An Oracle White Paper
July 2011

ADF Design Fundamentals

Using JavaScript in ADF Faces Rich Client Applications

Introduction

ADF Faces is an Ajax-enabled rich JavaServer Faces component framework that uses JavaScript to render client-side components, implement rich component functionality, validate user input and convert user data input. ADF Faces exposes a complete client-side architecture that wraps the browser specific Document Object Model (DOM) API and renders ADF Faces components as client-side JavaScript objects for internal framework use, for use by ADF Faces component developers and for use by ADF Faces application developers.

Observations on help forums, mailing lists, and during code reviews show that application developers use JavaScript in ADF Faces applications primarily for three reasons:

- they have a use case requiring it
- they don’t know any better yet
- because they can

The ADF Faces architecture is designed such that application developers don’t need to write JavaScript code themselves for most of the use cases they build. JavaScript in ADF Faces applications therefore should be used as an exception rather than the rule.

If JavaScript is used, developers are encouraged to only use public ADF Faces client framework APIs instead of direct browser DOM manipulation and to stay away from using ADF Faces JavaScript objects stored in internal package hierarchies.

This whitepaper is a guideline for using JavaScript in ADF Faces. It shows how to add scripting to a page, how to listen for component events and how to work with the ADF Faces client architecture. The paper introduces the basics of using JavaScript in ADF Faces, followed by an exploration of practical use cases and their implementation. The design fundametals this paper teaches are summarized in best practices, that describe do's and don’ts when working with JavaScript in ADF and ADF Faces.

As a pre-requisite, readers are expected to be familiar with JavaServer Faces, ADF Faces and especially with JavaScript programming.

Best Practice1: An oven alone doesn't make a cook. Developers should ensure they understand the ADF Faces client architecture before using JavaScript in ADF Faces applications.

Enhancing JavaServer Faces with ADF Faces

By design, JavaServer Faces is a server-centric framework that executes server-side logic in response to form submits. In JSF, a client request is handled by up to six lifecycle phases that:

- Prepare the server-side memory tree
- Apply request parameter values
- Perform validation
- Update the business model
- Invoke component listeners and actions
- Prepare the response for render

While the JSF programming model proved to be productive to use, scalable and mature, the user experience in term of richness of the UI and client-side interactivity was poor. JavaServer Faces component providers like Oracle thus looked for ways to improve the JavaServer Faces model with client-side functionality, which in the case of ADF Faces had been solved by introducing partial component refresh and submit. In ADF Faces rich client, a part of Oracle JDeveloper 11*g*, the Oracle UI component set became an Ajax-enabled view layer framework that, alongside UI components, also provided the real client functionality missing in standard JSF.

- Ajax-based partial refresh – UI components use partial submit and refresh action to update server-side state, as well as dependent component state, without refreshing the page as a whole
- Drag and drop – ADF Faces provides a declarative model for developers to implement drag and drop functionality without coding JavaScript. Instead it's the framework that handles the drag and drop action, invoking server-side events for the developer to implement the desired business logic.

- Client-side validation – In addition to the server-side request lifecycle, ADF Faces implements a client-side request lifecycle that mirrors the server-side behavior. It allows data validation and conversion to execute on the client, in which case users get immediate feedback about possible errors caused by data input they provided. Not only is this convenient for application users, it also helps prevent unnecessary server round trips.
- Rich UI functionality – Many components like tables and panel splitter can be resized on the client by the user with mouse. Developers don't need to code for this functionality, as it is part of the client-side component rendering.
- Lightweight dialogs – Instead of opening dialogs in a separate browser window, lightweight dialogs use DHTML to open them inline to the page. This approach provides better performance and better usability and it also sidesteps the hassle of popup blockers interfering with the application functionality.

To accomplish this, ADF Faces extends the JavaServer Faces architecture, adding a complete client-side component framework and JavaScript architecture, while still adhering to the JSF standards. Most of the client-side functions used internally by the ADF Faces components are public and can be used by ADF Faces application developers alike to implement client-side application use cases, which is what the remainder of this paper is all about.

ADF Faces Client Architecture

ADF Faces uses a hybrid approach in rendering web user interfaces. By design, the majority of UI components are rendered in HTML that is generated on the server-side for a request. Only components with functionality, like a table whose columns users can change the display order of using drag and drop, have a JavaScript object representation.

Figure 1 shows how the ADF Faces client JavaScript object hierarchy compares to the server-side JavaServer Faces component hierarchy at runtime.

Every ADF Faces component, whether layout or UI component, is represented by two types of JavaScript classes: a public component object and an internal peer object.

The public object is used by the interface developer when programming on the ADF Faces client-side or when building custom ADF Faces components. Peer objects are like renderer classes and dispatch between the browser-agnostic public component interface and the browser-specific DOM implementation. They shield the ADF Faces client programming from browser differences, ensuring that ADF Faces exposes a consistent API for cross-browser client development.

