
```

Now to actually include map in a View in the JET application:

d. Add a DIV as map container to the HTML file

The View contains a DIV that will act as the container for the map. It also contains a Checkbox Set with checkboxes for five different countries. The checkbox set is data bound to the ViewModel; any change in selection status will trigger an event listener. Additionally, the *currentCountries* variable in the ViewModel is updated with any change by the user.

```
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/openlayers/4.6.4/ol-debug.css" />

<h2>Workarea with Map - Select Countries</h2>

<div id="div1">

        <oj-checkboxset id="countriesCheckboxSetId" labelled-by="mainlabelid"
class="oj-choice-direction-row" value="{{currentCountries}}"

                on-value-changed="[[selectionListener]]">

                <oj-option id="uruopt" value="uy">Uruguay</oj-option>

                <oj-option id="romopt" value="ro">Romania</oj-option>

                <oj-option id="moropt" value="ma">Morocco</oj-option>
```

```
                <oj-option id="spaopt" value="es">Spain</oj-option>

                <oj-option id="indopt" value="in">India</oj-option>

        </oj-checkboxset>

        <br/>

</div>

<div id="map2" class="map"></div>
```

e. Add JavaScript code to initialize and manipulate the map to the ViewModel JS file

Add OpenLayers dependency in workArea.js:

```
define(
    ['ojs/ojcore', 'knockout', 'jquery', 'ol', 'ojs/ojknockout',
'ojs/ojinputtext', 'ojs/ojbutton', 'ojs/ojlabel', 'ojs/ojcheckboxset'],
    function (oj, ko, $, ol) {
        'use strict';
        function WorkAreaViewModel() {
            var self = this;
```

The following code defines a countryMap – a collection of five elements (one for each of five countries) that hold longitude and lattitude for each country, as well as a display name and country code (also the key in the map). Subsequenty, an OpenLayers *feature* is created for each country, and referenced from the countryMap element for later use.

```
self.currentCountries = ko.observableArray([]);



self.countryMap = {};

self.countryMap['in'] = { "place_id": "177729185", "licence": "Data ©
OpenStreetMap contributors, ODbL 1.0. http://www.openstreetmap.org/copyright",
"osm_type": "relation", "osm_id": "304716", "boundingbox": ["6.5546079",
"35.6745457", "68.1113787", "97.395561"], "lat": "22.3511148", "lon":
"78.6677428", "display_name": "India", "class": "boundary", "type":
"administrative", "importance": 0.3133568788165, "icon":
"http://nominatim.openstreetmap.org/images/mapicons/poi_boundary_administrativ
e.p.20.png", "address": { "country": "India", "country_code": "in" } };

self.countryMap['es'] = { "place_id": "179962651", "licence": "Data ©
OpenStreetMap contributors, ODbL 1.0. http://www.openstreetmap.org/copyright",
"osm_type": "relation", "osm_id": "1311341", "boundingbox": ["27.4335426",
"43.9933088", "-18.3936845", "4.5918885"], "lat": "40.0028028", "lon": "-
4.003104", "display_name": "Spain", "class": "boundary", "type":
```

```
"administrative", "importance": 0.22447060272487, "icon":
"http://nominatim.openstreetmap.org/images/mapicons/poi_boundary_administrativ
e.p.20.png", "address": { "country": "Spain", "country_code": "es" } };

self.countryMap['ma'] = { "place_id": "217466685", "licence": "Data ©
OpenStreetMap contributors, ODbL 1.0. http://www.openstreetmap.org/copyright",
"osm_type": "relation", "osm_id": "3630439", "boundingbox": ["21.3365321",
"36.0505269", "-17.2551456", "-0.998429"], "lat": "31.1728192", "lon": "-
7.3366043", "display_name": "Morocco", "class": "boundary", "type":
"administrative", "importance": 0.19300832455819, "icon":
"http://nominatim.openstreetmap.org/images/mapicons/poi_boundary_administrativ
e.p.20.png", "address": { "country": "Morocco", "country_code": "ma" } }

self.countryMap['ro'] = { "place_id": "177563889", "licence": "Data ©
OpenStreetMap contributors, ODbL 1.0. http://www.openstreetmap.org/copyright",
"osm_type": "relation", "osm_id": "90689", "boundingbox": ["43.618682",
"48.2653964", "20.2619773", "30.0454257"], "lat": "45.9852129", "lon":
"24.6859225", "display_name": "Romania", "class": "boundary", "type":
"administrative", "importance": 0.30982735099944, "icon":
"http://nominatim.openstreetmap.org/images/mapicons/poi_boundary_administrativ
e.p.20.png", "address": { "country": "Romania", "country_code": "ro" } };

self.countryMap['uy'] = { "place_id": "179428864", "licence": "Data ©
OpenStreetMap contributors, ODbL 1.0. http://www.openstreetmap.org/copyright",
"osm_type": "relation", "osm_id": "287072", "boundingbox": ["-35.7824481", "-
30.0853962", "-58.4948438", "-53.0755833"], "lat": "-32.8755548", "lon": "-
56.0201525", "display_name": "Uruguay", "class": "boundary", "type":
"administrative", "importance": 0.18848351906936, "icon":
"http://nominatim.openstreetmap.org/images/mapicons/poi_boundary_administrativ
e.p.20.png", "address": { "country": "Uruguay", "country_code": "uy" } };



for (const c in self.countryMap) {

    // create a feature for each country in the map

    var coordinates = ol.proj.transform([1 * self.countryMap.lon, 1 *
self.countryMap.lat], 'EPSG:4326', 'EPSG:3857');

    var featurething = new ol.Feature({

        name: self.countryMap.display_name,

        geometry: new ol.geom.Point(coordinates)

    });

    self.countryMap.feature = featurething;

}
```

Then add the code to do the initialization of the Map itself – to be performed when the DOM is ready and the DIV target container is available:

```
$(document).ready
    (
    // when the document is fully loaded and the DOM has been initialized
    // then instantiate the map
    function () {
        initMap();
    })

function initMap() {
    self.elem = document.getElementById("text-input");
    self.map = new ol.Map({
        target: 'map2',
        layers: [
            new ol.layer.Tile({
                source: new ol.source.OSM()
            })
        ],
        view: new ol.View({
            center: ol.proj.fromLonLat([-2, -5]),
            zoom: 3
        })
    });
}
```

Also add the code for the selectionListener to be executed whenever countries are selected or deselected.

This code adds OpenLayers *features* for each of the currently selected countries. Next, construct a *layer* which contains these features and has a specific style (red circle with big X) associated with it. Finally, add this layer to the map – to have the features displayed in the web page.

```
// triggered whenever a checkbox is selected or deselected
    self.selectionListener = function (event) {
        console.log("Country Selection Changed");

        var vectorSource = new ol.source.Vector({}); // to hold features for
currently selected countries
        for (var i = 0; i < self.currentCountries().length; i++) {
            // add the feature to the map for each currently selected country
            vectorSource.addFeature(self.countryMap[self.currentCountries()[i]
].feature);
        }//for

        var layers = self.map.getLayers();
        // remove the feature layer from the map if it already was added
        if (layers.getLength() > 1) {
            self.map.removeLayer(layers.item(1));
        }
        //Create and add the vector layer with features to the map
        // define the style to apply to these features: bright red, circle
with radius 10 and a X as (text) content
```

```
        var vector_layer = new ol.layer.Vector({
            source: vectorSource
            ,style: function(feature) {
                var style = new ol.style.Style({
                    image: new ol.style.Circle({
                      radius: 10,
                      stroke: new ol.style.Stroke({
                        color: '#fff'
                      }),
                      fill: new ol.style.Fill({
                        //color: '#3399CC' // light blue
                        color: 'red' // light blue
                    })
                    }),
                    text: new ol.style.Text({
                      text: "X",
                      fill: new ol.style.Fill({
                        color: '#fff'
                      })
                    })
                  });
                  return style;
                }
          } )
        self.map.addLayer(vector_layer);

    }//selectionListener
}
```

f. References

Source code in GitHub Repo: https://github.com/lucasjellema/jet-and-openlayers

Blog article by Enno Schulte (Virtual7) on adding Socket.io as third part library to a JET 3.x application: http://www.virtual7.de/blog/2017/07/oracle-jet-3-add-third-party-libraries-example-socket-io/

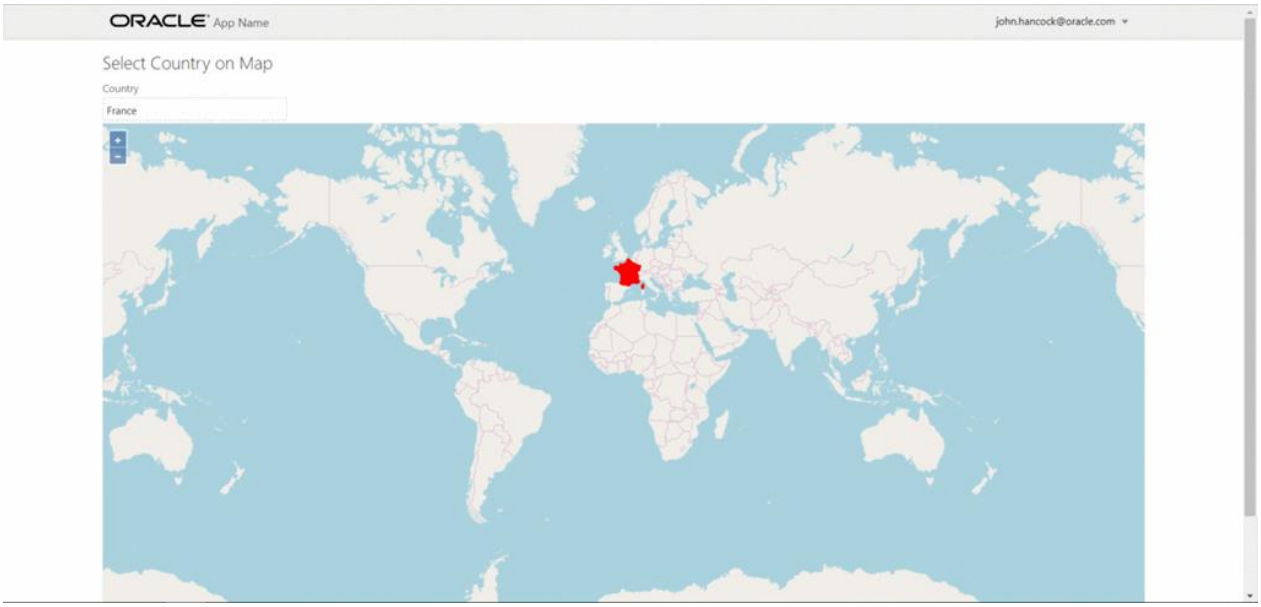
Documentation on adding 3rd party libraries to JET 4.0: https://docs.oracle.com/middleware/jet410/jet/developer/GUID-EC40DF3C-57FB-4919-A066-73E573D66B67.htm#JETDG-GUID-EC40DF3C-57FB-4919-A066-73E573D66B67

OJET Docs Checkbox Set – http://www.oracle.com/webfolder/technetwork/jet/jsdocs/oj.ojCheckboxset.html

4. Using an OpenLayers map to select countries in an Oracle JET application

In this lab, we take the previous one a step further – or actually several steps. By adding interactivity to the map, we allow users to select a country on the world map and we notify JET components of this selection. The synchronization works both ways: if the user types the name of a country in a JET input text component, this country is highlighted on the world map. Note that the map has full zooming and panning capabilities.

Check out this animated gif to see the end result: https://i0.wp.com/technology.amis.nl/wp-content/uploads/2018/01/Webp.net-gifmaker-2.gif?ssl=1



The challenges to address when implementing this functionality:

- add vector layer with countries (features)
- highlight country when mouse is hovering over it
- add *select interaction* to allow user to select a country
- communicate country selection event in *map* to “regular” JET component
- synchronize map with country name typed into JET component

The steps:

- Create mapArea.html with input-text, informational DIV and map container (DIV)
- Create mapArea.js for the mapArea module
- Add a div with data bind for mapArea module to index.html
- Download a file in GEOJSON format with annotated geo-json geometries for the countries of the world
- Initialize the map with two layers – raster OSM world map and vector country shapes

- Add overlay to highlight countries that are hovered over
- Add Select Interaction – to allow selection of a country – applying a bold style to the selected country
- Update JET component from country selection
- Set country selection on map based on value [change]in JET component

Create mapArea.html with input-text, informational DIV and map container (DIV)

```
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/openlayers/4.6.4/ol-debug.css" />

