AMIS SIG ADF and JavaScript–Hands-on

6th July 2017

In this hands-on, you will get a guided tour around the use of JavaScript in ADF 12c rich web application. ADF is a rich web client framework that internally uses plenty of JavaScript to power rich UI components, partial page rendering and other aspects of the user experience. Although ADF is largely declarative and focused mainly on the (Java) server side for creating the user interface, there is a lot to be gained with ADF Faces from using JavaScript. A richer, faster and leaner user experience can be achieved, functionality is sometimes much harder or even virtually impossible to implement with only server side programming, with smart JavaScript we may lower the load on the server thereby improving the scalability of the application, we can leverage the fairly recent HTML5 APIs for client side persistence, file upload, etc. and through JavaScript we are able to embed rich 3rd party components in our ADF Faces UI – from Google Maps to advanced visualization (beyond ADF DVT).

In this session we will discuss how JavaScript can be integrated into out-of-the-box ADF Faces – release 12c. What are the available hooks for injecting JavaScript, how can we programmatically manipulate UI components, how do we achieve client to server (AJAX) and server to client (including push) interaction and how do we embed 3rd party web components. We will discuss the development with JavaScript in JDeveloper and the browser – including source control, debugging, logging, JSON.

The sources for these hands on labs can be found in this GitHub repository: https://github.com/lucasjellema/adf-and-javascript

Plain HTML5 and JavaScript

Understanding JavaScript is perhaps easiest outside the context of ADF Faces. Navigate to directory html5-examples-no-adf in the workshop sources. Open file simple-html-with-static-eventlistener.html in a browser. Click on the input field. Type a few characters. Click on Tab or mouse click outside of the field. Observe what happens when you enter the field (focus event) and when you leave the field (blur event).

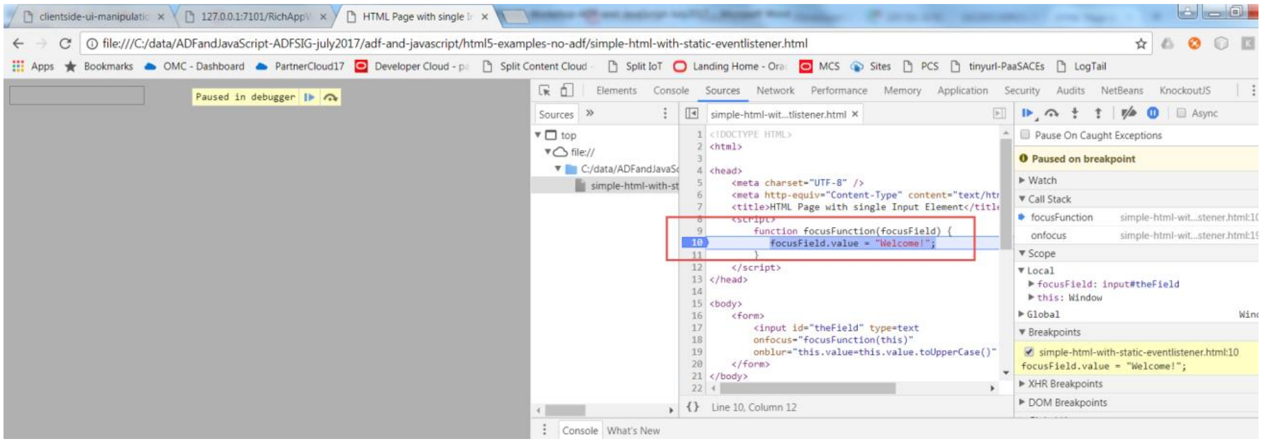
Simple Input Field

Open the page source – preferably using Ctrl Shit I in Chrome or Firefox to bring up the developer tools. Locate the input element. Check out how the focus and blur events are listened for.

Change the text that is shown when the field is entered – from Welcome! to a more sensible default value. Instead of changing the value of the field to uppercase when the user tabs out of the field, add a few characters to the value – for example: “New Value:” + <value in field>. Note: you make these changes using a text editor, save them and reload the document in the browser.

Debugging and Breakpoints

Place a breakpoint in the JavaScript debugger in the function focusFunction():



file:///C:/data/ADFandJavaScript-ADFSIG-july2017/adf-and-javascript/html5-examples-no-adf/simple-html-with-static-eventlistener.html

```
<!DOCTYPE HTML>
<html>

<head>
    <meta charset="UTF-8" />
    <meta http-equiv="Content-Type" content="text/htr
    <title>HTML Page with single Input Element</titl
    <script>
        function focusFunction(focusField) {
            focusField.value = "Welcome!";
        }
    </script>
</head>

<body>
    <form>
        <input id="theField" type=text
        onfocus="focusFunction(this)"
        onblur="this.value=this.value.toUpperCase()"
    </form>
</body>
```

Line 10, Column 12

Paused on breakpoint

Call Stack
focusFunction simple-html-wit...stener.html:10
onfocus simple-html-wit...stener.html:19

Scope
Local
focusField: input#theField
this: Window

Breakpoints
☑ simple-html-with-static-eventlistener.html:10
focusField.value = "Welcome!";

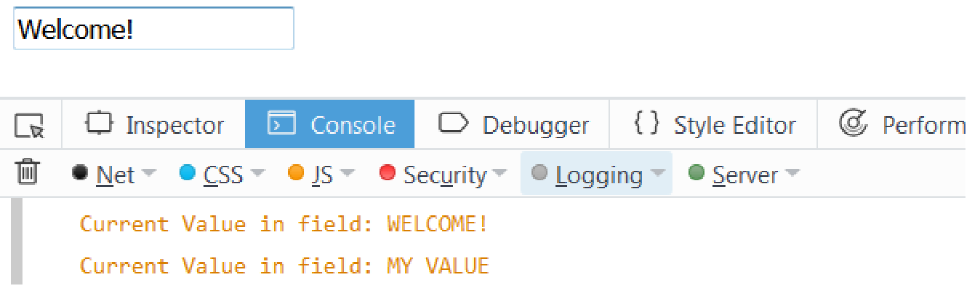
Navigate to the field. The breakpoint should cause the program flow to be interrupted. You get a chance to check on the call stack and the current values of variables – and even manipulate the variables – just as you can in JDeveloper for ADF or Java debugging.

Remember this use of breakpoints, it will come in handy all the time when doing JavaScript development.

Add a line in the function focusFunction to write a line to the console:

```
        console.log("Current Value in field: "+ focusField.value);
```

Save, refresh the browser, navigate to the field, type a value, tab out of the field and go back there again. Now check the text written to the browser console:



```
Current Value in field: WELCOME!
Current Value in field: MY VALUE
```

(screenshot from Firefox)

Browser Events

Open file simple-html-with-event-listeners.html in the browser. It will look and feel pretty much the same as the previous document. When you take at the source for this document, it is a little different. The event listeners are added here at run time – dynamically, programmatically. This process of dynamic initialization I started in the line:

```
window.addEventListener('DOMContentLoaded', init, false);
```

This line adds a callback listener to the Browser: when the DOM content is loaded (all document's elements are loaded), then call the function init(). This function will do the dynamic manipulation of the document and its elements.