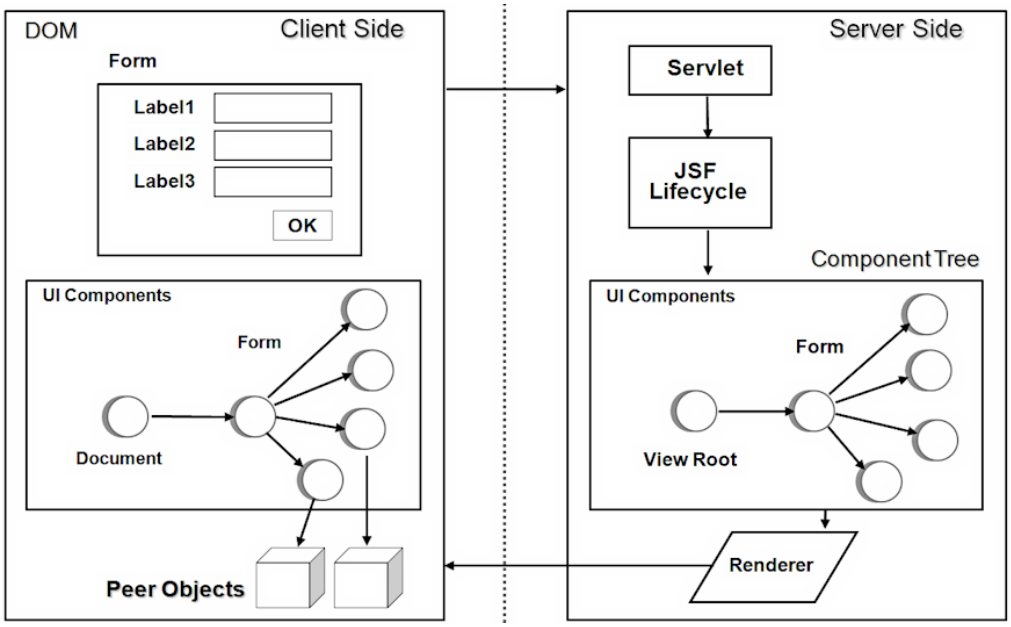


Figure 1: ADF Faces server-side and client-side architecture

ADF Faces JavaScript components

In JavaScript, there are no classes and class instances. Instead there are objects and prototypes to implement an inheritance model. The ADF Faces inheritance model starts with the `AdfObject` root object, which is extended in all of the UI components. The `AdfObject` JavaScript object is the ADF Faces client equivalent to the `Object` class in Java.

As in Java, all ADF Faces UI component JavaScript objects have a constructor that is called when the component is initialized. The constructor calls an internal initialize method, which ensures that a component is set up before it is used on a page. UI Components that are created as an extension of a parent component use the `AdfObject.createSubclass` method to define this relationship.

Except for utility objects, all objects at the minimum extend from `AdfObject`. Client-side ADF Faces UI component objects adhere to the naming convention used for their server-side peers, employing a prefix of "Adf" though. For example, the ADF Faces input text component, which is an instance of `RichInputText` on the server, is represented by the `AdfRichInputText` object on the client. Similarly, a command button, an instance of `RichCommandButton` on the server, is represented by the `AdfRichCommandButton` object on the client.

As a developer working with JavaScript in ADF Faces you don't need to bother about JavaScript object instantiation because it is automatically handled by the ADF Faces framework.

Public methods that are exposed on a component, or on its parent object, can be used to read and write component property states and to perform operations on a component, such as launching a popup dialog or performing a relative search.

Figure 2 shows the JavaScript reference doc for the ADF Faces `AdfRichInputText` client component. The exposed properties and component hierarchy looks identical to the server side equivalent. The complete ADF Faces JavaScript reference is available at:

http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12046/toc.htm

Figure 2: ADFRichInputText JavaScript component documentation

ADF Faces JavaScript object convention

When using JavaScript in ADF Faces to build applications, you work within the context of the ADF Faces client framework, for which a few rules apply:

ADF Faces JavaScript package structure

JavaScript does not adhere to package structuring as a means to build and enforce unique object names. If there are two JavaScript objects with the same name, then a conflict cannot be avoided as could be done in Java where specifying the package and class name ensures instantiation of the right object. In JavaScript, hierarchical object structures help to organize and load external object classes. In ADF Faces, two package structures are used: *oracle.adf.view.js* and *oracle.adfinternal.view.js*. The *oracle.adfinternal.view.js* package contains those JavaScript objects that should only be used internally by the ADF Faces component framework. Changes to internal objects are applied without further notice and with no guarantee for backward compatibility, which is why Oracle does not want application developers to use these objects in their custom JavaScript functions. Instead developers should use only those objects that are stored in the public *oracle.adf.view.js* package.

Note: The JavaScript reference documentation that Oracle provides for the ADF Faces client components clearly flags internal objects with a warning. The documentation is provided as part of the Oracle Fusion Middleware documentation on the Oracle Technology Network (OTN). Figure 2 shows an example of this.

ADF Faces JavaScript object scopes

JavaScript doesn't know about abstract classes, or private, public and protected scopes. In ADF Faces however all of these are implemented through naming guidelines and coding patterns that application developers should not disregard.

Abstract JavaScript classes in ADF Faces are implemented by some of its functions returning `AdfAssert.failedInAbstractFunction` when invoked. Abstract objects are indirectly used through their subclasses.

Object scopes in ADF Faces, not to be confused with JavaServer Faces memory scopes, are implemented by the following naming conventions:

SCOPE	NAMING CONVENTION
public	Public function names follow a camel case pattern, starting with a lowercase letter, as in findComponent, or findComponentByAbsoluteId.
private	Private functions have a leading underscore in their name. Those functions are not shown in the ADF Faces public JavaScript documentation, but in the ADF Faces source code and DOM inspection tools like firebug. Application developers should not use methods in a private scope.
protected	Protected functions are for internal use. They start with an uppercase letter, followed by a camel case notation, in the naming.
package	Package functions are private to the package scope, the file structure the JavaScript object is in. These functions names start with a double underscore and also should not be used by application developers.