<h2>Select Country on Map</h2>

<div id="componentDemoContent" style="width: 1px; min-width: 100%;">

    <div id="div1">


        <oj-label for="text-input">Country</oj-label>

        <oj-input-text id="text-input" value="{{selectedCountry}}" on-value-
changed="[[countryChangedListener]]"></oj-input-text>

    </div>

</div>

<div id="info"></div>

<div id="map2" class="map"></div>
```

Create ViewModel mapArea.js for the mapArea module

```
define(
    ['ojs/ojcore', 'knockout', 'jquery', 'ol', 'ojs/ojknockout',
'ojs/ojinputtext', 'ojs/ojlabel'],
    function (oj, ko, $, ol) {
        'use strict';
        function MapAreaViewModel() {
            var self = this;

            self.selectedCountry = ko.observable("France");
            self.countryChangedListener = function(event) {
            }
...
      }
```

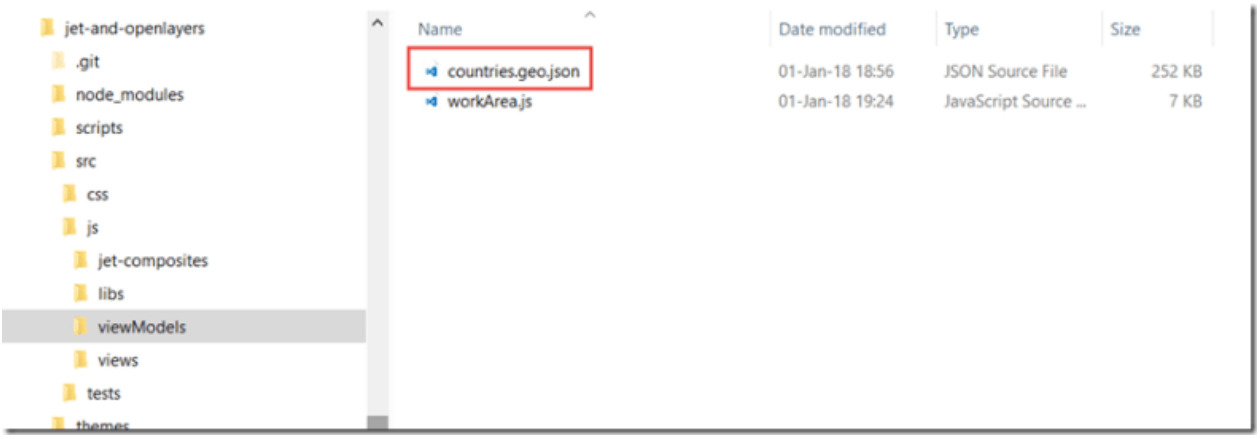
```
        return new MapAreaViewModel();
    }
);
```

Add a DIV with data bind for mapArea module to index.html

```
...</header>
<div role="main" class="oj-web-applayout-max-width oj-web-applayout-content">
<div data-bind="ojModule:'mapArea'" />
</div>
<footer class="oj-web-applayout-footer" role="contentinfo">
...
```

Download a file in GEOJSON format with annotated geo-json geometries for the countries of the world

Download a GEOJSON file with country data from GitHub: https://github.com/johan/world.geo.json and place the file in the directory src\js\viewModels of the JET application:



Initialize the map with two layers – raster OSM world map and vector country shapes

```
function MapAreaViewModel() {
    var self = this;

    self.selectedCountry = ko.observable("France");
    self.countryChangedListener = function(event) {
        // self.selectInteraction.getFeatures().clear();
        // self.setSelectedCountry(self.selectedCountry())
    }


    $(document).ready
        (
        // when the document is fully loaded and the DOM has been initialized
        // then instantiate the map
        function () {
            initMap();
        })
```

```
    function initMap() {
        var style = new ol.style.Style({
            fill: new ol.style.Fill({
                color: 'rgba(255, 255, 255, 0.6)'
            }),
            stroke: new ol.style.Stroke({
                color: '#319FD3',
                width: 1
            }),
            text: new ol.style.Text()
        });

        self.countriesVector = new ol.source.Vector({
            url: 'js/viewModels/countries.geo.json',
            format: new ol.format.GeoJSON()
        });
       self.map2 = new ol.Map({
            layers: [
                new ol.layer.Vector({
                    id: "countries",
                    renderMode: 'image',
                    source: self.countriesVector,
                    style: function (feature) {
                        style.getText().setText(feature.get('name'));
                        return style;
                    }
                })
                , new ol.layer.Tile({
                    id: "world",
                    source: new ol.source.OSM()
                })
            ],
            target: 'map2',
            view: new ol.View({
                center: [0, 0],
                zoom: 2
            })
        });
 }//initMap
```

Add overlay to highlight countries that are hovered over

Note: this code is added to the initMap function:

```
// layer to hold (and highlight) currently selected feature(s)
var featureOverlay = new ol.layer.Vector({
    source: new ol.source.Vector(),
    map: self.map2,
    style: new ol.style.Style({
        stroke: new ol.style.Stroke({
            color: '#f00',
```

```
            width: 1
        }),
        fill: new ol.style.Fill({
            color: 'rgba(255,0,0,0.1)'
        })
    })
});

var highlight;
var displayFeatureInfo = function (pixel) {

    var feature = self.map2.forEachFeatureAtPixel(pixel, function (feature) {
        return feature;
    });

    var info = document.getElementById('info');
    if (feature) {
        info.innerHTML = feature.getId() + ': ' + feature.get('name');
    } else {
        info.innerHTML = ' ';
    }

    if (feature !== highlight) {
        if (highlight) {
            featureOverlay.getSource().removeFeature(highlight);
        }
        if (feature) {
            featureOverlay.getSource().addFeature(feature);
        }
        highlight = feature;
    }

};

self.map2.on('pointermove', function (evt) {
    if (evt.dragging) {
        return;
    }
    var pixel = self.map2.getEventPixel(evt.originalEvent);
    displayFeatureInfo(pixel);
});
```

Add Select Interaction – to allow selection of a country – applying a bold style to the selected country

This code is based on this example: http://openlayers.org/en/latest/examples/select-features.html .

```
// define the style to apply to selected countries
var selectCountryStyle = new ol.style.Style({
    stroke: new ol.style.Stroke({
        color: '#ff0000',
        width: 2
    })
    , fill: new ol.style.Fill({
```

```
        color: 'red'
    })
});
self.selectInteraction = new ol.interaction.Select({
    condition: ol.events.condition.singleClick,
    toggleCondition: ol.events.condition.shiftKeyOnly,
    layers: function (layer) {
        return layer.get('id') == 'countries';
    },
    style: selectCountryStyle

});
// add an event handler to the interaction
self.selectInteraction.on('select', function (e) {
    //to ensure only a single country can be selected at any given time
    // find the most recently selected feature, clear the set of selected
features and add the selected the feature (as the only one)
    var f = self.selectInteraction.getFeatures()
    var selectedFeature = f.getArray()[f.getLength() - 1]
    self.selectInteraction.getFeatures().clear();
    self.selectInteraction.getFeatures().push(selectedFeature);
});
```

and just after the declaration of self.map2:

```
...
    self.map2.getInteractions().extend([self.selectInteraction]);
```

Update JET component from country selection

Add to the end of the select event handler of the selectInteraction:

```
var selectedCountry = { "code": selectedFeature.id_, "name":
selectedFeature.values_.name };
 // set name of selected country on Knock Out Observable
self.selectedCountry(selectedCountry.name);
```

Create

```
self.setSelectedCountry = function (country) {
    //programmatic selection of a feature
    var countryFeatures = self.countriesVector.getFeatures();
    var c = self.countriesVector.getFeatures().filter(function (feature) {
return feature.values_.name == country });
    self.selectInteraction.getFeatures().push(c[0]);
}
```

Set country selection on map based on value [change]in JET component

Implement the self.countryChangedListener that is refered to in the mapArea.html file in the input-text componentL:

```
self.countryChangedListener = function(event) {
    self.selectInteraction.getFeatures().clear();
    self.setSelectedCountry(self.selectedCountry())
}
```

Create the following listener (for the end of loading the GeoJSON data in the countriesVector); when loading is ready, the current country value in the selectedCountry observable backing the input-text component is used to select the initial country:

```
var listenerKey = self.countriesVector.on('change', function (e) {
    if (self.countriesVector.getState() == 'ready') {
        console.log("loading dione");
        // and unregister the "change" listener
        ol.Observable.unByKey(listenerKey);
        self.setSelectedCountry(self.selectedCountry())
    }
});
```

References

GitHub Repo with the code (JET Application) : https://github.com/lucasjellema/jet-and-openlayers .

Countries GeoJSON file – https://github.com/johan/world.geo.json

Open Layers Example of Select Interaction – http://openlayers.org/en/latest/examples/select-features.html

Open Layers API – Vector: http://openlayers.org/en/latest/apidoc/ol.source.Vector.html

Event Listener for OpenLayers Vector with GEOJSON source – https://gis.stackexchange.com/questions/123149/layer-loadstart-loadend-events-in-openlayers-3/123302#123302

Animated Gif maker – http://gifmaker.me/

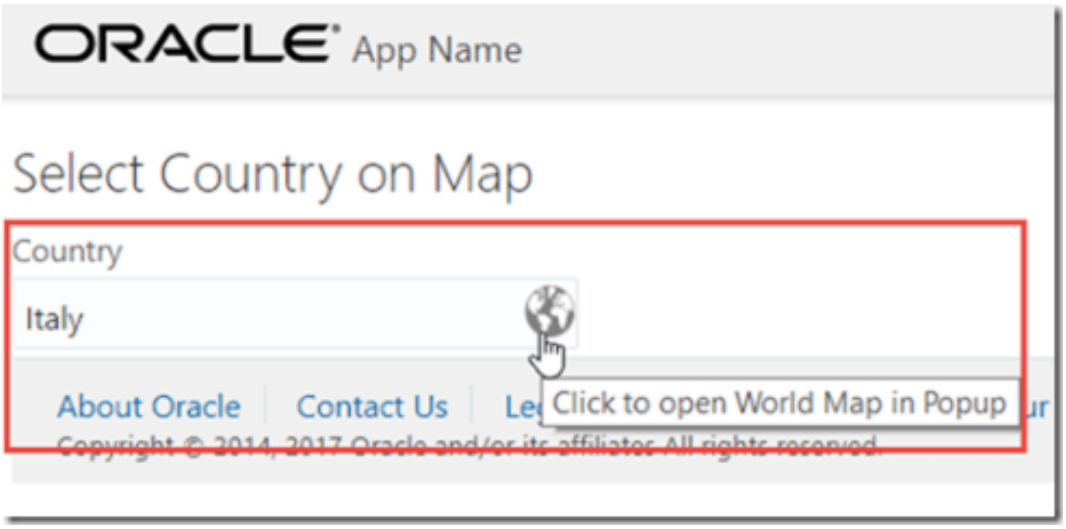
OpenLayers 3 : Beginner's Guideby Thomas Gratier; Erik Hazzard; Paul Spencer, Published by Packt Publishing, 2015

OpenLayers Book – Handling Selection Events – http://openlayersbook.github.io/ch11-creating-web-map-apps/example-08.html

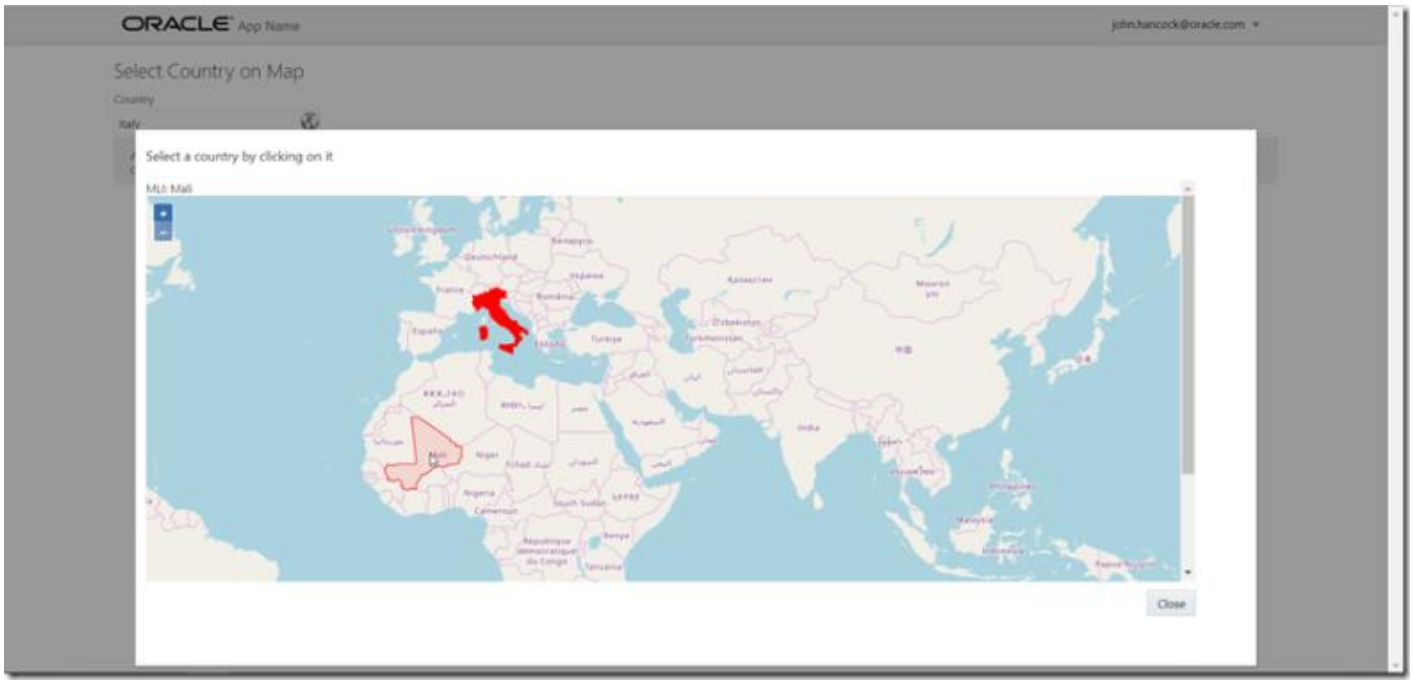
5. Promoting a simple Oracle JET input text component to a Visual Country Selection component

To select a country (or any other geographical entity), we can offer our end users a dropdown list, a list of values in a popup window, an auto suggest input text component or a more visual, map based approach. In this practice, we create the latter: Open a modal popup that presents a map of the world and allows the user to inspect and select a country.

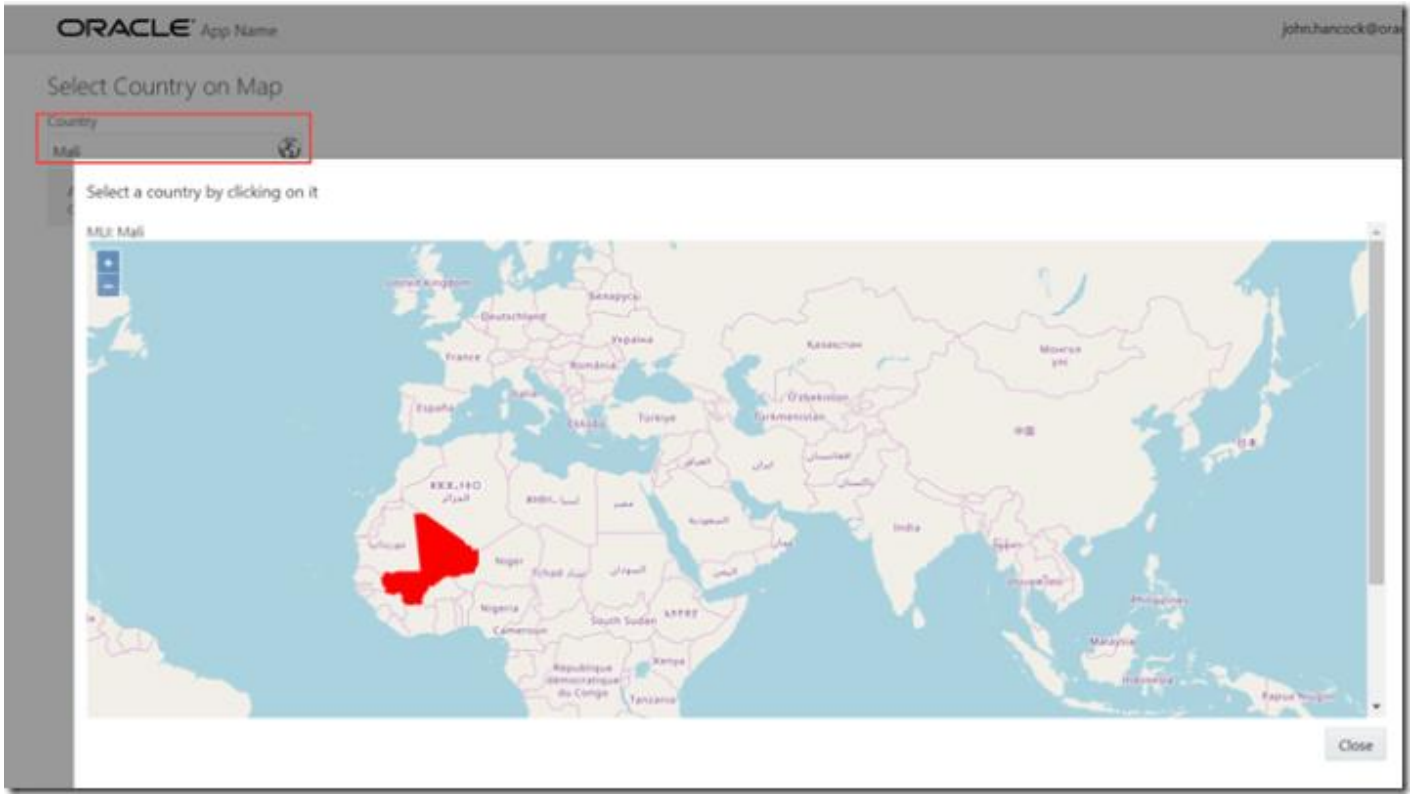
This exercise demonstrates how we go from a simple input text component and extend it: with a background image (a little globe) that the user can click.



When the user clicks the icon, a popup opens and shows the world map. If the input component contains a valid country name, this country is selected in the map. The user can click on any country and in doing so, selects the country.



The name of the country is written back to the input text component (Mali in the next figure).



The main challenges addressed in this practice:

- Embed the OpenLayers map in an Oracle JET popup component
- Embed a clickable icon in an input text component to allow the end user to open the popup with a mouse click

- Link up the map with the input text component

Starting with the situation from the previous lab, this lab describes how to create an input text component with clickable icon that opens a popup window that shows the map.

The final code is found here on GitHub: https://github.com/lucasjellema/jet-and-openlayers.

Embed the OpenLayers map in an Oracle JET Popup Component

This step is a simple one. I used the entry in the JET Cookbook on Popup Component: http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html?component=popup&demo=modal. I copied the styles used in this example to the file src/css/app.css:

```
.demo-popup {
    width: 80vw;
    height: 80vh;
    display: none;
  }
  .demo-popup-body {
    width: 75vw;
    height: 75vh;
    display: flex;
    flex-direction: column;
    align-items: stretch;
  }
  .demo-popup-header {
    align-self: flex-start;
    margin-bottom: 10px;
  }
  .demo-popup-content {
    align-self: stretch;
    overflow: auto;
    flex-basis: 60vh;
  }
  .demo-popup-footer {
    align-self: flex-end;
    margin-top: 10px;
  }
```

Next I added the popup component in the mapArea.html file, with the DIV that acts as the map container in

```
<div id="popupWrapper">
    <oj-popup class="demo-popup" id="countrySelectionPopup" tail="none"
position.my.horizontal="center" position.my.vertical="bottom"
position.at.horizontal="center"
        position.at.vertical="bottom" position.of="window"
position.offset.y="-10" modality="modal" data-bind="event:{'ojAnimateStart':
startAnimationListener}">
        <div class="demo-popup-body">
            <div class="demo-popup-header">
                <h5>Select a country by clicking on it</h5>
            </div>
```

```
            <div class="demo-popup-content">
                <div id="countryInfo"></div>
                <div id="mapContainer" class="map"></div>
            </div>
            <div class="demo-popup-footer">
                <oj-button id="btnClose" data-bind="click: function()
                            {
                              var popup =
document.querySelector('#countrySelectionPopup');
                              popup.close();
                            }">
                    Close
                </oj-button>
            </div>
        </div>
    </oj-popup>
</div>
```

I added the function startAnimationListener to the ViewModel in mapArea.js:

```
self.startAnimationListener = function (data, event) {
    var ui = event.detail;
    if (!$(event.target).is("#countrySelectionPopup"))
        return;

    if ("open" === ui.action) {
        event.preventDefault();
        var options = { "direction": "top" };
        oj.AnimationUtils.slideIn(ui.element, options).then(ui.endCallback);
        // if the map has not yet been initialized, then do the initialization
now (this is the case the first time the popup opens)
        if (!self.map) initMap();
    }
    else if ("close" === ui.action) {
        event.preventDefault();
        ui.endCallback();
    }
}
```

The initialization of the map is taken care of in open stage for the popup this startAnimationListener – on the first occasion this popup is opened.

Embed a clickable icon in an input text component to allow the end user to open the popup with a mouse click

It is important that the user has a convenient way of opening the country selection popup. Just like the date and time input components provide a little clickable icon, inline in the input component, I want the country selector to consist of a regular JET input text component with an inline icon that the user can click on to bring up the popup.

The HTML code to make this happen is added to mapArea.html:

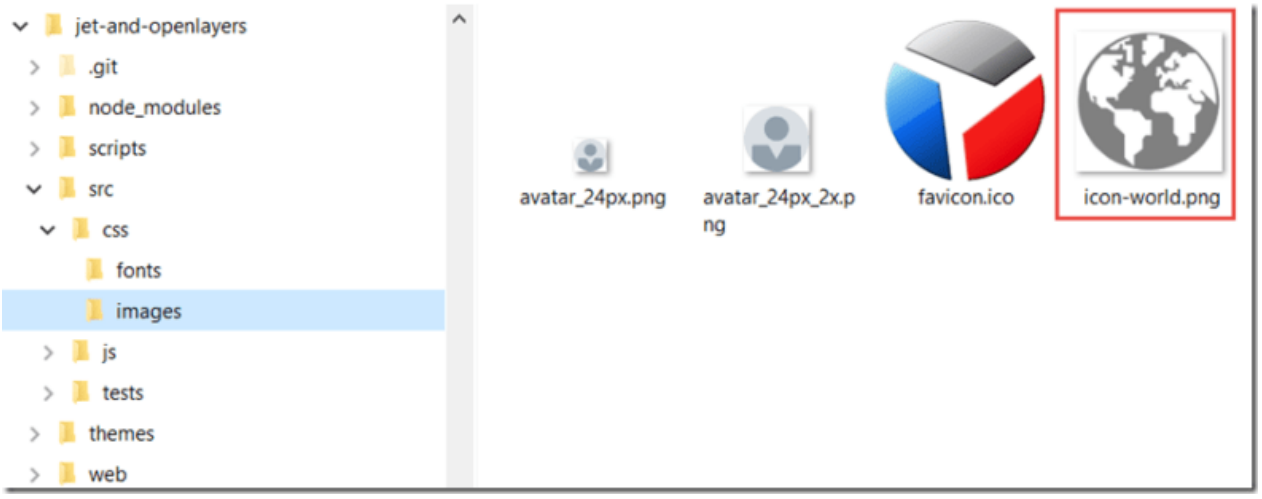
```
<div id="countryInput">
    <oj-label for="country-input">Country</oj-label>
    <oj-input-text id="country-input" value="{{selectedCountry}}" on-value-
changed="[[countryChangedListener]]"
    ></oj-input-text>
    <img id="countrySelectorOpen" src="css/images/icon-world.png"
class='iconbtnintext' title ="Click to open World Map in Popup" data-
bind="click: openPopup" />
</div>
```

Note how the click event on the image is bound to the KnockOut ViewModel's openPopup function using the data-bind notation,

The CSS style *iconbtnintext* takes care of the positioning of the icon. This style is added to app.css:

```
.iconbtnintext {
    position:absolute;
    cursor:pointer;
    width:22px;
    height:23px;
    margin-left:-24px;
    padding-top:4px;
}
```

Finally the icon to be used – icon-world.png – is downloaded from https://github.com/lucasjellema/input-country-composite-component/blob/master/src/js/jet-composites/input-country/images/icon-world.png and copied to src/css/images.



References

JET Cookbook on Popup
Component: http://www.oracle.com/webfolder/technetwork/jet/jetCookbook.html?component=popup&demo=modal

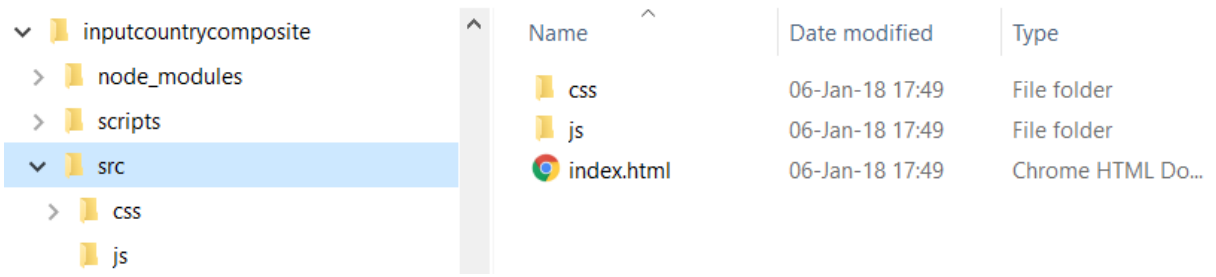
6. Baby steps with a Composite Component

Let's first generate a new JET application in which we will create and test our composite component. Run the following command with the JET CLI on the command line

ojet create inputcountrycomposite --template=basic

```
c:\data>ojet create inputcountrycomposite --template=basic
 Oracle JET CLI
Processing template... basic
Oracle JET: Your app structure is generated. Continuing with library install...
Performing npm install may take a bit...
Invoking npm install
npm WARN deprecated qunitjs@2.4.1: 2.4.1 is the last version where QUnit will be published as 'qunitjs'. To receive futu

added 556 packages and removed 1 package in 47.059s
Writing  oraclejetconfig.json
Oracle JET: oraclejetconfig.json file exists...checking config...
Oracle JET: Your app is ready! Change to your new app directory inputcountrycomposite and try ojet build and serve...
```

At this point, the project scaffold has been created.

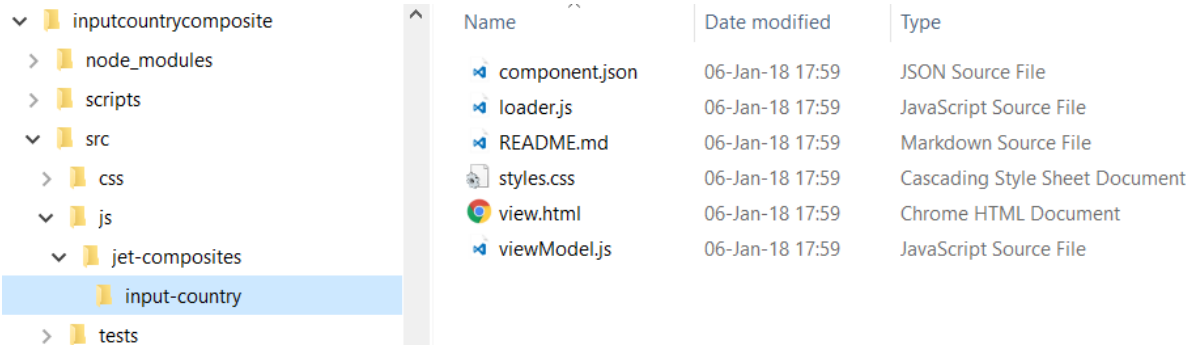


Change to the generated directory *inputcountrycomposite*. Instantiate a new composite component through the CLI:

ojet create component input-country

```
c:\data>cd inputcountrycomposite

c:\data\inputcountrycomposite>ojet create component input-country
 Oracle JET CLI
Oracle JET: add component input-country finished.
```

This will generate the artefacts that make up [the absolute minimum implementation of] the composite component, in a directory called after the composite component that is located under the src/js/jet-composites directory:



Open file component.json that describes the component – its functionality and all of its properties and events. Replace the contents of this file with:

```
{

  "name": "input-country",

  "displayName": "input-country",

  "description": "This component allows users to input the name of a country - either by typing it or by selecting it on a map presented in a popup.",

  "version": "1.0.0",

  "jetVersion": "^4.1.0",

  "properties": {

    "countryName": {

      "description": "Property to hold the name of the selected country; a string can be passed in, or a data bound expression",

      "type": "string",

      "writeback": true

    }

  },

  "methods": {},

  "events" : {},

  "slots": {}

}
```

Here we declare a single property called *countryName* of type string. In later stages we will extend the component's definition with more properties, a callback function and an event.

Review the loader.js file. This file takes care of registering the composite component, thus making it available in the consuming JET application.

Open file view.html, the file that defines the markup for the component. This can be a combination of standard HTML5 components, out-of-the-box JET components and custom composite components. Update the file to:

```
<h3 data-bind="text: $props.countryName"/>
```

This specifies the component will render an *h3* element with its content derived from the *countryName* property that has been passed in.

In order to consume this bare component, we will add a module called *workarea* to the JET application. Create directories *views* and *viewModels* under *src/js*. Create file *workarea.html* in the *views* directory. Open the file and copy and paste this markup code:

```
<h2>Workarea with Composite Component input-country</h2>

<div>

    <input-country country-name="{{country}}" country-selection-
handler={{handleCountrySelection}}

     on-country-selected={{countrySelectedHandler}}  label="Enter your country"/>

</div>

<h4 data-bind="text: upperCountry"></h4>

<div>

    <input-country country-name="{{country2}}" country-selection-
handler={{handleCountry2Selection}} label="Destination of Next Holiday Trip"

     on-country-name-changed={{handleCountryNameChangedHandler}}  />

</div>

<h4 data-bind="text: country2"></h4>

<br/>
```

Here we see two instances of the input-country component. The attribute *country-name* corresponds to the property *countryName* and is data bound to the viewModel. The other attributes – *label*, *country-selection-handler* and *on-country-selected* – are to be discussed at a later stage and can be ignored for now.

Next create file *workarea.js* in the *viewModels* directory and set its content to:

```
requirejs.config(

    {
```

```
      // create path mapping for input-country module
      paths:
      {
        'input-country':'jet-composites/input-country'
      }
    });
define(
    ['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojinputtext'
    , 'input-country/loader'
],
    function (oj, ko, $) {
        'use strict';
        function WorkareaViewModel() {
            var self = this;
            // initialize two country observables
            self.country = ko.observable("Italy");
            self.country2 = ko.observable("Indonesia");
            // a computed observable, based on the first country observable
            // whenever self.country is updated, this observable is also modified
            self.upperCountry = ko.computed(function() {
                return this.country().toUpperCase();
            }, self);


            // function to be called back from input-country component
            self.handleCountrySelection= function (selectedCountryName,
selectedCountryCode) {
                console.log('Callback Function to Handle Country Selection name '+
selectedCountryName+ ' and code '+ selectedCountryCode);
            }
```

```
            // function to be called back from input-country component

            self.handleCountry2Selection= function (selectedCountryName,
selectedCountryCode) {

                console.log('Callback Function to Handle Country Selection name '+
selectedCountryName+ ' and code '+ selectedCountryCode);

            }



            // function to handle the countrySelected event that can be published by
the input-country component

            self.countrySelectedHandler = function(countrySelectedEvent) {

                console.log("countrySelectedHandler - to handle countrySelected event
"+JSON.stringify(countrySelectedEvent.detail))

            }



            self.handleCountryNameChangedHandler = function(countryNameChangedEvent) {

                console.log("handleCountryNameChangedHandler - to handle out-of-the-
box countryNameChanged event "+JSON.stringify(countryNameChangedEvent.detail))

            }







            }



        return new WorkareaViewModel();

    }

);
```

A path mapping is created for the input-country module – relative to the directory that holds he workarea.js file. We will see URLs later on, referring to resources like an image and a JSON document inside the input-country module. The relative references used in these URLs extend from this path mapping for the module.

The new composite needs to be activated in order to be used in the *workarea* module. To that end, the reference 'jet-composites/input-country/loader' is added in the *define* call.

Two observables are created in the ViewModel and initialized with values (Italy and Indonesia). These observables are passed as property to the input-country component. We will make use of the functions you see in this code snippet in later stages in this article.

Open file index.html. Between the <header> and <footer> elements, add this reference to the *workarea* module:

```
<div role="main" class="oj-web-applayout-max-width oj-web-applayout-content">
   <h3>index.html: Module workarea</h3>
   <div data-bind="ojModule:'workarea'"/>
 </div>
```

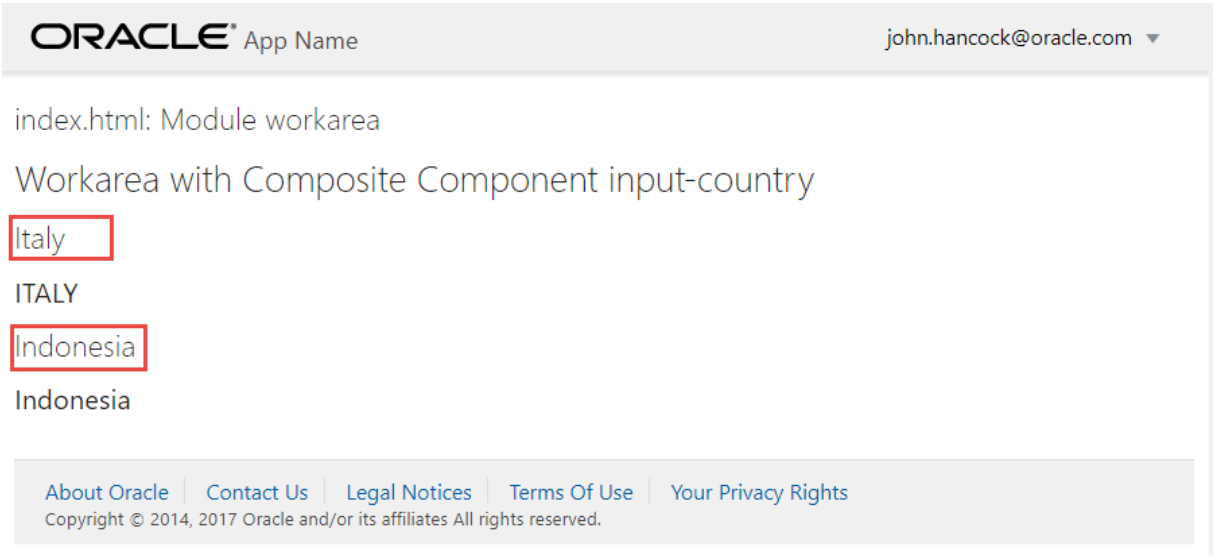
Finally, in order to enable the JET application to work with modules, you need to update the file main.js and add the reference to *ojs/ojModule* in the *require* call:

```
require(['ojs/ojcore', 'knockout', 'appController', 'ojs/ojknockout', 'ojs/ojbutton',
'ojs/ojtoolbar', 'ojs/ojmenu','ojs/ojmodule'],
```

You are now ready to run the application and see the input-country component in action. Navigate to the root directory of the project on the command line and run:

ojet serve

This command should result in the application being compiled, built and run. Your web browser is launched at url localhost:8000 and should display the application as follows:



Note: the red rectangles were added to indicate the output from the two input-country instances.

7. Enriching the Input Country Composite Component

In this stage, we will enrich the component a little. You need to copy the image file *icon-world.png* to a new directory *images* under *jet-composites/input-country*.

Open the file styles.css that contains the style definitions for the composite component, and add:

```
input-country .iconbtnintext {

  position:absolute;

  cursor:pointer;

  width:22px;

  height:23px;

  margin-left:-24px;

  padding-top:4px;

}
```

This style is applied to the image – to shrink it to icon-size and position it appropriately. Subsequently, change the content of view.html to:

```
<div :id='[['country-input-container'+ $uniqueId]]'>

    <oj-label data-bind="attr: {for: 'country-input'+ $uniqueId }" ><span data-
bind="text: $props.label"></span></span></oj-label>

    <oj-input-text :id="[['country-input'+ $uniqueId]]"
value="{{$props.countryName}}"></oj-input-text>

    <img  :id="[['countrySelectorOpen'+ $uniqueId]]" :src='[[require.toUrl("input-
country/images/icon-world.png")]]' class='iconbtnintext' title="Click to open World
Map in Popup"