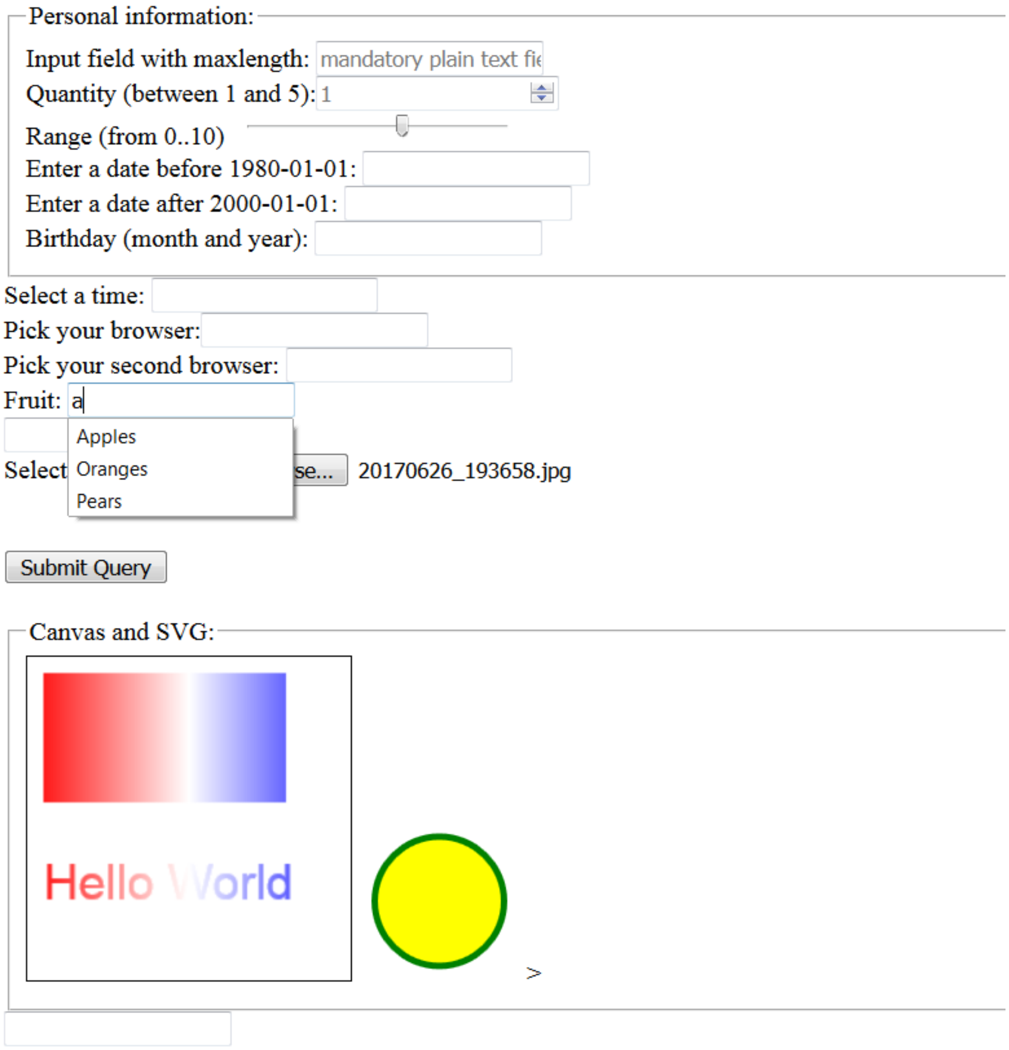
Prevent the event listener for the focus event from being added to the field – it is quite annoying to have the value Welcome! constantly overwriting your own entries…

To get a feel for all events that fly around an input element, open document html5-examples-no-adf/simple-html-to-track-events.html in the browser. Ensure that the console is visible. Click on the input element. Double click, type a few characters and press a few non-character keyboard keys, right mouse click and tab out of the field. Check the console for the events that have come through. Note: each event carries a payload – such as the source (the input field) DOM Element and the key pressed.

HTML5 Components

To see some new HTML5 in action, open document new-html5-components.html – preferably in Chrome (not all elements are supported in Firefox).

Check out the various flavors for input (range, time, week, list => data set), some attributes (such as placeholder) and elements like canvas and svg:



Personal information:

Input field with maxlength: mandatory plain text fie

Quantity (between 1 and 5): 1

Range (from 0..10)

Enter a date before 1980-01-01:

Enter a date after 2000-01-01:

Birthday (month and year):

Select a time:

Pick your browser:

Pick your second browser:

Fruit: a

Select

20170626_193658.jpg

Canvas and SVG:

Check out the source of the document to see what is done in HTML to make this happen (not very much, that is the beauty).

You may also want to look at html5-fileupload.html to see the File API in action.

First Steps with JavaScript in ADF Faces

The ADF 12c Web Application in directory RichAppWithJavaScript in the workshop resources contains examples for many different use cases for JavaScript in ADF. It is useful to have this application as a reference when trying out things in your own application.

Note: the application RichAppWithJavaScript is standalone. It does not require a database connection or any external services. It uses the Placeholder Datacontrol to provide some data bindings. Use of Java is limited to a few managed beans that respond to client side events in partial page requests.

Create your own brand new ADF Web application MyApp. You do not need a Model project – but it is OK to have one.

Create a new page – mypage – and add two InputText elements.

Add a clientListener that responds to the blur event (type) to the first element. Specify removeDashes for the method attribute.

Add an af:resource of type javascript as a child to the af:document tag. Add a snippet of JavaScript inside the af:resource:

```
<af:resource type="javascript">

     function removeDashes(event) {

        var inputComp = event.getCurrentTarget();

        inputComp.setValue(inputComp.getSubmittedValue().replace(/-/g, ''));

     }

 </af:resource>
```

Run the page and see what happens when you enter a value that contains dashes into the input field and then tab out of the field.

Now create a JavaScript library called mypage.js – a new resource of type JavaScript under public-html/resources/js. Copy and paste the function removeDashes() to this new library. Remove the af:resource element from mypage and replace it with the following element:

```
<af:resource type="javascript" source="/resources/js/mypage.js"/>
```

Rerun the page. Check in the debugger in the browser – Ctrl plus shift plus I – that the library is loaded to the browser and that the function removeDashes() is invoked.

This makes the exact same function(ality) available in the page, only this time provided through a standalone JavaScript library that can easier be managed and edited as well as reused. The only downside: insight the <af:resource> tag, we can have EL expressions. In JavaScript libraries, we cannot.

Add client listener to the second inputText element, for the focus event. Set the method to highlight. Add a function highlight to the library mypage.js:

```
function highlight(event) {

    var inputComp = event.getCurrentTarget();

    inputComp.setProperty("contentStyle", "background-color:yellow;");

}
```

Save all changes and refresh the page in the browser. Navigate to the second input element, and see it being highlighted.

Note: how can you undo the highlighting when the user navigates out of the field?

We would like the second field to be updated when the first field is changed. Let’s say the second field should contain the uppercase value of the first field.

To achieve this, we need to extend function removeDashes() that is invoked at the blur event for the first field. We need to find the second inputText component and set its value. That can be done with these two lines:

```
var theField = inputComp.findComponent(<ID for the second inputText element>);

theField.setValue(inputComp.getSubmittedValue().toUpperCase());
```

Add these lines to the function, save changes and refresh the page in the browser. Check if indeed the second inputText is refresh with changes in the first. Note: we have use one of the ways to locate an ADF Rich Client component – the JavaScript counterpart of the JSF UI Component in Java. When we look for components in the same ‘naming container’ as a component we already have a reference to – frequently because that component is the source of an event – we can use this method to find components we want to interact with in our code. Another option is to search components in the page using their absolute identifier – which included the id that we specified in the JSF source as well as all enclosing naming containers, separated by double colons “::”.

We can be even more aggressive in our synchronization: instead of refreshing only when the focus moves away from the first inputText element, we can refresh the second field after each key-press. To do this, add a clientListener on the first inputText element for the keyUp event. Set the method for this event to synchronizeField. Add a new function synchronizeField(event) in the mypage.js library. Move the two lines you just added to removeDashes() to this new function.

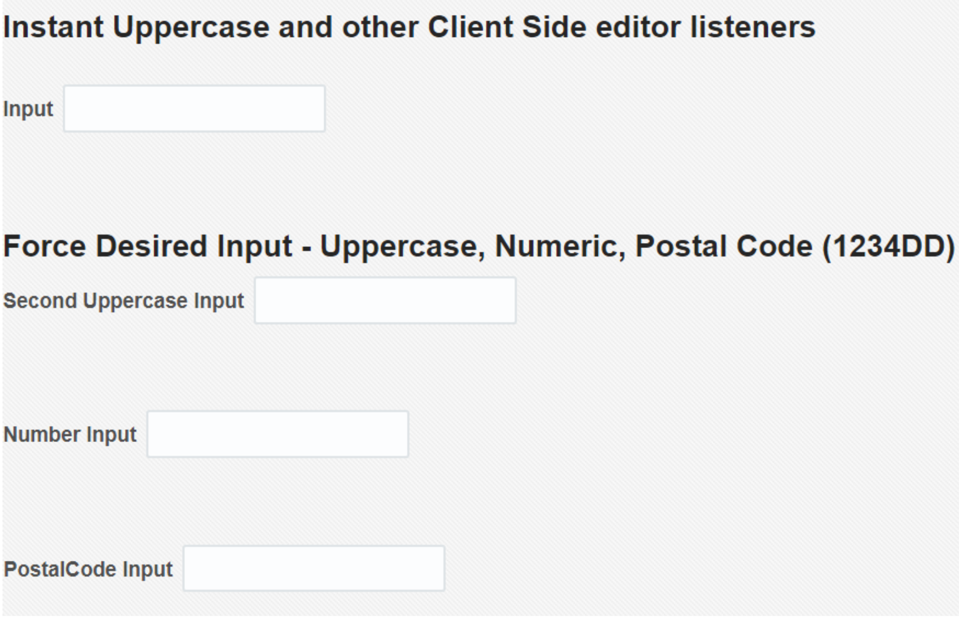
Save the changes, refresh the page. Every character you type in the first inputText should now be echoed in the second inputText element.