Only public objects and functions are good to use in ADF Faces application development. Functions that are not public are for internal framework use or use by ADF Faces component developers. The JavaScript reference documentation shown in Figure 2 also lists protected and abstract functions for component and event objects, and developer should be aware of this.

Best Practice 2: Protected, package, or private methods and variables that DOM inspection tools like Firebug show for ADF Faces components and events should not be used in custom ADF Faces application development.

JavaScript in ADF Faces 101

ADF Faces exposes public JavaScript APIs that application developers use to implement client-side development use cases. Though using JavaScript is a valid approach in ADF Faces, it should be used with care and only if there is no native solution to a problem in ADF Faces that provides the same functionality. One good reason for using JavaScript sensibly in ADF Faces applications are future version upgrades. For the Oracle JDeveloper IDE, it is easier to automatically detect and update well formatted XML metadata tags than to adapt JavaScript code to possible API changes.

Before looking closely at code samples, it helps to understand the basics of how to use JavaScript in ADF Faces pages and page fragments.

Adding JavaScript to on a page

To use JavaScript on an ADF Faces view, application developers either add JavaScript code to the page source or reference it in an external library file.

To ensure that the JavaScript sources are efficiently downloaded to the client, ADF Faces provides the `af:resource` component tag, which must be added as a direct descendent of the `af:document` tag.

```
<af:document>
  <af:resource type="javascript" source="/customJsCode.js"/>
  …
</af:document>
```

```
<af:document>
  <af:resource type="javascript">
    function customJsFunction(){ … }
  </af:resource>
  …
</af:document>
```

Finding ADF Faces components on the client

The hierarchy of ADF Faces components on the client is the same as the component hierarchy on the server. The difference though is that not all ADF Faces components are rendered as client objects by default, but server-side generated HTML. For JavaScript to be able to access an ADF Faces component by its client component API, the first development task, therefore, is to ensure that a client component is generated.

Creating client components

Only components with behavior, like ADF Faces tables, have a default JavaScript object representation. For all other components, a client object is created when one of the following conditions is met:

- A component has an `af:clientListener` tag added. The `af:clientListener` component allows JavaScript function to listen for ADF Faces component events, like select or disclosure events, and browser DOM events like focus, mouse hover, mouse click, or keyboard events.
- The component has its **clientComponent** property set to true.

Note that all components, even those without a client object, are visible in the browser DOM tree. Developers accessing components through DOM manipulation operate on the generated HTML output of a component, which is not the same as working with the component using the ADF Faces public JavaScript API. Any DOM access should be avoided when using ADF Faces, because there is no guarantee that the rendered component output will stay the same between ADF Faces versions.

Best Practice 3: Application developers should ensure that a component exists on the client before accessing it from JavaScript. ADF Faces client component objects are created by setting the component *clientComponent* property to true or adding an `af:clientListener` tag.

Finding components on a page

ADF Faces client components can be searched either absolutely on the ADF Faces page object or relative to another component. JavaScript functions that are called in response to a JavaScript component event (like "action", "click", or "selection") get access to the triggering component by calling `getSource()` on the event object that is passed into the function as an argument. This component handle then can be used to perform a relative search. If the component to look up on the client is not related to the event-triggering component, then developers need to perform an absolute search starting from the base page object.

Absolute search

The `AdfPage.PAGE` object is the base page management class in Oracle ADF Faces. It grants access to all components either by their component id or the component locator, which is used for components like table that stamp their children. The following three functions are exposed on the `AdfPage.PAGE` object to look up components on a page.

- **findComponent** – The `AdfPage.PAGE.findComponent` function searches a component by its client id. The component client id is the id value that gets generated into the HTML page output. It is different from the component id defined at design time. Because the client id is not stable, and may even change when a page is rerun, developers should never hard code the client id value in their scripts. Instead the client id should be read at runtime from the server-side component and dynamically added to the JavaScript, for example by using the `af:clientAttribute` tag on the component that raises the JavaScript event.
- **findComponentByAbsoluteId** – The recommended way to look up client-side ADF Faces components is through `AdfPage.PAGE.findComponentByAbsoluteId`. The absolute component Id is built from the component Id and the Ids of the naming containers that are located between the document root and the target component on the page. A table "t1" that is contained in a Panel Collection "pc1" would have an absolute Id of "pc1:t1" if there are no more naming containers involved.
- **findComponentByAbsoluteLocator** – For most components, the absolute locator used within calls to `AdfPage.PAGE.findComponentByAbsoluteLocator` is the same as the absolute component Id. Only components (like tables) that stamp their children differ in that to access the stamped child component, the locator Id contains a key that addresses the stamped component instance. The absolute locator for a table cell component "c1" may appear as "pc1:t1[stamp key].c1".

Best Practice 4: Developers should not hardcode any client component id that shows in the generated HTML output for a page into their JavaScript functions. Client id values may change even between runs of a view. Instead developers can dynamically determine the client Id by a call to the ADF Faces component's `getClientId` method in a managed bean.