       />

</div>
```

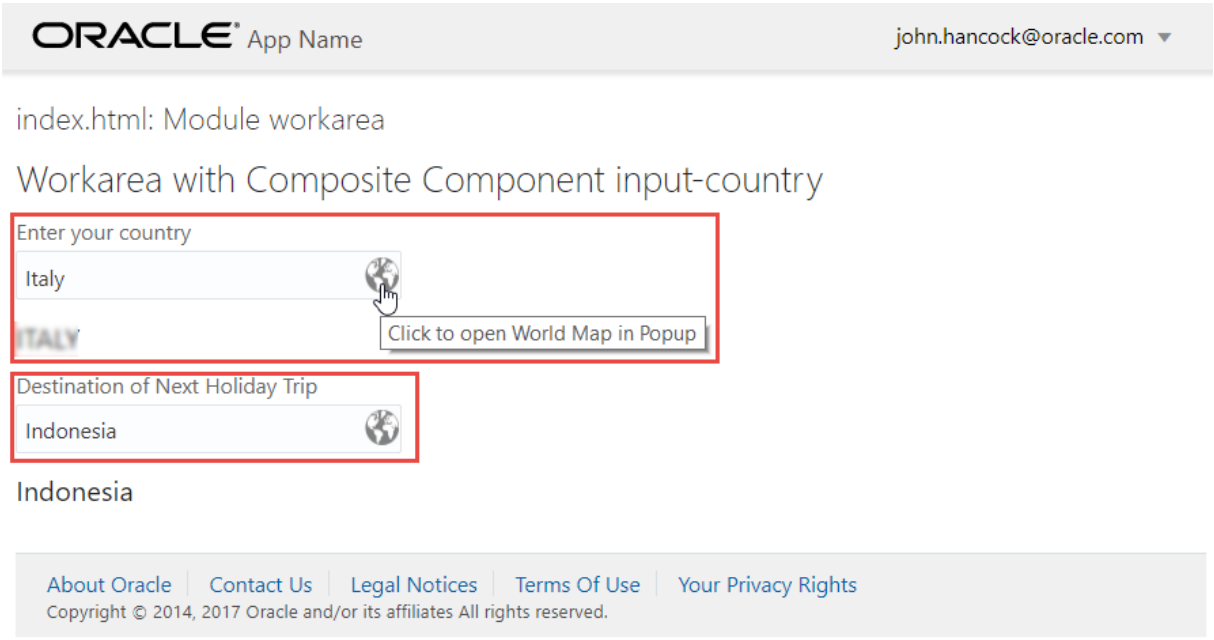
Quite a bit is going on here. The component now renders an input-text component with a label and an *img* element based on the image file that you just added, using a local reference that is turned to the appropriate URL through the call to *require.toUrl*. It also references a new property called *label* for which the definition should be added to *component.json*:

```
    "label": {

      "description": "This property is used to specify the text to be shown as the
label (aka prompt) for the input field",

      "type": "string",
```

```
    "value": "Country"
  }
```

Note: we use $uniqueId in view.html to ensure that the HTML elements rendered by our component have unique identifiers, even if multiple instances of the component are included in the same page. See this article by Duncan Mills for some background: https://blogs.oracle.com/groundside/jet-composite-components-xvii-beware-the-ids.

Some of these changes are not picked up automatically, so you will have to abort the current process and run *ojet serve* again to load the somewhat enriched component:



The component now renders a standard JET input-text component with an associated label that takes its value from the *label* property passed in from *workarea.html*. The image is styled as small icon, shown right aligned inside the input-text field. It invites the user to click – but if you do, nothing happens yet. That is for the next stage.

8. Add the Popup to the Input Country Composite Component

To make a popup appear when the icon is clicked on – as is our aim – we have to add the popup to view.html like so:

```
<div :id="[['country-input-container'+ $uniqueId]]">

  <oj-label data-bind="attr: {for: 'country-input'+ $uniqueId }">

    <span data-bind="text: $props.label"></span>

    </span>

  </oj-label>

  <oj-input-text :id="[['country-input'+ $uniqueId]]"
value="{{$props.countryName}}"></oj-input-text>

  <img :id="[['countrySelectorOpen'+ $uniqueId]]" :src='[[require.toUrl("input-
country/images/icon-world.png")]]' class='iconbtnintext'

    title="Click to open World Map in Popup" data-bind="click: openPopup" />

</div>

<div id="popupWrapper">

  <oj-popup class="input-country-country-selection-popup"
:id="[['countrySelectionPopup'+ $uniqueId ]]" tail="none"
position.my.horizontal="center"

    position.my.vertical="bottom" position.at.horizontal="center"
position.at.vertical="bottom" position.of="window" position.offset.y="-10"

    modality="modal" data-bind="event:{'ojAnimateStart': startAnimationListener}">

    <div class="input-country-country-selection-popup-body">

      <div class="input-country-country-selection-popup-header">

        <h5>Select a country by clicking on it</h5>

      </div>

      <div class="input-country-country-selection-popup-content">

        <div :id="[['countryInfo'+ $uniqueId ]]"></div>

        <div :id="[['mapContainer'+ $uniqueId ]]"></div>

      </div>

      <div class="input-country-country-selection-popup-footer">

        <oj-button :id="[['btnClose'+ $uniqueId ]]" data-bind="click: function()

  {
```

```
    var popup = document.querySelector('#countrySelectionPopup'+$uniqueId);
    popup.close();
  }">
          Cancel
        </oj-button>
      </div>
    </div>
  </oj-popup>
</div>
```

This adds the JET Popup component to be opened as a modal window – with a title, a *cancel* button and no other content [yet]. You will probably notice the data bound property *click* on the image, that refers to a function called *openPopup*. This function is to be added to *viewModel.js* along with function *startAnimationListener*:

```
define(
    ['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojbutton', 'ojs/ojpopup'], function (oj, ko, $) {
        'use strict';

        function InputCountryComponentModel(context) {
            var self = this;
            // save a reference to the unique identity of the composite component instance - also used in generating the element id values in view.html
            // see https://blogs.oracle.com/groundside/jet-composite-components-xvii-beware-the-ids for reference
            self.unique = context.unique;
            self.composite = context.element;

            self.openPopup = function () {
                $('#countrySelectionPopup' + self.unique).ojPopup("open");
            }//openPopup
```

```
            self.startAnimationListener = function (data, event) {

                var ui = event.detail;

                if (!$(event.target).is("#countrySelectionPopup" + self.unique))

                    return;


                if ("open" === ui.action) {

                    event.preventDefault();

                    var options = { "direction": "top" };

                    oj.AnimationUtils.slideIn(ui.element,
options).then(ui.endCallback);

                    // if the map has not yet been initialized, then do the
initialization now (this is the case the first time the popup opens)

                    if (!self.map) initMap();



                }

                else if ("close" === ui.action) {

                    event.preventDefault();

                    ui.endCallback();

                }

            }



        };


        //Lifecycle methods - uncomment and implement if necessary

        //ExampleComponentModel.prototype.activated = function(context){

        //};


        //ExampleComponentModel.prototype.attached = function(context){

        //};
```

```
        //ExampleComponentModel.prototype.bindingsApplied = function(context){
        //};

        //ExampleComponentModel.prototype.detached = function(context){
        //};

        return InputCountryComponentModel;
    });
```

Note how the *ojButton* and *ojPopup* modules are included in the define() for this viewModel.

Styles associated with the *class* attributes in view.html are added to styles.css. These take care of assigning proper width and height to the popup:

```
.input-country-country-selection-popup {
  width: 80vw;
  height: 80vh;
  display: none;
}
 .input-country-country-selection-popup-body {
  width: 75vw;
  height: 75vh;
  display: flex;
  flex-direction: column;
  align-items: stretch;
}
 .input-country-country-selection-popup-header {
  align-self: flex-start;
  margin-bottom: 10px;
}
 .input-country-country-selection-popup-content {
  align-self: stretch;
```

```
  overflow: auto;
  flex-basis: 60vh;
}
 .input-country-country-selection-popup-footer {
  align-self: flex-end;
  margin-top: 10px;
}
```

Note how the style names for the popup have been associated with an imaginary input-country namespace by prefixing the style names with the name of our component. This is done to reduce the chances of collisions with styles defined in other components or in the consuming applications.

You now need to restart *ojet serve* in order to pick up the changes in styles.css. Clicking the icon at this point results in a popup – with little content:

9. Add OpenLayers to the Component and a world map to the Popup

In this stage we will start the integration of OpenLayer in our component. Extract the contents from the OpenLayers distribution zip file – ol-debug.js, ol.js and ol.css - to a new directory: libs\openlayers under the composite component's root: src\js\jet-composites\input-country.

Modify the contents of loader.js to load the OpenLayers JavaScript library (as module inputCountry/ol) and CSS source, like this:

```
/**

  Copyright (c) 2015, 2017, Oracle and/or its affiliates.

  The Universal Permissive License (UPL), Version 1.0

*/

define(['ojs/ojcore', 'text!./view.html', './viewModel', 'text!./component.json',
'css!./styles', 'css!./libs/openlayers/ol', 'ojs/ojcomposite'],

  function(oj, view, viewModel, metadata) {

    oj.Composite.register('input-country', {

      view: {inline: view},

      viewModel: {inline: viewModel},

      metadata: {inline: JSON.parse(metadata)}

    });

  }

);
```

The other changes are all in viewModel.js. First of all, the inputCountry/ol module needs to be added to the *define* call:

```
define(

    ['ojs/ojcore', 'knockout', 'jquery',  './libs/openlayers/ol-debug',
'ojs/ojbutton', 'ojs/ojpopup'], function (oj, ko, $, ol) {

        'use strict';
```

Then, the OpenLayers world map has to be initialized when the popup opens. Therefore, this line should be added to the *openPopup* function:

```
// if the map has not yet been initialized, then do the initialization now (this is
the case the first time the popup opens)

if (!self.map) initMap();
```

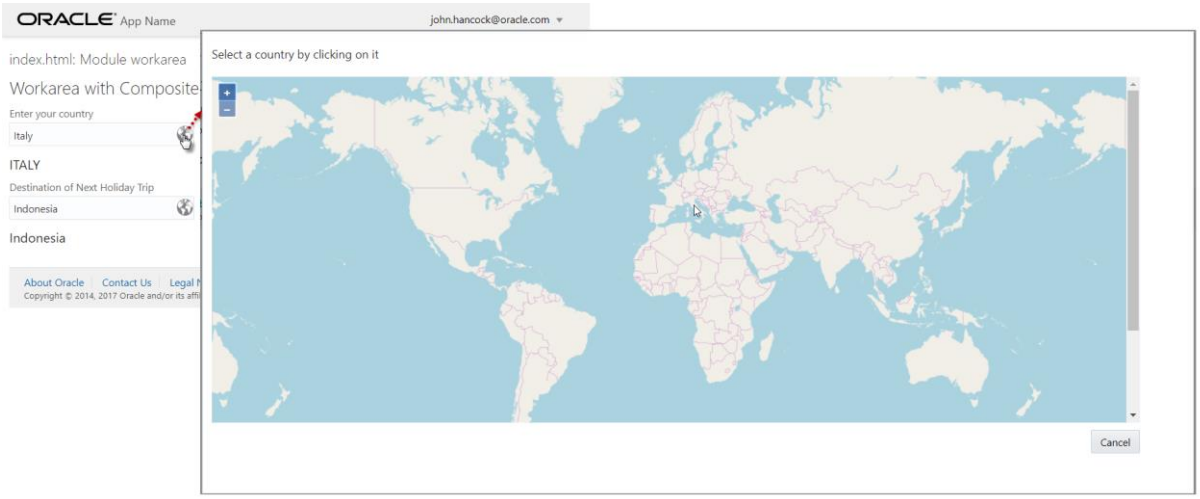
Finally the function that does the actual map initialization needs to be added to the InputCountryComponentModel:

```
function initMap() {
    self.map = new ol.Map({
        layers: [
            new ol.layer.Tile({
                id: "world",
                source: new ol.source.OSM()
            })
        ],
        target: 'mapContainer'+self.unique,
        view: new ol.View({
            center: [0, 0],
            zoom: 2
        })
    });
}//initMap
```

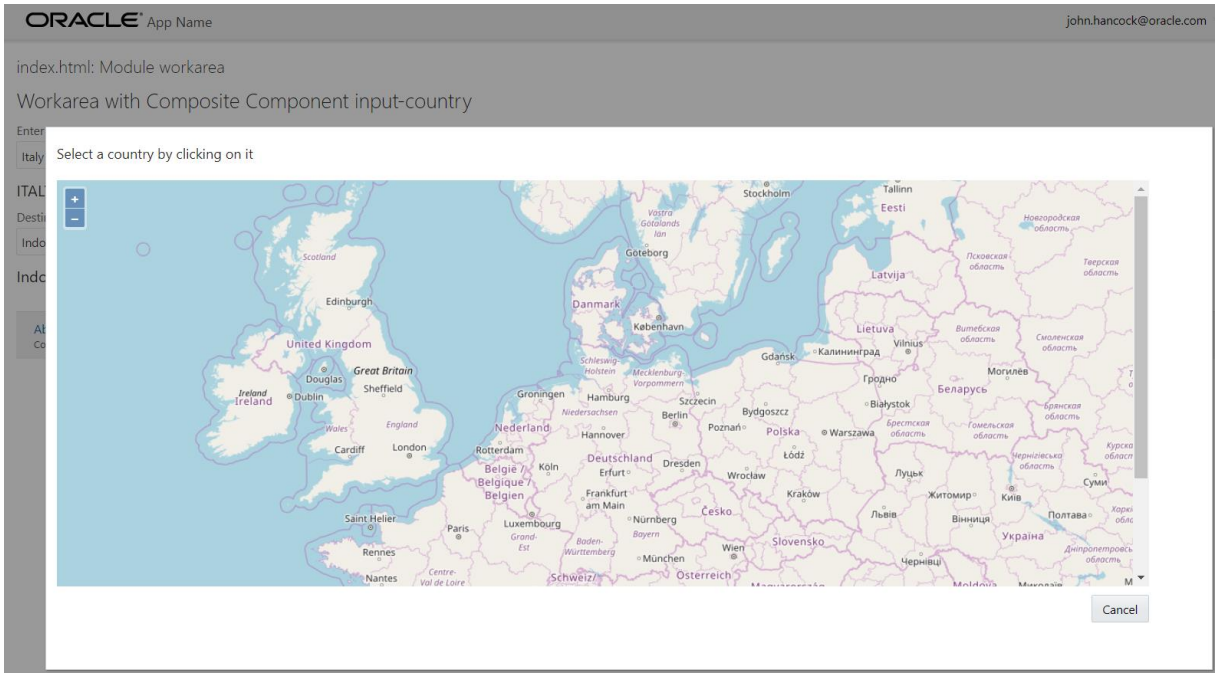
Here you see the first little bit of OpenLayers: a *map* is instantiated with a *layer* based on the OpenStreetMap data source and with a *view* centered at *coordinates* [0,0] with zoom level two (almost completely zoomed out). Feel free to experiment with these coordinates and the zoom level.

Note the use of *self.unique* for setting the element id of the map container element to the *target* property on the map object. The value of this property was set during the initialization of *InputCountryComponentModel,* based on the unique property in the *context* parameter. Its value corresponds to what the expression *$uniqueId* resolves to in the view *view.html*.

Restart *ojet serve*. When the application is running again, a popup with a world map in it should open when you click the image in the input field:



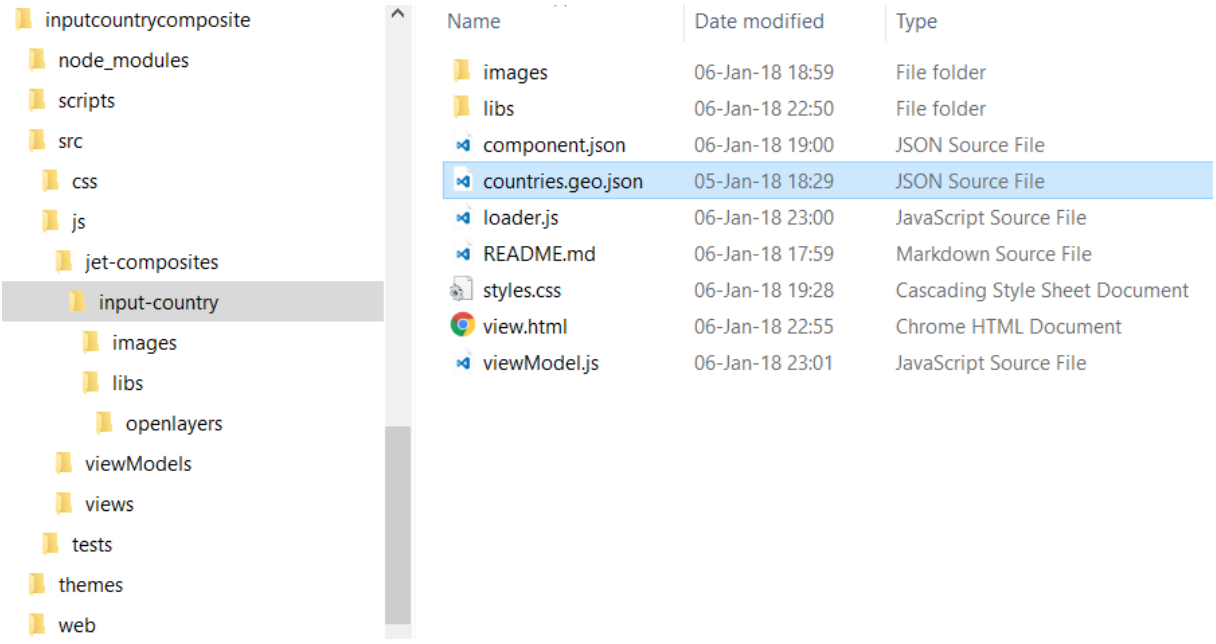
Out of the box, maps in OpenLayers can be panned – click plus drag – and be zoomed in (with double click) and zoomed out (using shift+double click), or using corresponding gestures on touch devices or the controls in the upper left hand corner of the map.



10. Leverage OpenLayers features for a Richer Map Experience in the Composite Component

If we want to select countries on the map, we should demarcate them and provide a visual hint as to which country the user is hovering over. Additionally when a country is clicked, it should be highlighted as to make clear that it was selected. This can easily be achieved using OpenLayers. First we all add a second layer to the map – one that draws the country outlines based on a GeoJSON file with vector coordinates.

Copy file *countries.geo.json* to the root directory of the input-country component, as shown below.



Replace function initMap() in viewModel.js with the following code:

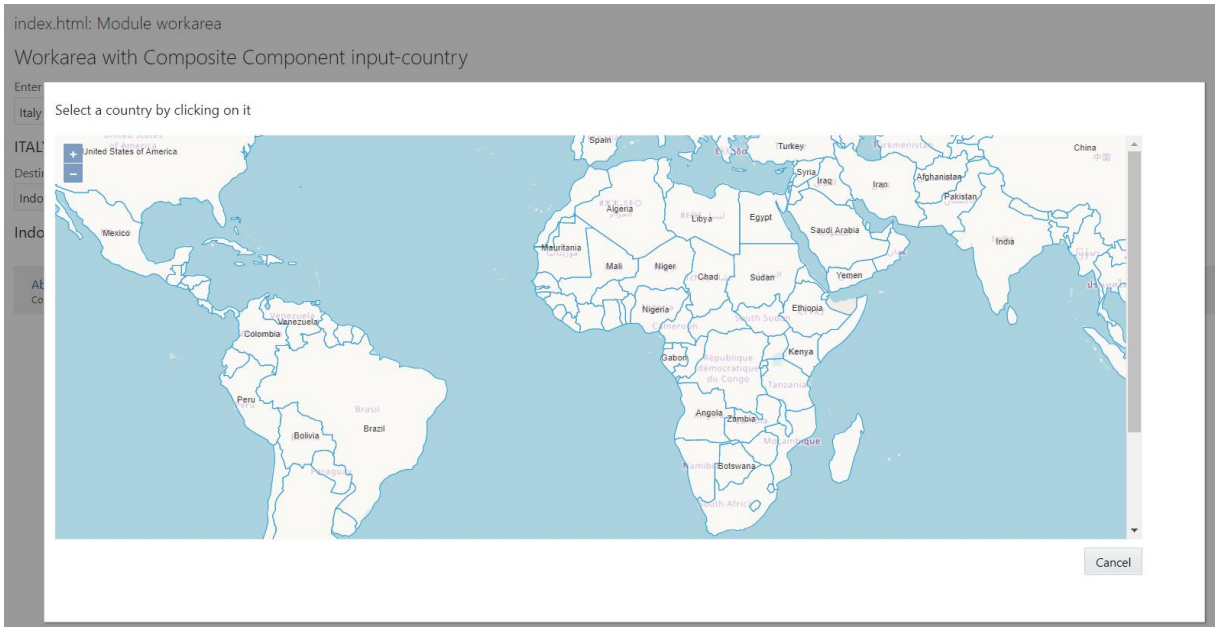
```
function initMap() {
    var style = new ol.style.Style({
        fill: new ol.style.Fill({
            color: 'rgba(255, 255, 255, 0.6)'
        }),
        stroke: new ol.style.Stroke({
            color: '#319FD3',
            width: 1
        }),
```

```
    text: new ol.style.Text()
}); //style

self.countriesVector = new ol.source.Vector({
    url: require.toUrl('input-country/countries.geo.json'),
    format: new ol.format.GeoJSON()
});

self.map = new ol.Map({
    layers: [new ol.layer.Tile({
        id: "world",
        source: new ol.source.OSM()
    }),
    new ol.layer.Vector({
        id: "countries",
        renderMode: 'image',
        source: self.countriesVector,
        style: function (feature) {
            style.getText().setText(feature.get('name'));
            return style;
        }
    })

    ],
    target: 'mapContainer'+self.unique,
    view: new ol.View({
        center: [0, 0],
        zoom: 2
    })
```

```
        });

    }//initMap
```

Vector *countriesVector* is created based on the country features in the countries.geo.json file. Based on the vector, a second *layer* is added to the *map*, to render the outlines of the countries using the *style* that defines the *color* and *width* of the borders as well as the country specific text.

Restart *ojet serve* and inspect the map. Play with the *style* to see the effects on the map.



We will next add a visual effect to the map to indicate the country currently hovered over. Add this code to function initMap():

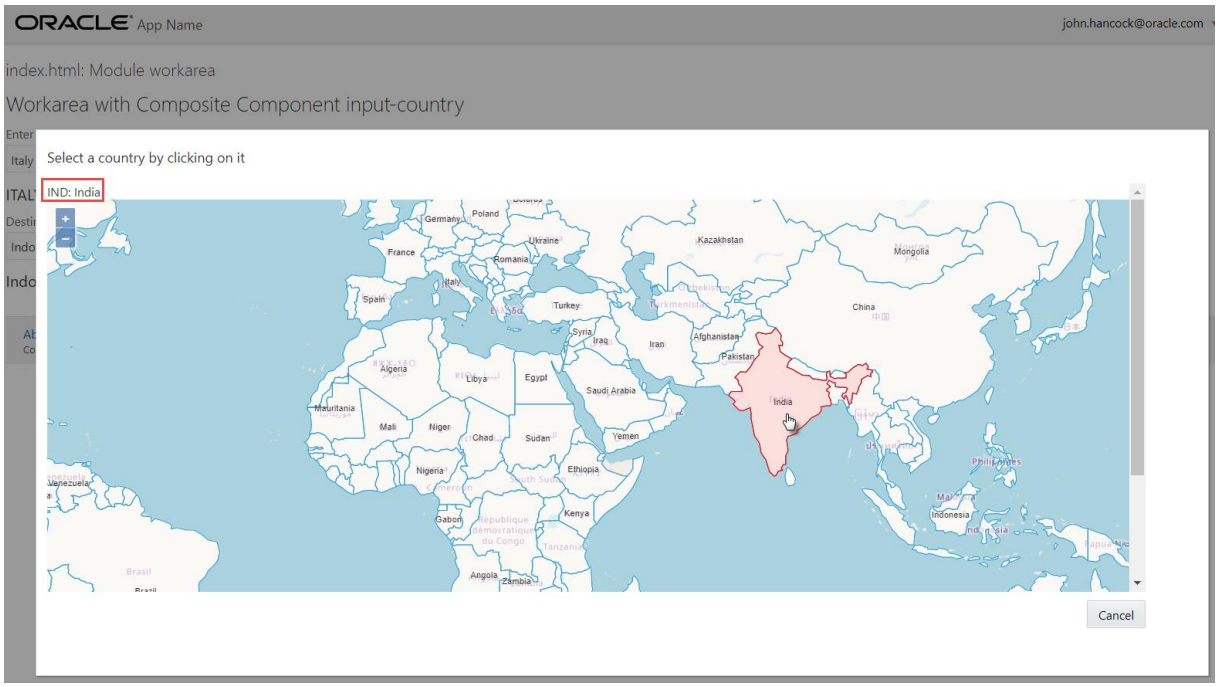
```
        // layer to hold (and highlight) currently hovered over highlighted
(not yet selected) feature(s)

        var featureOverlay = new ol.layer.Vector({

            source: new ol.source.Vector(),

            map: self.map,

            style: new ol.style.Style({

                stroke: new ol.style.Stroke({

                    color: '#f00',

                    width: 1

                }),

                fill: new ol.style.Fill({
```

```
                        color: 'rgba(255,0,0,0.1)'
                    })
                })
            });

            var highlight;