Inspect Demo Application

Open the demo application RichAppWithJavaScript in JDeveloper. Run the application. Take a look at the Simple Input page. Try out the instant summation by entering numeric values in the input fields A and B. Click on the Required Field, then try to move out of the field without providing a value. Tough! The final field – with special validation – engages the server for doing the validation, using a special construct that we will take a closer look at later on.

For now, in JDeveloper open page simple-input.jsf and library AppLibrary.js and check out how the summation is implemented and how the required field validation that does not let you escape has been set up.

Next, open page Instant Uppercase in the browser.



Instant Uppercase and other Client Side editor listeners

Input

Force Desired Input - Uppercase, Numeric, Postal Code (1234DD)

Second Uppercase Input

Number Input

PostalCode Input

Type *world* in the first field. Then type *hello* in front of world. Mmmm, not so easy.

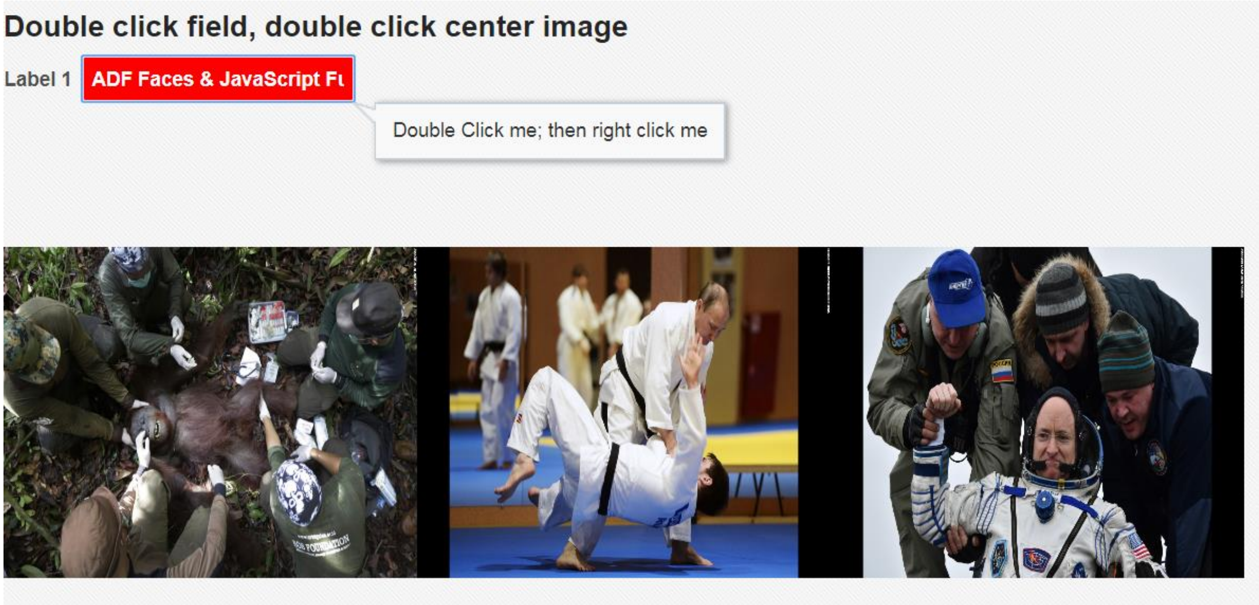
The second field has similar functionality – but slightly smarter. Do the same again: type world, then add hello in front of it.

Navigate to the Numeric field. Try to type a text into the field. Try to type numbers. Try to type H3ll0 W0r1d.

Navigate to the Postal Code field. Try to enter something that is not a string of format 1234DD.

Now back in JDeveloper, open page simple-input-instant-upper.jsf and its associated JavaScript library AppLibraryInstantUpper.js. Inspect how the various behaviors you have just tried out have been implemented – largely using keyUp and keyDown event listeners and all strictly client side.

Open the page Client Side UI Manipulation in the browser:



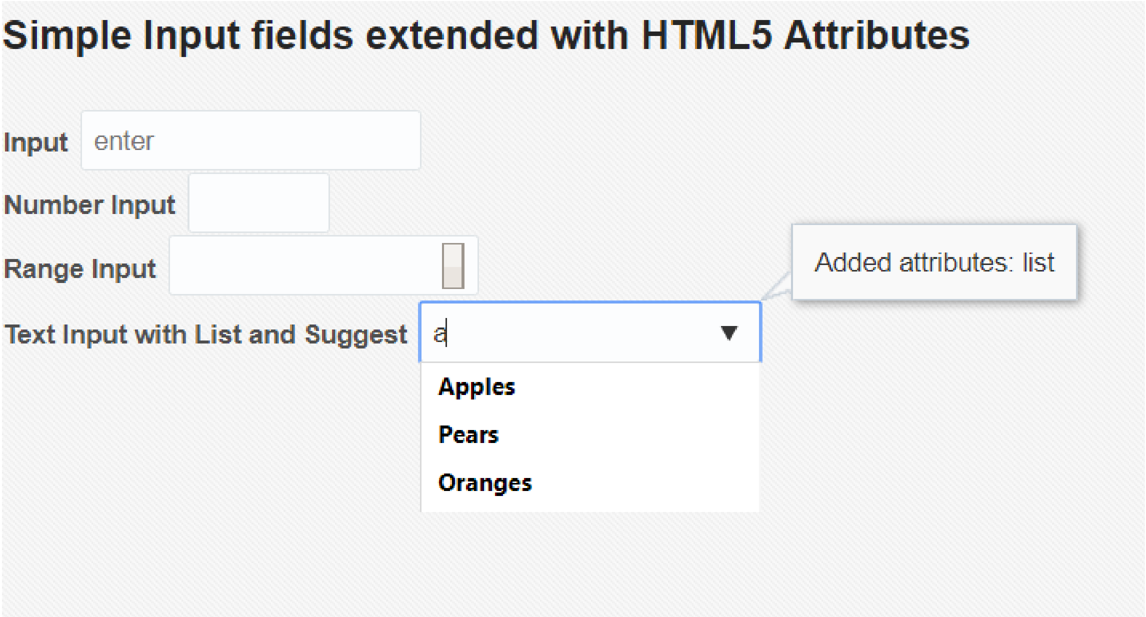
Double click field, double click center image

Label 1 ADF Faces & JavaScript Fu

Double Click me; then right click me

Double click on the input field. Then right click on that field. Double click on the Judo image. And then double click the same image again. What you see here are all examples of client side component manipulation. We can manipulate what the user experiences quite easily. This can be done through the server side – but to get a smooth experience (no network latency, no server processing) and reduce the load on the server, we can also implement this functionality in the client.

Take a look at page HTML5 Attributes and Listeners in the demo application.



Simple Input fields extended with HTML5 Attributes

Input enter

Number Input

Range Input

Text Input with List and Suggest a

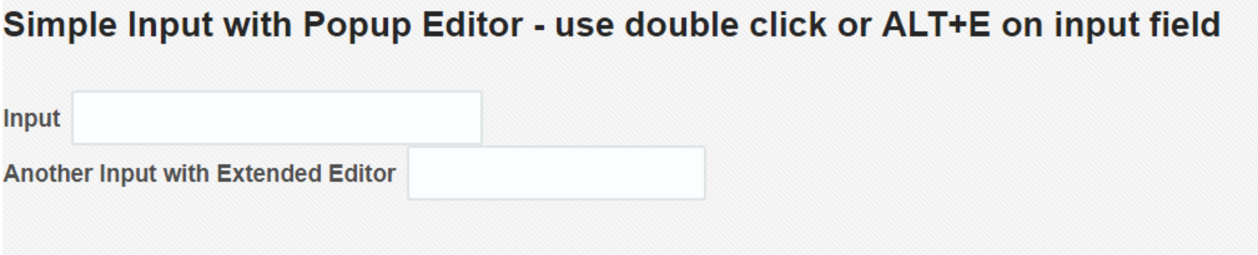
Added attributes: list

In this page, a number of standard ADF Input Text components have been manipulated dynamically from JavaScript executing right after the page has loaded. By adding attributes and listeners to the HTML DOM elements generated for these ADF Faces components, we can add HTML5 functionality that is not supported out of the box in ADF Faces. Especially the data list (Google Suggest op plain input text element) is interesting, as is the placeholder attribute.

Client Side Popup Editor

Manipulating popups is frequently useful to provide additional information to users. Popups can be opened using the showPopupBehavior tag and also programmatically, in Java and in JavaScript.

Open page Popup Editor.