Relative search

A search relative to another ADF Faces component is performed by one of the following functions:

- **findComponent** – The `findComponent` method searches a component starting from a given component instance. The component id to look up must be unique within the scope of the closest ancestor naming container. For example, to search a component "c1" located in a naming container hierarchy "nc1:nc2" from a button located in "nc1", the relative search Id to use is "nc2:c1".

  Leading colons in the search Id allow the search to move from the current naming container into the parent container. For example, starting a component search from a button "b1" in a naming container hierarchy "nc1:nc2" and prefixing the component search id with two colons ("::c1") searches for the component "c1" in the parent naming container "nc1". Each additional colon addresses one more parent container so that ":::c1" would search for the component in the parent naming container of "nc1", which also could be the document root. A leading single colon searches the component directly from the document root.

- **findComponentByAbsoluteLocator** – This function is exactly the same as the one used on the `AdfPage.PAGE` object. It requires the absolute locator Id provided in the search. Though the function starts the search relative from a component, it internally calls out to `AdfPage.PAGE.findComponentByAbsoluteLocator`.

Handling client component events

ADF Faces client components behave like their server-side pendants and raise events in response to component changes. Developers work with events in that they either listen for them or queue events on a client-side component to invoke the defined component behavior. Event listeners are defined as JavaScript functions and receive the event object as an argument.

ADF Faces client component events

ADF Faces client component events allow developers to listen and to respond to component changes like value change, disclosure, query, focus, popup, and many more. Each event, when captured, provides information about the component it is invoked on, the event itself, and each event exposes functions developers use to manipulate the event object before it is sent to the server.

In addition to ADF Faces component events, the ADF Faces client event architecture provides wrappers for native browser DOM events like focus events, mouse events, and keyboard events for developers to use.

All component event object names start with "Adf", followed by the event name, and end with "Event". A very common event to listen to is `AdfActionEvent`, which is triggered by command components like button or link. The `AdfActionEvent` object extends `AdfComponentEvent`, which itself extends `AdfPhaseEvent`, which is derived from `AdfBaseEvent`.

The `AdfComponentEvent` is the parent class of all ADF Faces component events. The function added by the `AdfComponentEvent` object is `queue(Boolean isPartial)`, which allows developers to queue an event defined on a component.

For example, developers who want to invoke an action event on a command button using JavaScript, use a convenient function exposed on the `AdfActionEvent` object. For this they

1. Look up the command button on the client-side
2. Call AdfActionEvent.queue(commandButton, true | false), where the second argument is passed as true if the event should be sent as a partial submit

The `AdfPhaseEvent` object adds lifecycle functions to the event for developers to know about the current event phase, like bubbling or capture, and whether or not an event can be canceled.

Most of the functions developers work with when handling client events in ADF Faces are defined on the `AdfBaseEvent` class. Developers, for example, may call `cancel()`, `stopBubbling()`, or `setPartial()` on an event object to change the behavior of an event.

If an event opens a dialog, for example to export table content to MS Excel, then calling `noResponseExpected()` on the client event ensures that ADF Faces doesn't wait for the call to return.

Another useful function, discussed later in an example, is `preventUserInput()` that, when called, launches a glass pane that prevents user input for the time it takes an event to finish processing.

Application developers don't need to worry about which function is exposed on which event object. Instead, they just work with the top-level event, like `AdfAcctionEvent`, to invoke these functions. In fact, functions maybe defined as abstract in the base classes to be overridden in the component event class.

Note: Mouse and keyboard events are not component events but native browser events. These events are spawned as native browser DOM events and only wrapped by the ADF Faces client-side event framework. How to listen and respond to keyboard events is explained later in this paper.

Best Practice 5: Application developers should call `cancel()` on all events that don't need to propagate to the server

Listening for client events

To listen for a component event, developers use the `af:clientListener` tag located in the Oracle JDeveloper Component Palette's "Operation" category. The `af:clientListener` tag exposes two attributes

- **Method** – The method attribute is the name of the JavaScript function to call in response to the component event. The JavaScript function needs to take a single argument for the event object that is passed in.
- **Type** – The type attribute defines the event the client listener handles.

In the example shown in Figure 3, the *Method* attribute is set to "handleValueChange" and the *Type* attribute to "valueChange". The JavaScript function to receive the event thus must be defined as:

```
<af:resource type="javascript">
  function handleValueChange(evt){ … }
</af:resource>
```

The "evt" argument receives the event object, which in the case of a value change event is *`AdfValueChangeEvent`*.

Note: JavaScript source codes can be added to the page sources, surrounded by the *`af:resource`* tag, or referenced in an external JavaScript file.