            // function to get hold of the feature under the current mouse
position;
            // the country associated with that feature is displayed in the info
box
            // the feature itself is highlighted (added to the featureOverlay
defined just ovehead)
            var displayFeatureInfo = function (pixel) {
                var feature = self.map.forEachFeatureAtPixel(pixel, function
(feature) {
                    return feature;
                });

                var info = document.getElementById('countryInfo'+self.unique);
                if (feature) {
                    info.innerHTML = feature.getId() + ': ' + feature.get('name');
                } else {
                    info.innerHTML = ' ';
                }

                if (feature !== highlight) {
                    if (highlight) {
                        featureOverlay.getSource().removeFeature(highlight);
                    }
                    if (feature) {
```

```
            featureOverlay.getSource().addFeature(feature);
        }
        highlight = feature;
    }

};

self.map.on('pointermove', function (evt) {
    if (evt.dragging) {
        return;
    }
    var pixel = self.map.getEventPixel(evt.originalEvent);
    displayFeatureInfo(pixel);
});
```

The *featureOverlay* vector is associated with the map. It defines a bold style – red fill and bold red outline – for the features it contains. The *pointermove* event listener on the map is triggered when the user moves the mouse around. The function *displayFeatureInfo* is invoked to find the feature (i.e. country) at the current mouse position, display its name in the info DIV element in the upper left hand corner and add that country to the *featureOverlay* vector.

The screenshot demonstrates the visual effect in the map:



Now we also want to select a country, not just hover over it. For this we will add an OpenLayers *Interaction* element to the map. This code too is added to (the end of) function initMap():

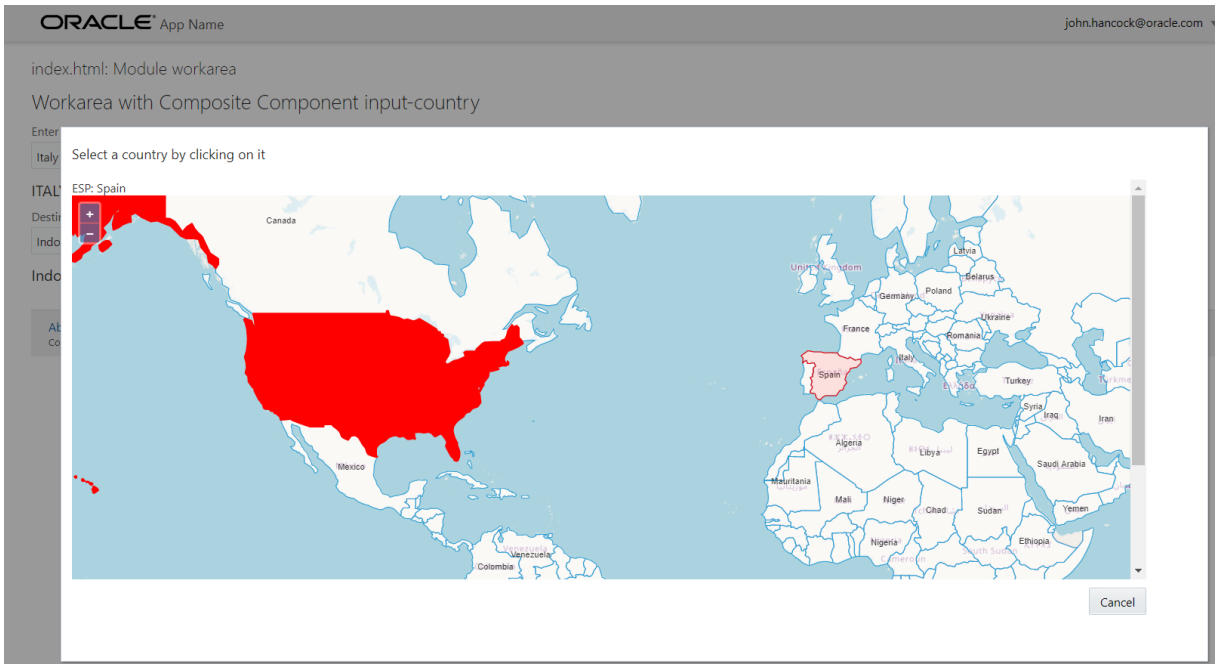
```
// define the style to apply to selected countries
var selectCountryStyle = new ol.style.Style({
    stroke: new ol.style.Stroke({
        color: '#ff0000',
        width: 2
    })
    , fill: new ol.style.Fill({
        color: 'red'
    })
});
self.selectInteraction = new ol.interaction.Select({
    condition: ol.events.condition.singleClick,
    toggleCondition: ol.events.condition.shiftKeyOnly,
    layers: function (layer) {
        return layer.get('id') == 'countries';
    },
```

```
                    style: selectCountryStyle


                });


                self.map.getInteractions().extend([self.selectInteraction]);


                // add an event handler to the interaction
                self.selectInteraction.on('select', function (e) {
                    //to ensure only a single country can be selected at any given
time
                    // find the most recently selected feature, clear the set of
selected features and add the selected the feature (as the only one)
                    var f = self.selectInteraction.getFeatures()
                    var selectedFeature = f.getArray()[f.getLength() - 1]
                    self.selectInteraction.getFeatures().clear();
                    self.selectInteraction.getFeatures().push(selectedFeature);
                    self.countrySelection = { "code": selectedFeature.id_, "name":
selectedFeature.values_.name };
                });
```

The interaction is triggered by the select event – a single click on a feature in the *countries* layer – and undone with shift+click. The *selectCountryStyle* is applied to the selected [country] feature. Feel free to play with this style – that currently fills and outlines the feature with bright red.

The *select* event handler on this interaction element ensures that only a single country can be selected at any one time (as that is the intended behavior of the component although multi-selection can be interesting too). Details for the selected country are stored in the variable *self.countrySelection*. We will use this variable when the selection is passed back to the consuming page.

This change is immediately reloaded to the browser, and now we can select a country by clicking on it:


ORACLE App Name
john.hancock@oracle.com
index.html: Module workarea
Workarea with Composite Component input-country
Select a country by clicking on it
ESP: Spain
Canada
Mexico
Colombia
Venezuela
Latvia
Belarus
Poland
Germany
Ukraine
France
Romania
Italy
Spain
Turkey
Syria
Iraq
Iran
Algeria
Libya
Egypt
Saudi Arabia
Mauritania
Mali
Niger
Chad
Sudan
Yemen
Nigeria
Ethiopia
Cancel

11. Data Binding the Input-Country Component

The purpose of our component is to allow users to select a country and also to return that selection to the underlying viewModel. To make that happen, we should take the selected country on the map and set its name on the *countryName* property.

First, add a Save button in view.html, to allow users to save their selection.

```
<oj-button :id="[['btnSave'+ $uniqueId ]]" data-bind="click: save">
   Save
</oj-button>
```

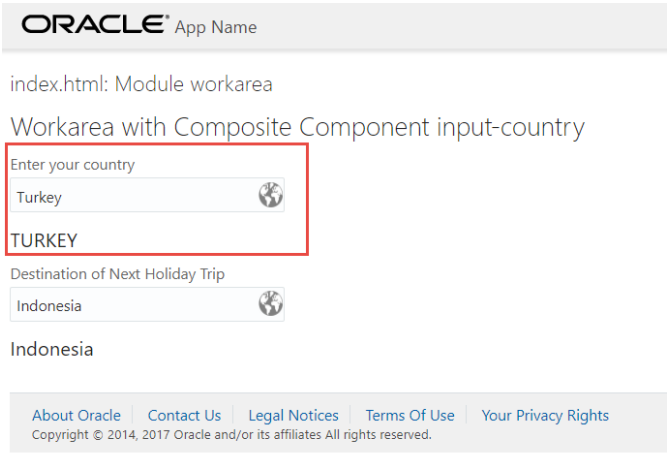
The button invokes a save function that should be defined on the viewModel:

```
          context.props.then(function (propertyMap) {
                //Store a reference to the properties for any later use
                self.properties = propertyMap;
                //Parse your component properties here

                // property countrySelectionHandler may contain a function to be
called when a country has been selected by the user
                self.callbackHandler = self.properties['countrySelectionHandler'];
            });



            // this function writes the selected country name to the two way bound
countryName property, calls the callback function and publishes the countrySelected
event
            // (based on the currently selected country in self.countrySelection)
            self.save = function () {
                if (self.countrySelection && self.countrySelection.name) {
                  // set selected country name on the observable
                  self.properties['countryName'] = self.countrySelection.name;
                  // notify the world about this change
```

```
                if (self.callbackHandler) {
self.callbackHandler(self.countrySelection.name, self.countrySelection.code) }

                }

                // close popup

                $('#countrySelectionPopup' + self.unique).ojPopup("close");

            }//save
```

This snippet also has us read the *propertyMap* from the context parameter that is passed in to the composite component with all the properties defined on the component – in our case in workarea.html. When the *context.props* promise is resolved, the propertyMap is stored in *self.properties* where it can be accessed in the *save* function. The *countryName* property is defined as *writeback* in *component.json*. This means that changes in the property value are communicated back to the observable that is bound to the property – in our case the two observables in workarea.js.

When the user selects a country and presses the Save button, the contents of the input-text field now reflects the selection. What's more: the *h3* element underneath the field is also updated. This happens because it is bound to the same observable as the input-country component and is therefore updated at the same time.

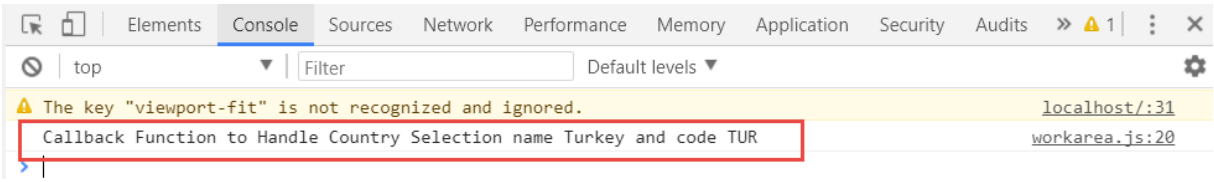


You probably have noticed property *self.callbackHandler* that was set based on a property *countrySelectionHandler* in the component's propertyMap. You may recall that in workarea.html values were defined for this property, bound to functions *handleCountrySelection* and *handleCountry2Selection* in workarea.js. The definition of this property has to be added to component.json:

```
    "countrySelectionHandler": {
      "description": "Provide a function to be called back whenever a country has been selected",
      "type": "function(string,string):boolean"
    }
```

When the user presses the Save button on the popup after a selection was made, the function provided in property *countrySelectionHandler* will be invoked. It should accept two input parameters that will receive the name and code of the selected country. In the next stage we will discuss events as an alternative mechanism for publishing the country selection.

Check the console window in your browser to see the effect of the callback function – a single line of logging is written.



Data binding happens in two directions. Not only does our component pass back the selected country, it also receives the initial country name and it should show that country as pre-selected on the map.

Open viewModel.js and add this code snippet to replace the current openPopup{} function.