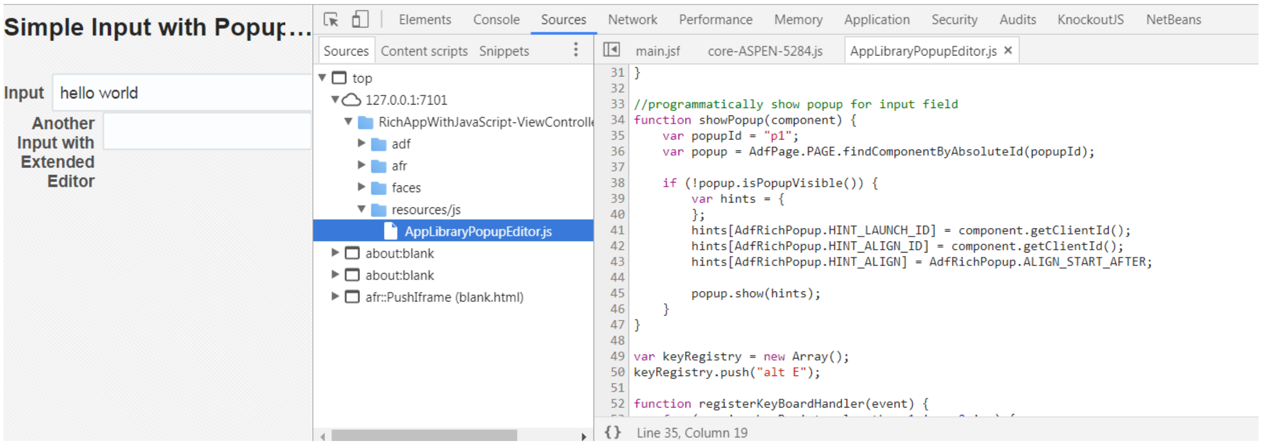
Simple Input with Popup Editor - use double click or ALT+E on input field

Input

Another Input with Extended Editor

Type *hello* in the first input field. Then double click on the field or type Alt+E. A pop up appears with a more elaborate editor. Append *world* to *hello.* Click on OK. See how the value is copied back from the editor to the original field. Try the same thing from the second input field. Also see what happens after editing the value in the popup and closing the popup with Cancel instead of OK.

Open the Developer Tools in the browser – Ctrl plus Shift plus I.



Input hello world

Another Input with Extended Editor

```
}

//programmatically show popup for input field
function showPopup(component) {
    var popupId = "p1";
    var popup = AdfPage.PAGE.findComponentByAbsoluteId(popupId);

    if (!popup.isPopupVisible()) {
        var hints = {
        };
        hints[AdfRichPopup.HINT_LAUNCH_ID] = component.getClientId();
        hints[AdfRichPopup.HINT_ALIGN_ID] = component.getClientId();
        hints[AdfRichPopup.HINT_ALIGN] = AdfRichPopup.ALIGN_START_AFTER;

        popup.show(hints);
    }
}

var keyRegistry = new Array();
keyRegistry.push("alt E");

function registerKeyBoardHandler(event) {
```

(screenshot Chrome)

Open the source AppLibraryPopupEditor.js.

The function init is called when the page is first loaded. This function calls registerKeyBoardHandler to make sure that Alt+E is trapped. When Alt+E is caught, the callback function is invoked. It checks if the keycode is indeed Alt+E and if the component with current focus is an input text component. If so, function showPopup is invoked to launch the popup, associated with the launching input component.

When the popup is launched, function extendedFieldPopupOpening () is invoked. It remembers the launching component and copies the value from the launching component to the extendedEditor field in the popup.

Function handleDialog() is called when the popup is closed. If the popup is closed with OK, then function copyExtendedEditorValueToOriginal () is called to copy the value from the extended editor field in the popup back to the original component.

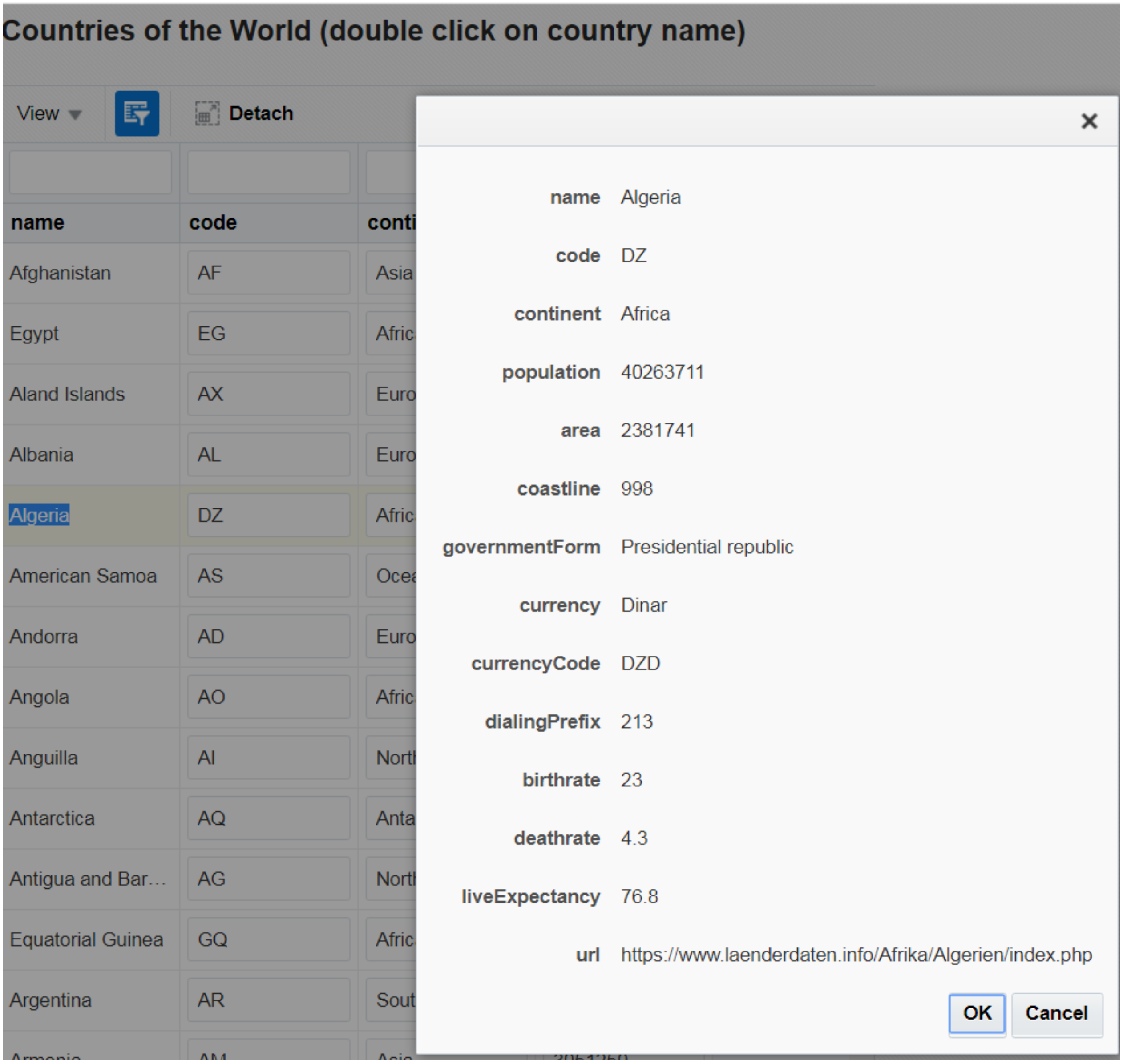
With the use of breakpoints, you can easily follow this flow in the browser.

Open popupeditor.jsf in JDeveloper. See how the init function is referenced from the *load* client listener on the document. Also see how the popup can be opened through double click on both input elements. Finally, check out the client listeners on the Popup (when opening) and Dialog (when closing) elements.

A similar approach can be used to show a popup that has detailed information about the current field(‘s value).