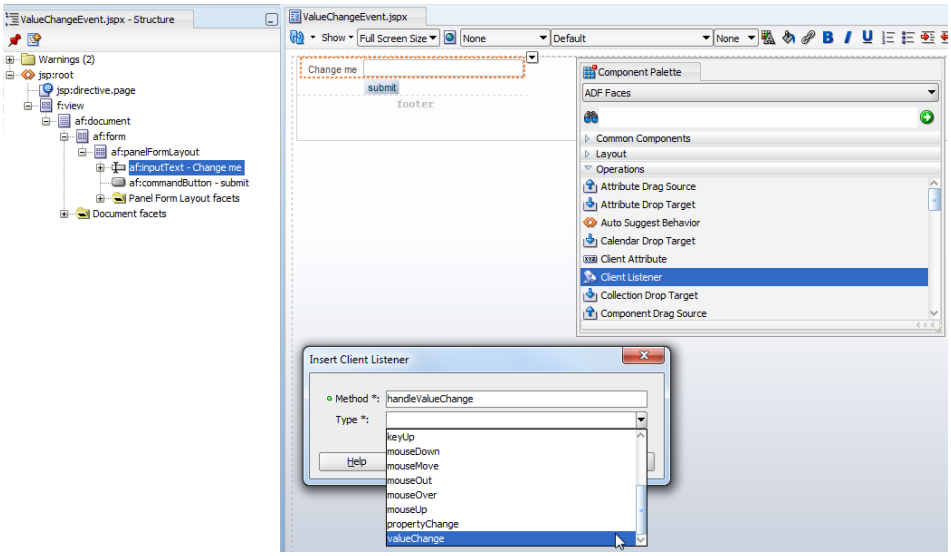


Figure 3: Adding a client listener to an Input Text component

The client listener definition added to the page sources for the use case in Figure 3 is:

```
<af:inputText label="Change me" id="it1">
   <af:clientListener method="handleValueChange"
                      type="valueChange"/>
</af:inputText>
```

How to prevent events from propagating to the server

When handling a client component event, application developers can prevent it from propagating to the server by a call to `cancel()` on the event object that is passed into the listener function. For example, canceling an action event does not execute the associated server-side action listener and action method.

```
function handleButtonPressed(evt){
 //evt is of type AdfActionEvent
 if(<some condition>){
   …
   //prevent server propagation of event
   evt.cancel();
 }
 else{
   …
 }
}
```

Not all events can be canceled on the client. To determine whether an event can be canceled on the client or not, event objects expose the `isCancelable()` function.

Passing additional information to a JavaScript function

An ADF Faces event object is passed as a single argument to JavaScript functions that are called from the *`af:clientListener`* behavior tag. Application developers use the event object to access the event source component and to handle the event. Sometimes additional information is needed for the script to execute. To access this extra information, developers can:

- Look up a UI component that contains the required information using the findComponent, the findComponentByAbsoluteId or the findComponentByAbsoluteLocator function of the ADF Faces client framework
- Add an `af:clientAttribute` to the server-side component on which the ADF client event is invoked and use Expression Language or a static string to provide the required information

- Use a JavaScript callback to allow the `af:clientListener` "method" attribute to have arguments

How to look up components in a relative or absolute search is covered earlier in this paper. The other two options follow:

Using af:clientAttribute

The `af:clientAttribute` tag is specific to ADF Faces and is used to extend the properties set exposed by an ADF Faces client component.

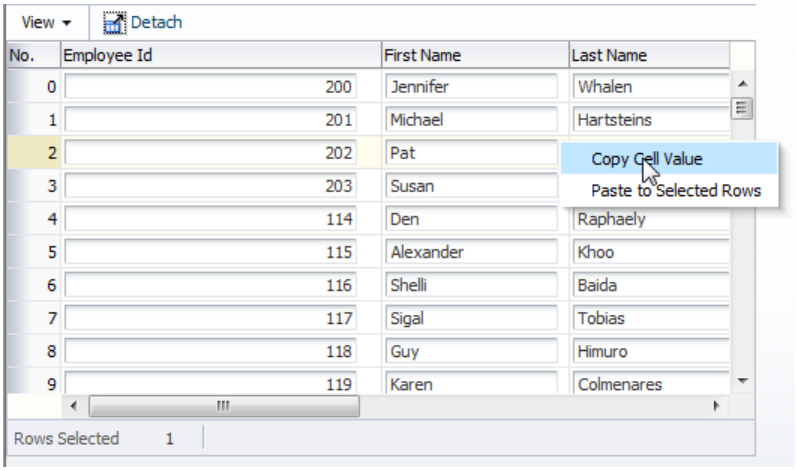


Figure 4: Using JavaScript to copy the selected table cell content ...

The sample shown in Figure 4 allows users to select a table cell on the client to copy its content to other rows using a context menu or keyboard shortcut.

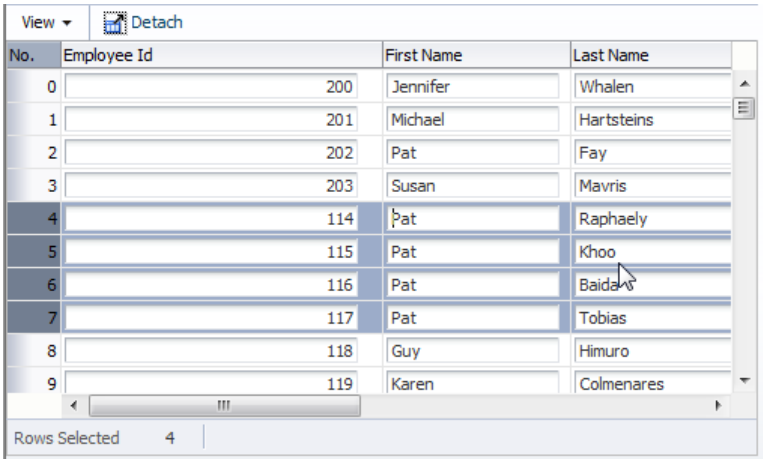


Figure 5: ... and paste it into other rows

To determine the table cell to copy the value from, additional information is needed about the table cell column and its row index. The information is provided by `af:clientAttribute` tags that read the row index from the table status variable and the column name from the ADF binding layer.

```
<af:table value="#{bindings.allEmployees.collectionModel}"
          var="row" varStatus="status" ... >
 ...
 <af:column ...>
  <af:inputText value="#{tableRow.bindings.firstName.inputValue}"
                ... >
    <af:clientListener .../>
    <af:clientAttribute name="colname"
        value="#{tableRow.bindings.firstName.name}"/>
    <af:clientAttribute name="rwKeyIndx"
        value="#{status.index}"/>
  </af:inputText>
 </af:column>
</table>
```