```
self.popupFirstTime = true;

self.openPopup = function () {
```

```
    $('#countrySelectionPopup' + self.unique).ojPopup("open");

    // if the map has not yet been initialized, then do the initialization now (this
is the case the first time the popup opens)

    if (!self.map) initMap();

    // set the currently selected country - but only if this is not the first time the
popup opens (and we can be sure that the country vector has been loaded)

    // note: as soon as the vector has finished loading, a listener fires () and sets
the currently selected country ; see var listenerKey in function initMap();

    if (!self.popupFirstTime) {

        self.selectInteraction.getFeatures().clear();

        if (self.properties['countryName'])

            self.setSelectedCountry(self.properties['countryName'])

    } else

        self.popupFirstTime=false;

}//openPopup



self.setSelectedCountry = function (country) {

    //programmatic selection of a feature; based on the name, a feature is searched
for in countriesVector and when found is highlighted

    var countryFeatures = self.countriesVector.getFeatures();

    var c = self.countriesVector.getFeatures().filter(function (feature) { return
feature.values_.name == country });

    self.selectInteraction.getFeatures().push(c[0]);

}
```

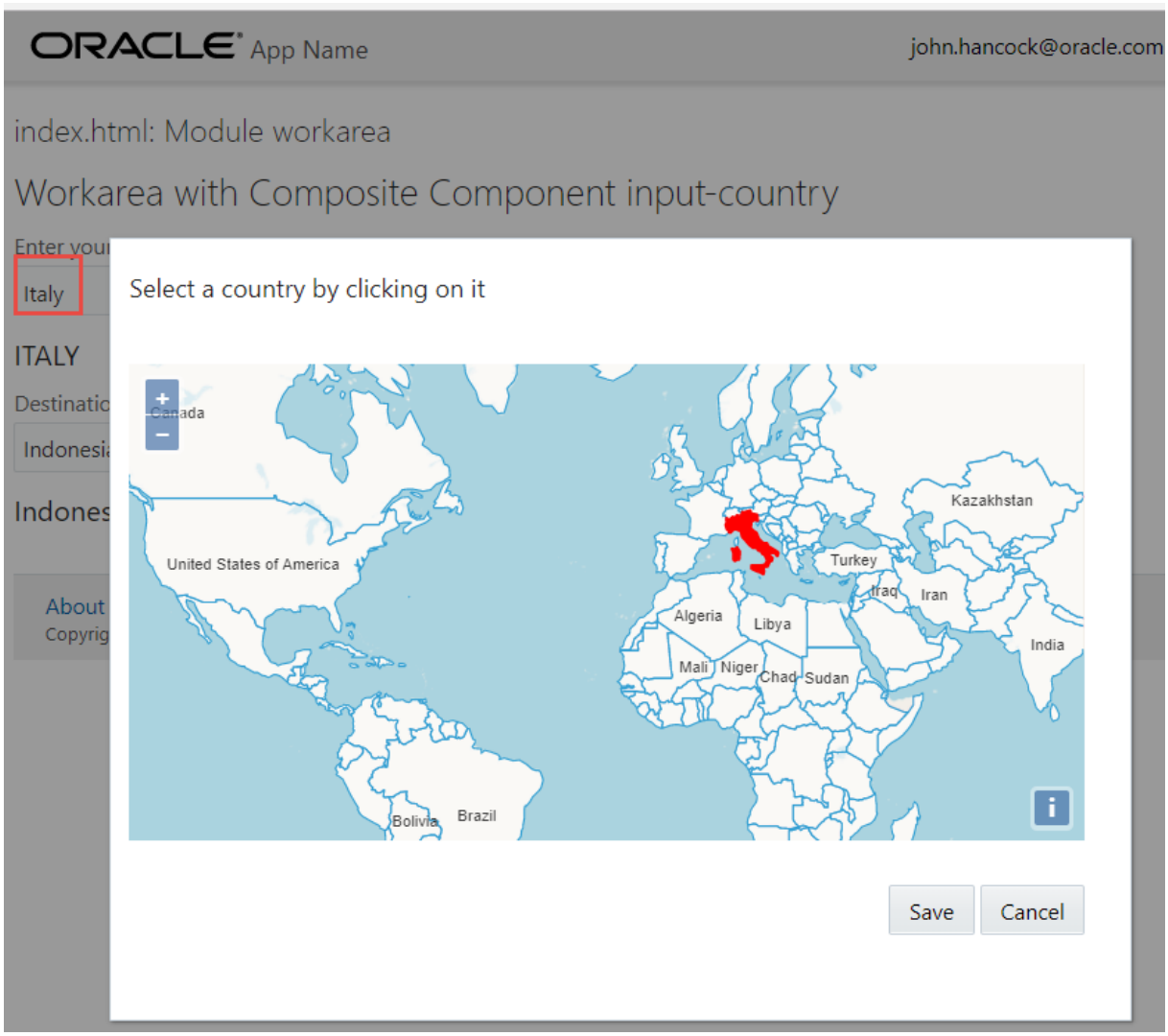
A new function is created – to add the feature for a country from the *countriesVector* to the selected features vector on the *selectInteraction*. This function is called from *openPopup* –but only if the popup has been opened before. Because the *countriesVector* is loaded asynchronously and typically is available only after the popup has loaded, we use an event listener on the *countriesVector* to set the selected country upon the initialization of the popup, the map and the vector.

Insert this snippet right after the definition of the *countriesVector*.

```
    // register a listener on the vector; as soon as it has loaded, we can select the
feature for the currently selected country

    var listenerKey = self.countriesVector.on('change', function (e) {
```

```
        if (self.countriesVector.getState() == 'ready') {
            // and unregister the "change" listener
            ol.Observable.unByKey(listenerKey);
            if (self.properties['countryName'])
                self.setSelectedCountry(self.properties['countryName'])
        }
    });
```

When you now open the popup, the country whose name is the current value of in the input field is preselected on the map. If you type the [correct, init-capped] name of a country into the field and then open the popup, this country will be highlighted.



It would be convenient for our users if they can confirm their selection with a single click on an already selected country – without the need for clicking the Save button. This click event listener on the map object does exactly that (add it near the end of the *initMap* function):

```
// handle the singleclick event- in case a country is clicked that is already selected

self.map.on('singleclick', function (evt) {

    var feature = self.map.forEachFeatureAtPixel(evt.pixel,

        function (feature, layer) {

            var clickCountrySelection = { "code": feature.id_, "name":
feature.values_.name };

            if (self.countrySelection && self.countrySelection.name &&
(self.countrySelection.name == clickCountrySelection.name)) {

                // the current selection is confirmed (clicked on a second time). We
interpret this as: Save the selected country and close the popup

                self.save();

                return;

            }

            return [feature, layer];

        });

});
```

12. Publishing and Handling the CountrySelected Event

Composite components can return data through writeback enabled data bound properties and by calling [back] functions – as we have seen in the previous section. Additionally, composite components can publish events that the consuming viewModel can process as it sees fit.

In workarea.html you may have seen the attribute *on-country-selected* that is bound to the function *self.countrySelectedHandler* that is defined in workarea.js. This function takes an event as it input – and expected this event to contain a property called *detail*. The contents of this property can be defined by us, in any way we see fit.

At present, the input-country component does not publish this custom event. It does however publish an out-of-the-box event called *countryNameChanged* – courtesy of the JET framework for all *writeback* enabled properties. The *countryNameChanged* event also carries a *detail* property that provides both the new and the previous values of the *countryName*. This event is handled by function *handleCountryNameChangedHandler* in workarea.js.

We will now also add support for publishing the custom *countrySelected* event in a few quick steps:

First, add the definition of the *countrySelected* event in component.json:

```
"events" : {

    "countrySelected" : {

      "description" : "The event that consuming views can use to recognize when a
country has been selected",

      "bubbles" : true,

      "cancelable" : false,

      "detail" : {

        "countryName" : {"type" : "string"}

        ,"countryCode" : {"type" : "string"}

      }

    }

  }
```

Next, create function *raiseCountrySelectedEvent* in viewModel.js that will raise the event when called:

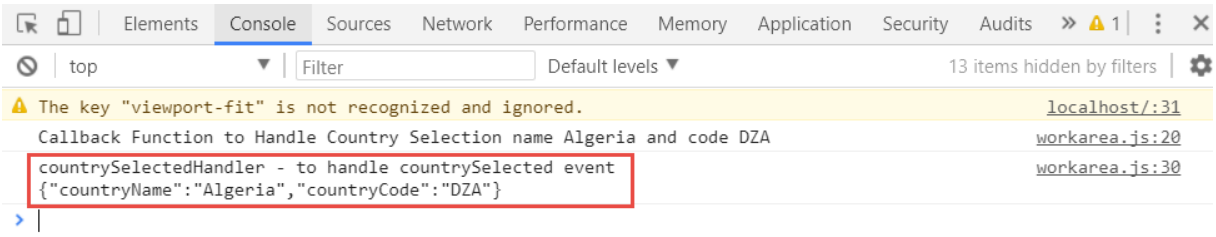
```
self.raiseCountrySelectedEvent = function (countryName, countryCode) {

    var eventParams = {

        'bubbles': true,

        'cancelable': false,
```

```
        'detail': {
            'countryName': countryName
            , 'countryCode': countryCode
        }
    };
    //Raise the custom event on the composite component
    self.composite.dispatchEvent(new CustomEvent('countrySelected',
        eventParams));
}
```

Finally, add this call to function *raiseCountrySelectedEvent* in function *save*:

```
// report the country selection event
self.raiseCountrySelectedEvent(self.countrySelection.name,
self.countrySelection.code);
```

When the user now selects a country, the *countrySelected* event is raised – just as is promised in the specification in component.json. In the console window in the browser, you can inspect the somewhat unexciting event handling by the *countrySelectedHandler* function in the *workarea* viewModel.



13. Package, Distribute and Reuse the Composite Component

The InputCountry component is now fully defined by the sources in the directory jet-composites/input-country.

You can create a zip-file from this directory by way of packaging up the component. This zip-file can be shared with developers working either on their own JET applications or on Visual Builder Cloud applications.

To use the component in a JET application, these steps are required:

- Unzip the zip-file to the src/js/jet-composites directory in the target JET project
- Add a reference to the component in the define() call in the viewModel for the view that consumes the component: 'jet-composites/input-country/loader' (just as in workarea.js in this article)
- Add the markup for the component to the view's HTML document: <input-country> (similar to workarea.html in this article)

14. Further Steps

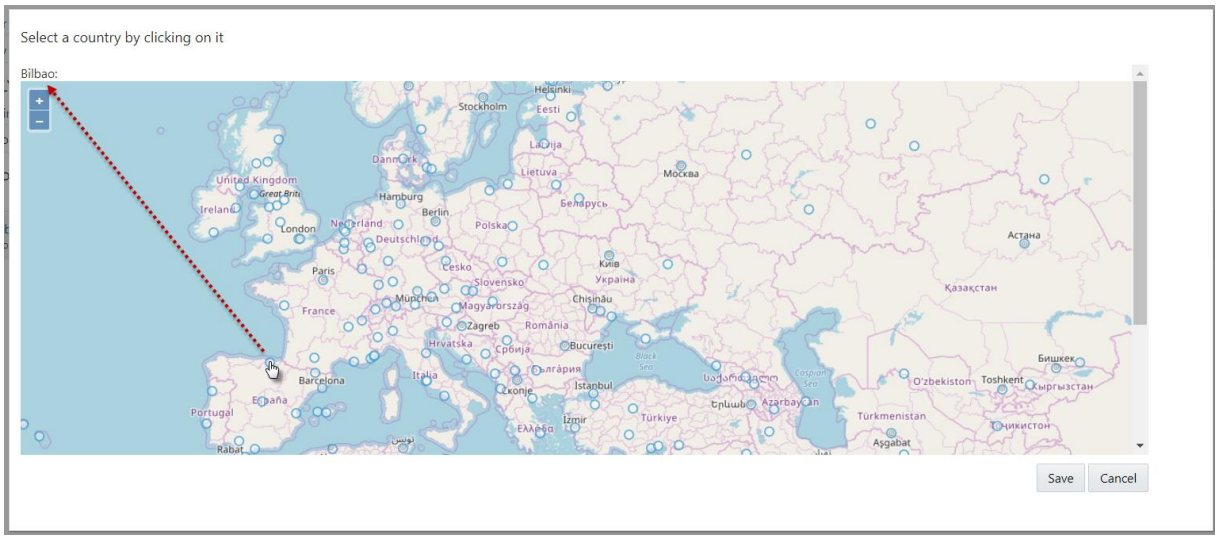
The two layers defined on the map could be reordered – OpenStreetMap tile on top of GeoJSON Countries vector - for a different and in my view nicer look & feel:

The component can easily be further tuned to support other behavior, such as displaying [and selecting] multiple countries, or regions or cities. As a first step, it takes hardly any effort for example to show major cities on the map:

- Download cities.json from https://github.com/mahemoff/geodata/blob/master/cities.geojson
- Replace the countries layer with this snippet for a layer for the cities in the map definition in viewModel.js:

```
, new ol.layer.Vector({
    id: "cities",
    renderMode: 'image',
    source: new ol.source.Vector({
        url: require.toUrl('input-country/cities.geojson'),
        format: new ol.format.GeoJSON()
    }),
})
```

The result is as follows, with a few hundred world cities being displayed on the map.



Good to know: GeoJSON files are available with details for tens of thousands of cities as well as for many other location based data sets. Take a look for example at 1500+ GeoJSON data sets published by the US Government at: https://catalog.data.gov/dataset?res_format=GeoJSON .

Additionally: on http://geojson.io you will find an editor that allows you to easily create your own GeoJSON file. Fun fact: Any *geojson* file in a GitHub repository is automatically rendered as an interactive, browsable map, annotated with geodata.