Country Table

Open the page CountryTable in the demo application. This page uses the Popup component to show details about a country:



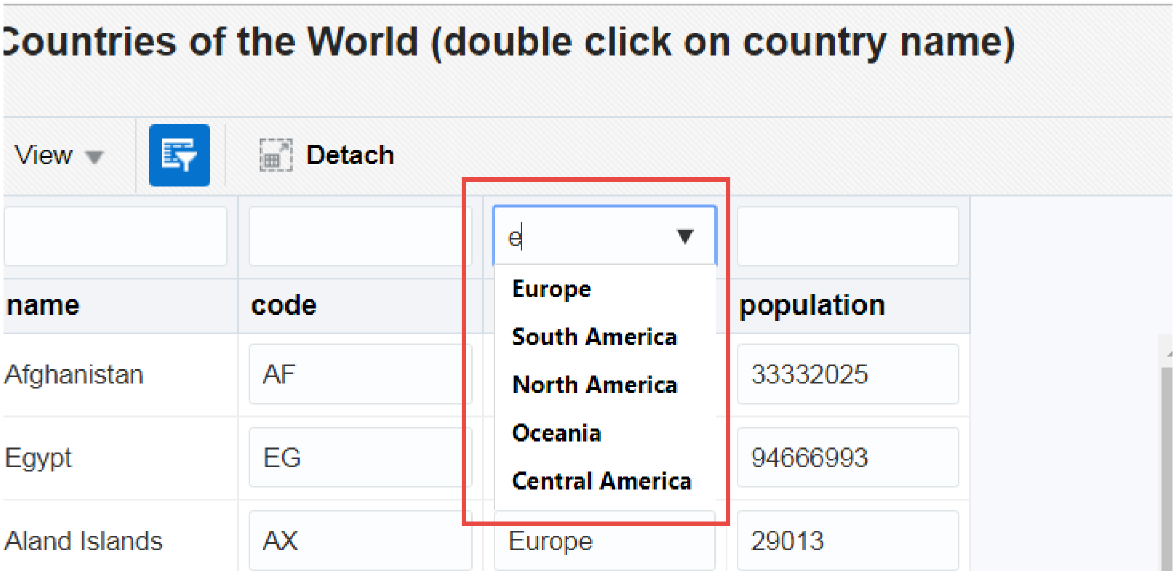
Countries of the World (double click on country name)

name	code	conti
Afghanistan	AF	Asia
Egypt	EG	Afric
Aland Islands	AX	Euro
Albania	AL	Euro
Algeria	DZ	Afric
American Samoa	AS	Ocea
Andorra	AD	Euro
Angola	AO	Afric
Anguilla	AI	Nortl
Antarctica	AQ	Anta
Antigua and Bar…	AG	Nortl
Equatorial Guinea	GQ	Afric
Argentina	AR	Sout
Armenia	AM	Asia

name	Algeria
code	DZ
continent	Africa
population	40263711
area	2381741
coastline	998
governmentForm	Presidential republic
currency	Dinar
currencyCode	DZD
dialingPrefix	213
birthrate	23
deathrate	4.3
liveExpectancy	76.8
url	https://www.laenderdaten.info/Afrika/Algerien/index.php

Check out the sources Countries.jsff and AppLibraryCountriesTable.js. Note: the data in this page is produced by a data binding based on a Placeholder Data Control that was created from a CSV file (countries.csv, also in the demo sources).

Another interesting feature in this page is the use of HTML5 list on the continent filter:



Countries of the World (double click on country name)

name	code		population
Afghanistan	AF		33332025
Egypt	EG		94666993
Aland Islands	AX	Europe	29013

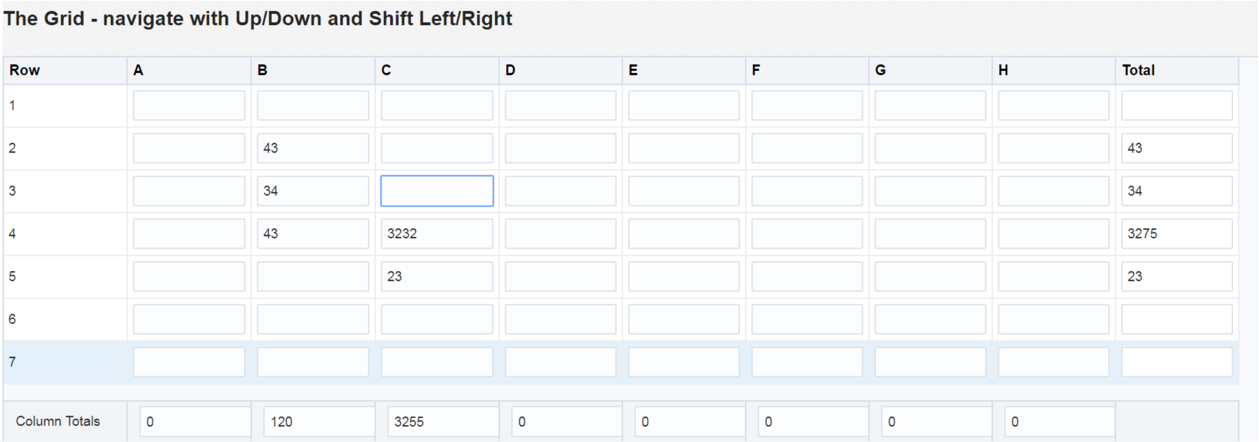
When the user navigates to the filter field – a normal inputText component- the component starts behaving as an auto suggest; type any letter and all values that contain the letter are presented. You still type a value that is not in the list – these are really just suggestions. Because it is all client side, the interaction is very smooth and quick.

In this case the values of the continent names are hard coded in the JavaScript library. Using the databound JSON mechanism – soon to be discussed – it would be simple to dynamically derive these values.

The Grid

Tables are complex. And very useful on many occasions. In The Grid, there are some examples of how we can interact on the client side with a table component and its constituents. In this particular example – there are row and column totals maintained on the client. Additionally, navigation through the grid can be done using arrow keys (up and down plus Shift Left and Shift Right); note that this arrow based navigation support wrap around – from top back to bottom and from end of row back to beginning.

Open the page The Grid and play around:



The Grid - navigate with Up/Down and Shift Left/Right

Row	A	B	C	D	E	F	G	H	Total
1									
2		43							43
3		34							34
4		43	3232						3275
5			23						23
6									
7									
Column Totals	0	120	3255	0	0	0	0	0	

Open TheGrid.jsf in JDeveloper, and its associated JavaScript library: AppLibraryTheGrid.js.

It should not come as a surprise that as before the page contains a client listener for the *load* event that calls a function in the library that registers key board handlers – for the arrow key operations. The most complex function we see is the callback() function that is invoked whenever one of the registered keyboard events fires. Depending on the event, and based on the current cell, the target for the navigation is determined and the focus is moved to this target.

To recalculate the row and column aggregates, each cell has a valueChange listener that invokes the aggregate() function in the library. The aggregates are calculated for the current row and column and the totals are updated.

Can you change the aggregate() function to have it calculate the average instead of the sum of the row and column?

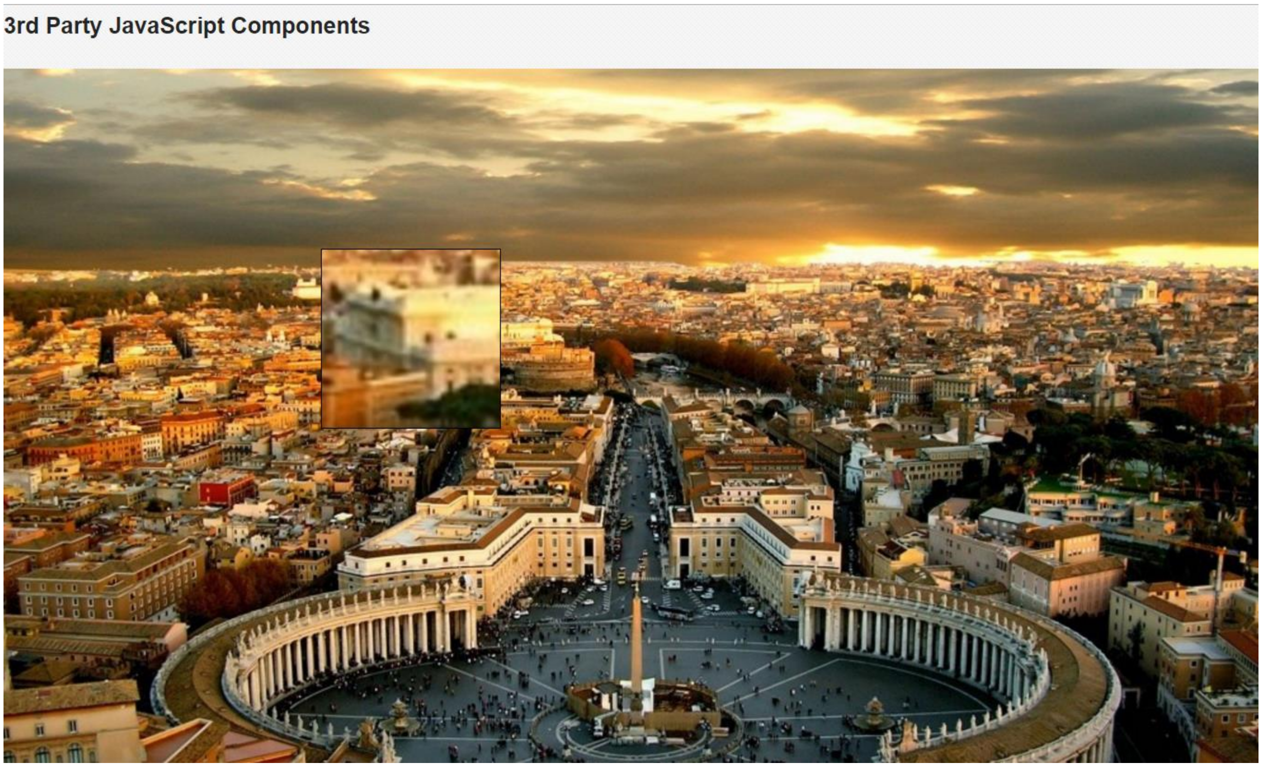
Even more interesting: can you extend the page with radio group or dropdown list that allows the user to select which aggregation function to employ? And when that selection is changed, updating all totals?