The JavaScript function that reads the additional information from the table cell input text component is:

```
function copyValueToSelectedRows(evt) {
  var txtField = evt.getSource();
  //get the name of the column which row cell to read
  var column = txtField.getProperty('colname');
  //get the index of the row to copy values from
  var rowKeyIndx   = txtField.getProperty('rwKeyIndx');
  …
}
```

The benefit of using `af:clientAttribute`, compared to the JavaScript callback, is that it is a declarative approach that allows the use of deferred and eagerly resolved EL expressions to access data values that should become available on the client.

Note: Only data types that exist in JavaScript can be passed down to the client. For example, an object of oracle.jbo.Key or an entity cannot be read on the client. The client attribute data types in the example above, therefore, are of type string and long.

Using JavaScript callback

The use of JavaScript callbacks is a second option to pass additional arguments into a JavaScript function referenced from an `af:clientListener` tag. The callback function wraps the component listener function, as shown here:

```
function openPopup(popupId, compId){
  return function(evt){
    …
    var popup = AdfPage.PAGE.findComponentByAbsoluteId(popupId);
    var hints = {align:"end_before", alignId:compId };
    popup.show(hints);
  }
}
```

In the example, the JavaScript callback function takes two arguments identifying the component Id of a popup and the Id of a UI component that the opened popup should be aligned with. The ADF Faces component listener function is returned by the callback and expects a single argument, which is the ADF Faces component event object.

To open the popup as a context menu of an `af:inputText` component, developers would call the JavaScript callback function as shown here:

```
<af:inputText …>
  <af:clientListener method=" openPopup('pc1:p1','txt1')"
                     type=" contextMenu"/>
</af:inputText>
```

Note: 'pc1:p1' references a popup 'p1' that is contained in a naming container 'pc1'. Though naming containers don't exist in JavaScript, at runtime they have their component Id added as a prefix to the components they contain.

The benefit of using JavaScript callbacks, compared to `af:clientAttribute` tags is that they allow to write generic script functions that don't make any assumption about the source component properties. JavaScript functions that don't depend on custom component attributes are better to reuse and store in external library files.

Invoking server-side Java from client-side JavaScript

A common requirement in Ajax is for the client to invoke server-side logic without performing page navigation. In ADF Faces, this can be implemented using the `af:serverListener` component, which allows developers to declarative register a server-side listener that should be executed when a custom client event is fired. The server-side listener is defined as a managed bean method that returns void and accepts a single argument of type `ClientEvent`. The client

event argument contains information about the origin, the component on which the event was raised, and the payload, the information that is send to the server.

For example, to mirror characters entered by a user in a text field using the keyboard, application developers would configure the `af:serverListener` like this:

```
<af:inputText id="it1" label="…">
  <af:clientListener method="handleKeyUp" type="keyUp"/>
  <af:serverListener type="MyCustomServerEvent"
                     method="#{mybean.handleServerEvent}"/>
</af:inputText>
```

The ServerListener component takes two attributes, "type" and "method". The "type" attribute is the name that developers use to reference the server listener component in a JavaScript call. The "method" attribute expects an EL expression referencing a managed bean method on the server. For example, the "handleServerEvent" method used in the preceding code snippet would be defined as:

```
public void handleServerEvent(ClientEvent ce){…}
```

Note: The component owning the `af:serverListener` component must have a client-side representation, which means that it must either have an `af:clientListener` child tag added or its "clientComponent" property set to "true".

To invoke a server listener from JavaScript, application developers use the `AdfCustomEvent` JavaScript object's queue method. The method signature is:

```
public AdfCustomEvent(AdfUIComponent source,
                      String type,
                      Object params,
                      Boolean immediate)
```

- The "source" argument is the UI component that originates the custom server event. In this example it is the component with the id "it1".
- The "type" argument takes the name of the ServerListener component ("MyCustomServerEvent" in this example).
- The "params" argument is a JSON-encoded message array consisting of name:value pairs delimited by commas, e.g. {arg1:value1,arg2:value2, …}
- The "immediate" flag determines whether the server call is handled during the JSF APPLY_REQUEST phase or during the InvokeApplication phase. The immediate flag is used if, for example, the event results in a page navigation, to suppress client-side field validation, or when cancelling a form edit. If the server event updates the ADF model,

for example to create a new form or table row, the immediate attribute needs to be set to **false**.

To continue with the above example, the JavaScript function that is called by the *af:clientListener* is shown here:

```
<af:resource type="javascript">
   function handleKeyUp (evt) {
     var inputTextComponen = event.getSource();
     AdfCustomEvent.queue(inputTextComponent,
                          " MyCustomServerEvent ",
                          {fvalue:component.getSubmittedValue()}
                         ,false);
     event.cancel();
  }
</af:resource>
```

This example sends the value of the input text field to the server for each key a users presses. So when "hello" is typed into the input field, the server receives the following notifications: "h", "he", "hel", "hell", "hello". To print the message on the server, the listener method in the needs to be completed as shown here:

```
public void handleServerEvent(ClientEvent ce){
 String message = (String) ce.getParameters().get("fvalue");
 // print the message to the console
 System.out.println(message);
}
```

As mentioned, the number of arguments sent to the server is up to the client developer and the use case the `af:serverListener` component is used for. The name of the arguments can be freely chosen.

Best Practice 6: Client-to-server calls should be use sensibly to reduce the amount of network roundtrips and thus to prevent slow performance due to network latency.

Calling JavaScript from Java

The reverse use case to invoking server-side logic from JavaScript is to invoke client-side JavaScript from a managed bean. To execute JavaScript from a managed bean, ADF Faces developers use the `ExtendedRenderKitService` class, which is part of the Apache MyFaces Trinidad component set that builds the foundation of ADF Faces.

The call to execute JavaScript on an ADF Faces application is independent of the script itself, so that it can be wrapped in a custom helper method as shown below.

```
import
  org.apache.myfaces.trinidad.render.ExtendedRenderKitService;
import org.apache.myfaces.trinidad.util.Service;
```

```
…
private void writeJavaScriptToClient(String script) {
  FacesContext fctx = FacesContext.getCurrentInstance();
  ExtendedRenderKitService erks =
     Service.getRenderKitService(fctx,
                                 ExtendedRenderKitService.class);
  erks.addScript(fctx, script);
}
```

A popular use case for calling JavaScript from a managed bean is to open a client-side ADF Faces popup dialog, as shown here:

```
StringBuilder script = new StringBuilder();
script.append(
  "var popup = AdfPage.PAGE.findComponentByAbsoluteId('p1');");
script.append("if(popup != null){");
script.append("popup.show();");
script.append("}");
writeJavaScriptToClient(script.toString());
```

Note. ADF Faces provides a behavior tag, `af:showPopupBehavior`, to open client-side popup dialogs. The behavior tag should be used instead of a custom JavaScript solution whenever possible. Also note that starting in Oracle JDeveloper 11*g* R1 PatchSet 2 (release 11.1.1.3), the RichPopup component class exposes a new method "show(PopupHints)" to launch popup dialogs from Java, providing an additional choice.

JavaScripts executed from a managed bean can be quite complex. In this example an LOV popup dialog is launched if the currently selected client-side ADF Faces component is of type `af:inputListOfValues`. For this, the JavaScript code contains references to the ADF Faces client component API:

```
StringBuilder script = new StringBuilder();
//ensure the component in focus is an LOV
script.append("var activeComponent =
                      AdfPage.PAGE.getActiveComponent();");
script.append(
       "if(activeComponent != null && " +
       "activeComponent instanceof AdfRichInputListOfValues &&" +
       "activeComponent.getReadOnly()==false){");
script.append(
       "var component = AdfPage.PAGE.getActiveComponent();");
script.append(
   "if(component.getTypeName() == 'AdfRichInputListOfValues'){");
        //queue the LOV event
```

```
script.append("AdfLaunchPopupEvent.queue(component,true);");
script.append("}");
script.append("}");
writeJavaScriptToClient(script.toString());
```

Note: When executing JavaScript from Java, developers should keep in mind the latency that exists on the Internet or LAN connection between the server and the client, as well as the latency in executing JavaScript added by the browsers. JavaScript calls from a managed bean are not queued, but are immediately sent to the client. Latency may become a problem if multiple calls are too quickly issued in a sequence, in which case a later call may override its predecessor.

Note: In Oracle JDeveloper 11*g* releases before version 11.1.1.2, popup dialogs could only be closed from JavaScript. Starting with Oracle JDeveloper 11*g* R1 PatchSet 2, a new API, `hide()`, became available on the `RichPopup` component class to close a popup from Java. Internally both `RichPopup` component methods, hide() and show(PopupHints), use JavaScript issued through the `ExtendedRenderKitService` object.

Logging

Logging is a troubleshooting mechanism that provides a better understanding of application runtime errors and behavior. Software design guidelines call for developers to write granular log traces about errors, application flows and application state information.

The ADF Faces client component architecture provides a client-side logging functionality that is similar to the logging in Java in that it allows developers to write log messages for specific log levels and view framework messages, both of which are sent to the configured log output, which by default is the browser console. In ADF Faces applications, logging is enabled by a context parameter setting in the web.xml configuration file:

```
<context-param>
  <param-name>oracle.adf.view.rich.LOGGER_LEVEL</param-name>
  <param-value>ALL</param-value>
</context-param>
```

Note: An example provided later in this section shows how client-side logging can be dynamically turned on and off using JavaScript on a page

Allowed context parameter log-level values are:

- SEVERE
- WARNING

- INFO
- CONFIG
- FINE
- FINER
- FINEST
- ALL

To write log messages, developers use the `AdfLogger` client object class as shown below

```
AdfLogger.LOGGER.logMessage(AdfLogger.<log level>,
                            'your message here');
```

The default log instance, which is used when writing log messages with the "LOGGER" static reference, writes messages to the browser console window. Figure 6 shows how log messages display in the in the Firebug console for the `AdfLogger.INFO` log level.