Third Party Component Integration

JavaScript is one of the most ubiquitous programming languages. The number of JS libraries is really large and the number of UI components that is out there, ready to be picked up and reused is pretty overwhelming. Of course using a 3rd party component is not the first thing we do with ADF Faces – the framework provides a pretty extensive collection out of the box already. However, sometimes there is something special that really makes sense to add – such as Google Map , a special editor (formulas, drawings), special visualizations and more.

In a later section, we will look at adding databound third party components. Let's first take a look at a fairly simple example: an Image Zoomer (based on blog article: `http://www.awasthiashish.com/search/label/javascript#sthash.R1Xw2cG4.00b48pG9.dpbs`).

Open page *Integrate 3rd Party Components* from the main menu. It presents an image of Rome and the Vatican. This picture has been enriched with the image zoomer component, a 3rd party component powered by jQuery. Both this component and jQuery have been added to the page.



3rd Party JavaScript Components

Check the sources in thirdparty-js-component.jsf and AppLibraryThirdParty.js. You will find that it does not take much to make this (somewhat crude) integration.

Client to Server

In addition to declarative PPR some autoSubmit and partialNavigation, ADF Faces supports programmatic (AJAX) calls to the server. Using serverListeners, we can implement such notifications from client to server in an elegant, structured manner.

Open the Client to Server demo in the browser. Click on the Help button. A separate window should open with relevant help. Sort of. Now check the console in JDeveloper. A line should be written there, because the server was notified of this call for help. Without blocking the user and the new browser window or tab in any way.

Click on Alt+T. The time displayed in the page should be refreshed. Again, the server was notified of the request for a fresh time and responded with the fresh time value and adding the Time UI component as a partial target.

The essence of the implementation is in:

- a clientListener (to respond to help button press and Alt+T keyboard click)
- a serverListener (that defines the communication to a specific managed bean on the server)
- the serverside managed bean (to respond to the client event)
- the Partial Page Refresh that brings the client event to the server and takes the response back the client where it can refresh UI components and execute additional custom JavaScript

Open page client-to-server.jsff , JavaScript library AppLibraryClient2Server.js and Java Class Client2ServerBean.

The page contains the help button which has clientListener (call JavaScript function when the button is pressed) and a clientAttribute (provide additional context to the JavaScript function about the button component). Also has the serverListener – the channel to the server side managed bean client2serverBean and its callForHelp method that handles this client event.

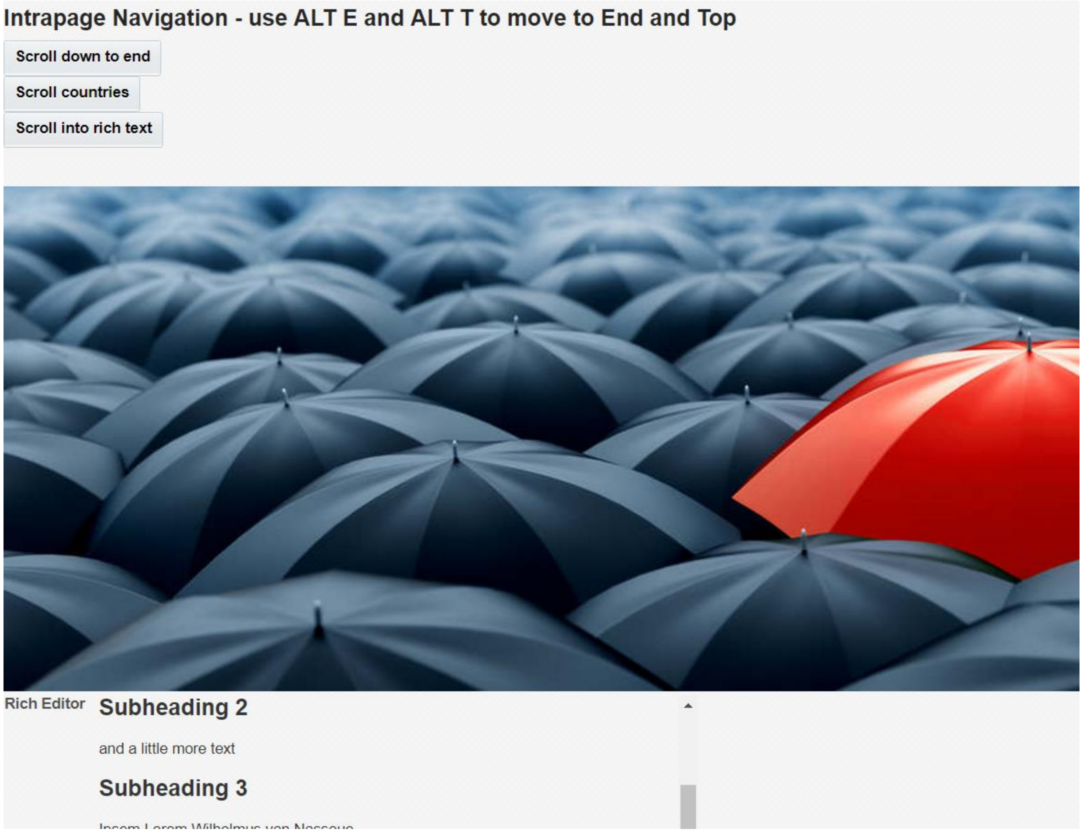
Check how client side function helpCallListener() creates an event to send to the server (with the value of the clientAttribute as payload) and subsequently engages the serverListener to channel the event ‘to the other side’. The server side implement is straightforward in this case: method callForHelp receives the event and write a line to the console.

The time refreshing capability is configured using a keyboard event listener – function callback – that waits for Alt+T events and then engages the serverListener refreshTimeListener. The server side for this channel – refreshTime – adds the timer field as partial target to the PPR response and as a result, this component is refreshed in the client with the latest time provide by method getTime() on the bean.

Note: see how the UI Component Binding is defined for timerField ; this is the correct way – that replaces the original approach using UIComponent instead of ComponentReference. (see: http://www.jobinesh.com/2011/06/safely-storing-uicomponent-component.html)

Intra-Page Navigation

Check out the demo for Intrapage Navigation – using the scrollComponentIntoViewBehavior tag.



Intrapage Navigation - use ALT E and ALT T to move to End and Top

Subheading 2

and a little more text

Subheading 3

Ipsom Lorem Wilhelmus van Nassoue

Open the page – intrapage-navigation.jsf and associated JavaScript library AppLibraryIntrapage.js to see how this navigation has been implemented. Note how we can even scroll into a rich text editor – jumping directly to a heading inside that component.

Also note how we scroll as a result of user actions (pressing a button) and programmatically (by pretending a user has activated a button). Note that the button we programmatically activate does not even have to be visible.

Client Side Event Bus

Copy the file AppLibraryClientEventbus.js from directory RichAppWithJavaScript\ViewController\public_html\resources\js in the demo application to the js directory in MyApp where you also created mypage.js.

Create a new page called ClientEventBus(.jsf). Add this element as child of the af:document tag:

```
<af:resource type="javascript"
source="/resources/js/AppLibraryClientEventbus.js"/>
```

Create two bounded taskflows – tf1 and tf2 – and add a single View in each – view1 and view2 respectively.

Create new JavaScript libraries tf1.js and tf2.js and associate these with the respective views/page fragments, adding these element as direct child of ui:composition:

```
<af:resource type="javascript" source="/resources/js/tf1.js"/>
<af:resource type="javascript" source="/resources/js/tf2.js"/>
```

Add this snippet to tf2.js:

```
function publishColorSelection(color) {
          publishEvent("color-picker-tf-colorSelectionEvent",
          {
              "selectedColor" : color
          });
      }

      function clickRed(evt) {
          evt.cancel();
          publishColorSelection('red');
      }

      function clickYellow(evt) {
          evt.cancel();
          publishColorSelection('yellow');
      }

      function clickBlue(evt) {
          evt.cancel();
          publishColorSelection('blue');
      }
```

And add this panelgroup to view2:

```
                <h:panelGroup id="pg1">
                    <af:button text="Red" id="b1" inlineStyle="color:red;"
partialSubmit="false">
                        <af:clientListener type="action" method="clickRed"/>
                    </af:button>
                    <af:button text="Yellow" id="b2" inlineStyle="color:yellow;"
partialSubmit="false">
                        <af:clientListener type="action" method="clickYellow"/>
                    </af:button>
                    <af:button text="Blue" id="b3" inlineStyle="color:blue;"
partialSubmit="false">
                        <af:clientListener type="action" method="clickBlue"/>
                    </af:button>
                </h:panelGroup>
```

Open view1 and add this selectOneRadio:

```
<af:selectOneRadio label="Color of your choosing" id="colorRadio"
                        clientComponent="true" styleClass="colorRadio">
                    <af:selectItem label="Red" value="red" id="si1"/>
                    <af:selectItem label="Yellow" value="yellow" id="si2"/>
                    <af:selectItem label="Blue" value="blue" id="si3"/>
                </af:selectOneRadio>
```

In tf1.js, add this snippet:

```
subscribeToEvent("color-picker-tf-colorSelectionEvent", handleColorSelection);

function handleColorSelection(payload) {
    console.log("ColorSelectionEvent consumed " + JSON.stringify(payload));
    var color = payload.selectedColor;
    console.log("selected color " + color);
    // invoke function to handle server side synchronization
    informServerAboutColorSelection(color);

    // get hold of ADF Faces Client Side AdfRichSelectOneRadio object
    // we use a trick that assumes only one element is rendered in the entire
page with a css style class colorRadio
    // (that also means that page fragment can be included only once and so can
the taskflow that this fragment is included in;
    // for this simple example, that is an acceptable limitation)
    var htmlElementForRadio = document.getElementsByClassName("colorRadio");
    // from the HTML element we get the element id and use that as the input for
the findComponentByAbsoluteId call that returns the actual ADF Faces
AdfRichSelectOneRadio object
    var radio =
AdfPage.PAGE.findComponentByAbsoluteId(htmlElementForRadio[0].id);
    // the radio group has multiple select items; their labels and values can be
retrieved
    var sis = radio.getSelectItems();
    // now we can lookup the value of the select item with the label
corresponding to the selected color:
    for (i = sis.length - 1;i >= 0;i--) {
        var si = sis[i];
        if (si.getLabel().toLowerCase() == color) {
            radio.setValue(si.getValue());
            break;
        }
        //if
    }
    //for
}
//handleColorSelection
```

Open page ClientEventBus. Add a panelSplitter component to the page. Add taskflow tf1 to the first facet of the panelSplitter and tf2 to the second. Now run the page.

What you have put together is the following:

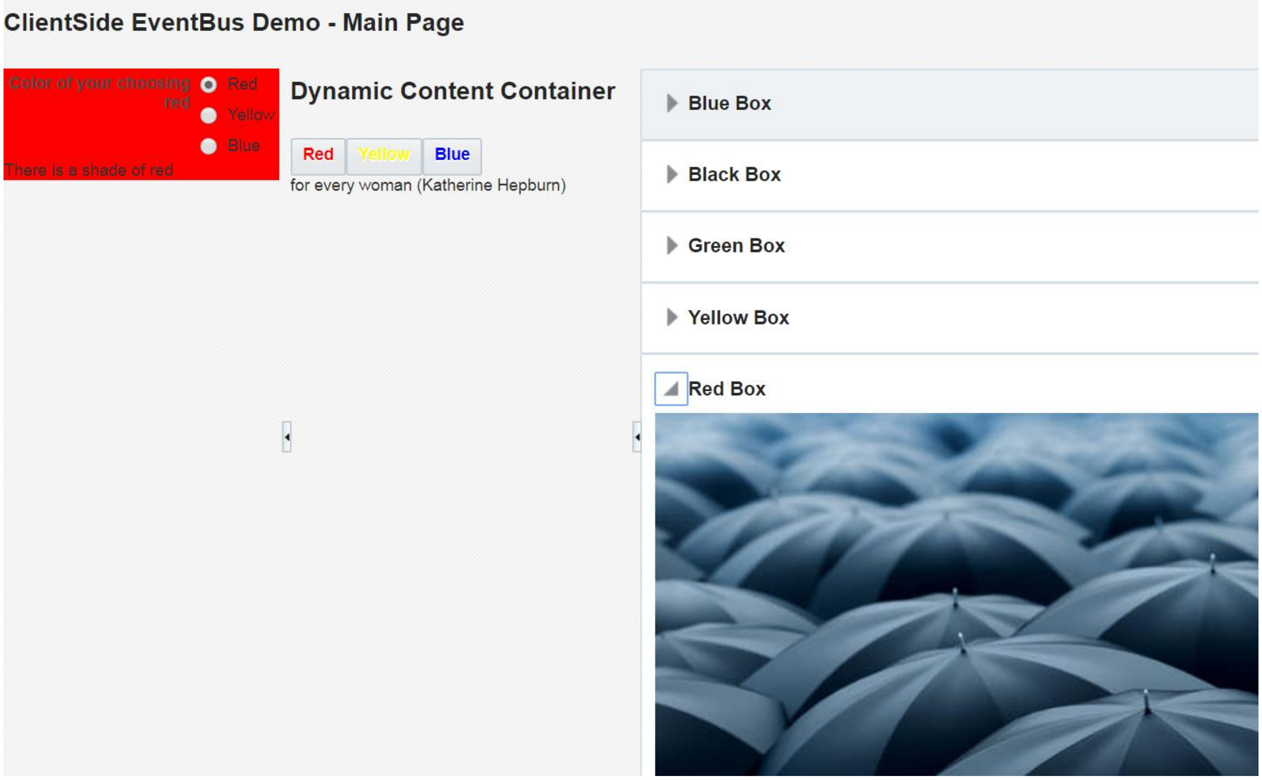
Taskflow 2 contains buttons that each represent a color (selection). When pressed, each button will publish a color-picker-tf-colorSelectionEvent to the client event bus.

Taskflow 1 has registered for this event when its JavaScript library was loaded. Whenever the client event bus receives this event, a call is made to the registered callback function: handleColorSelection. This function reads the event payload, locates the radio group component and updates that component with the color that was selected.

What is implemented now is quite interesting: client side components from two different encapsulated, mutually isolated taskflows interact on the client side without being aware of each other. The client side event bus decouples the (client sides of the) two taskflows. Note: this is similar – but much simpler – than the server side contextual events. Also note that this implementation is still pretty rudimentary.

Client Event Bus, Server Side Operations and Partial Page Refresh

A more elaborate implementation using the client event bus is found in the Client Event Bus demo


ClientSide EventBus Demo - Main Page

Color of your choosing red

● Red
○ Yellow
○ Blue

There is a shade of red

Dynamic Content Container

for every woman (Katherine Hepburn)

Open the Client Event Bus page in the demo application in the browser.

Select a color by pressing one of the buttons. See what happens:

1. radio group is synchronized
2. some quote is written – part in the left zone of the page and part in the center area
3. the corresponding box is disclosed and scrolled into view – in the right area of the page

This page contains four taskflows (the one with the buttons is nested inside the content container in the center area). They are not aware of each other. Pressing a button publishes a client side event, that results in updating the radio group.

The JavaScript function consuming the this event also reports the newly selected color to the server (function `informServerAboutColorSelection() in library AppLibraryMenuTaskflow.js) using a serverListener on the radio group.`

The `server side managed` bean menuBean (class MenuBean) handles this clientEvent. It extracts the color from the clientEvent’s payload, selects a quote befitting the color, adds the ColorPanelGridLayout component as a partial target and write a snippet of JavaScript to the PPR response. This snippet publishes another client event to the client event bus: newBackgroundColorEvent. This event contains the color as well as the selected quote in its payload.

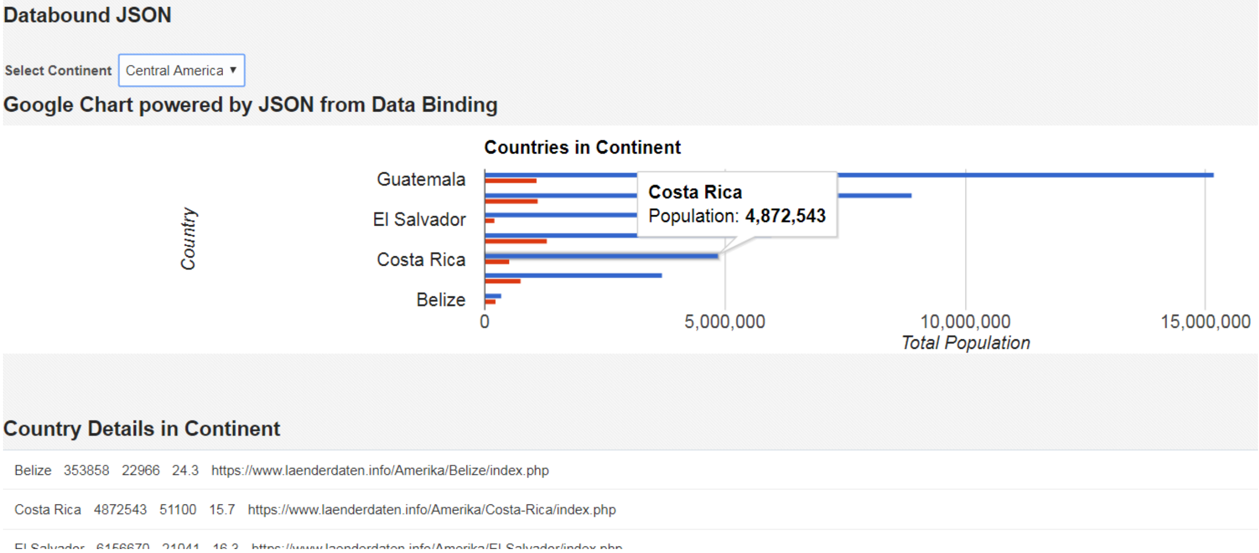
The panelbox taskflow (right side of the page) has subscribed to this event (in library AppLibraryPanelBoxTaskflow) and reacts to it when the PPR response is delivered to the browser; the panelBox corresponding to the color is disclosed and scrolled into view.

Taskflow contentContainer contains a JavaScript snippet that also registers to the newBackgroundColorEvent; when that event is consumed, the second part of the quote in the payload of the event is set on the outputText component statementPart2.

What should be done in order to support an additional color – say green? Can you get that done?

Databound JSON and 3rd Party Component Integration Part 2

Open the Data Bound demo page:



Databound JSON

Select Continent: Central America ▾

Google Chart powered by JSON from Data Binding

Country Details in Continent

Belize	353858	22966	24.3	https://www.laenderdaten.info/Amerika/Belize/index.php
Costa Rica	4872543	51100	15.7	https://www.laenderdaten.info/Amerika/Costa-Rica/index.php
El Salvador	6156670	21041	16.3	https://www.laenderdaten.info/Amerika/El-Salvador/index.php

This page has two iterator bindings – one corresponding with the Continents collection on the Placeholder Data Control, the other one with the detail CountriesInContinentSelection. Note: this would be exactly the same if the Data Control was an ADF BC Data Control based on an Application Module with ViewObjects and detail ViewObjects (detail end of ViewLink and defined as detail in the AM's Data Model).

The dropdown choice element is powered by the master iterator on continents. When a selection is made, the detail iterator is synchronized accordingly.

A key element in this example is a mechanism to make data from ADF Data Bindings available as JavaScript in memory data structures through the use of JSON as intermediate carrier.

The JSF page is defined in databound-json.jsf. This page has an associated page definition with data bindings for two collections in the placeholder data control. Both are used in the page – for the selectOneChoice to select a continent and the listView to show individual country data. The page also has a JavaScript library - AppLibraryDataboundJson.js – and uses a managed bean: jsonProviderBean based on Java Class JsonProvider.

Open the JSF page. Locate element jsonPayloadContainer. This element is invisble; however, it is rendered as client component. It specifies which data bound collections should be made available through JSON as JavaScript client side in memory data collections. Client attribute jsonPayloadAttributes contains a list of all clientAttributes that specify such collections – in this case only countryData. ClientAttribute countryData specifies #{bindings.CountriesInContinentDataType} as the IteratorBinding for which data should be pushed to the client and exposed in the object called countryData.

The EL expression used for the clientAttribute is a special one:
#{jsonProviderBean[bindings.CountriesInContinentDataType]}

. It injects the iterator binding into the get() method on the managed bean. This generic method will retrieve data from the IteratorBinding, serialize into a JSON string and return that string. The value of the clientAttribute countryData by the time the page arrives in the browser is set to this JSON string.

Leveraging Data Bound JSON in Client

The page has a load listener that invokes init() in the library. This function loads the Google Charts library – directly from a URL, not from a local file – and also initializes the JSON payloads. That means: function initializeJsonPayloads() reads the value of client attribute jsonPayloadAttributes for all elements with the CSS class jsonPayload. This value contains an array of one or more names of other client attribtutes that contain JSON strings. The function handeJsonPayloadContainer() loops over this array and parses the JSON string in each client attribute. The resulting object is stored under the name of the client attribute in the global variable jsonData from where it is available for use in JavaScript – for example for third party components.

The JSF page contains a panelHeader with id = ph1. This panelHeader is the designated container for the bar chart that we want to show for the top 8 countries by population in the selected continent.

Function drawMultSeries() is responsible for preparing the Google Charts bar chart. It uses the data from the countriesInContinent ADF data binding, available in the jsonData['countryData'] object to populate the data for the chart. Note how using the JavaScript stream equivalent the array of countries is sorted (by population descending) and sliced (back to top 8):

countryData.sort(function(a,b){ return parseInt(b.population) - parseInt(a.population); }).splice(8);

When a different continent is selected, an autoSubmit is performed and a PPR cycle takes place. Because the element jsonPayloadContainer has its binding attribute set up to #{jsonProviderBackingBean.jsonContainer} (and this bean is in request scope) the setJsonContainer method is invoked in every request – including PPR requests.

This allows us to refresh the JSON payload for the data bindings if they have been refreshed in the PPR cycle. Additionally, from method setJsonContainer() in Java Class JsonProviderBean, we can add a JavaScript snippet to the response to have the JSON payloads reinitialized:

```
        boolean isPPR = fctx.getPartialViewContext().isAjaxRequest();

        HttpServletRequest req =

            (HttpServletRequest) fctx.getExternalContext().getRequest();

        if (isPPR) {

            ExtendedRenderKitService erks = Service.getRenderKitService(fctx,
ExtendedRenderKitService.class);

            String myJavaScriptCode = "initializeJsonPayloads();";

            // this javaScript snippet is executed for each PPR request!

            erks.addScript(fctx, myJavaScriptCode);
            // add as partial target the UI Components that carry the client attributes with the
JSON payloads

            AdfFacesContext.getCurrentInstance().addPartialTarget(jsonContainer);

        }
```

The JSON container is added as partial target to have the regenerated JSON string delivered into the client side component, where function initializeJsonPayloads() can pick it up.

What's Next?

Several interesting challenges could be picked up at this stage:

- add second JSON bound data collection – for example for the continents collection – and create a client side component for continents
- add a selection components that allows the user to sort countries ascending or descending and by population, area, coastline, life expectancy or birthrate
- select other 3rd party components to power with the JSON data now available in the client
- use Canvas in the browser to client side visualize the data

Target Attribute

The RichAppWithJavaScript contains an example of using the *target* element –as described in this A-Team blog article: http://www.ateam-oracle.com/a-hidden-gem-of-adf-faces-12c-the-aftarget-tag/ . Using the target tag, we can specify exactly which components should be processed – their values – and rerendered during a partial page refresh cycle. This allows us to work around a number of common challenges, for example with PPR and required fields.

Check out the *target* page in the application, read the blog article and inspect the sources – especially target-tag.jsf.

Resources and References

Some useful resources, articles, sites etc.

Live Demo of ADF Faces: http://jdevadf.oracle.com/adf-richclient-demo

Documentation on ADF JavaScript API: https://docs.oracle.com/middleware/1221/adf/api-reference-javascript-faces/toc.htm

Oracle White Paper on JS in ADF 11g - http://www.oracle.com/technetwork/developer-tools/jdev/1-2011-javascript-302460.pdf (2011)

Navigating through a table using arrow keys: https://technology.amis.nl/2016/03/15/navigating-adf-editable-table-arrow-keys/

A Hidden Gem of ADF Faces 12c: The <af:target> Tag - http://www.ateam-oracle.com/a-hidden-gem-of-adf-faces-12c-the-aftarget-tag/

Introducing the Client Side Event Bus: C Lient Side event bus: https://technology.amis.nl/2017/01/11/client-side-event-bus-in-rich-adf-web-applications-for-easier-faster-decoupled-interaction-across-regions/

Trick for getting the real client id of a component: https://blogs.oracle.com/groundside/pattern-for-obtaining-adf-component-id-for-javascript

Prevent navigation from field upon failed validation: https://tompeez.wordpress.com/2017/01/09/pravent-nav-on-failed-validation/ (..never leave)

Implementing a client side validator: http://rohanwalia.blogspot.nl/2013/10/client-side-validation-in-adf-faces.html and more formally: https://myfaces.apache.org/trinidad/devguide/clientValidation.html

Set properties on UI components in JavaScript: http://www.awasthiashish.com/2015/08/set-adf-faces-component-properties.html#sthash.f3UYpYI3.dpbs