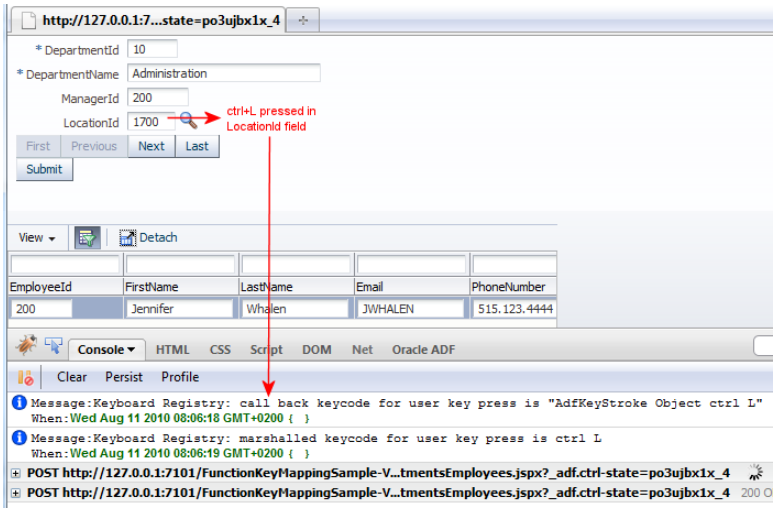


Figure 6: Keyboard registration on input text field and log information shown in Firebug console

The log information written to the console can also be defined locally by setting the log level from JavaScript on a page, or by defining a filter that looks at the log record to determine whether or not to print the information.

For example, configuring the log output in the web.xml file configuration to NONE and calling `AdfLogger.LOGGER.setLevel(AdfLogger.INFO)` in JavaScript on a page, prints informational logs to the client console.

Figure 7 shows a select list component that can be used to dynamically change the log level on the client.

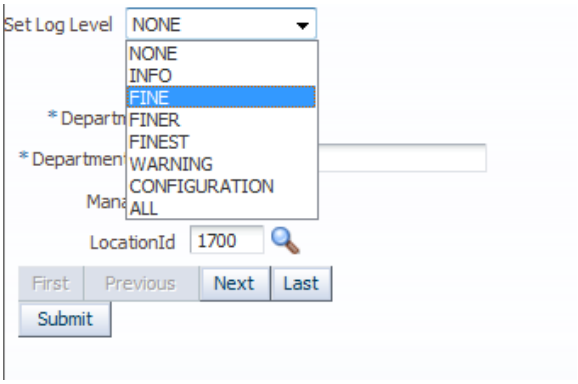


Figure 7: Select choice to dynamically set the logging level

The page source for the select component is shown below:

```
<af:selectOneChoice label="Set Log Level" id="soc1"
    value="#{browseDepartmentsBacking.loggerDefault}">
  <af:selectItem label="NONE"
                 value="AdfLogger.OFF" id="s6"/>
  <af:selectItem label="INFO"
                 value="AdfLogger.INFO" id="s8"/>
  <af:selectItem label="FINE"
                 value="AdfLogger.FINE" id="s2"/>
  <af:selectItem label="FINER"
                 value="AdfLogger.FINER" id="s3"/>
  <af:selectItem label="FINEST"
                 value="AdfLogger.FINEST" id="s7"/>
  <af:selectItem label="WARNING"
                 value="AdfLogger.WARNING" id="s1"/>
  <af:selectItem label="CONFIG"
                 value="AdfLogger.CONFIG" id="s5"/>
  <af:selectItem label="ALL"
                 value="AdfLogger.ALL" id="s4"/>
    <af:clientListener method="setLogLevel" type="valueChange"/>
</af:selectOneChoice>
```

The JavaScript function that is referenced from the `af:clientListener` component is shown below. It reads the submitted value from the select choice component to set it as the new client-side log level.

```
function setLogLevel(evt){
  var selectOneChoice = evt.getSource();
  var logLevel = selectOneChoice.getSubmittedValue();
  var logLevelObject = AdfLogger.NONE;

  //the select one choice returns a string. However, we need
  //a number without hard coding it up front. Using a switch
  //statement to handle the string to object translation

  switch(logLevel)
  {
    case 'AdfLogger.INFO':
      logLevelObject = AdfLogger.INFO;
      break;
    case 'AdfLogger.FINE':
      logLevelObject = AdfLogger.FINE;
      break;
    case 'AdfLogger.FINER':
      logLevelObject = AdfLogger.FINER;
      break;
    case 'AdfLogger.FINEST':
      logLevelObject = AdfLogger.FINEST;
      break;
   case 'AdfLogger.WARNING':
      logLevelObject = AdfLogger.WARNING;
      break;
   case 'AdfLogger.CONFIGURATION':
     logLevelObject = AdfLogger.CONFIGURATION;
     break;
   case 'AdfLogger.ALL':
     logLevelObject = AdfLogger.ALL;
     break;
   default:
     logLevelObject = AdfLogger.NONE;
  }
  _logger.setLevel(logLevelObject);

  //optional: cancel event if the server-side managed bean
  //that holds the default value for the af:selectOneChoice
  //component does not need to be updated
  evt.cancel();
```

```
}
```

Note the use of the switch statement to turn the log-level string statement into an `AdfLogger` object. Internally, the log levels are represented by numbers. However, to avoid having to hardcode the log-level numbers, which would make the JavaScript code dependent on the current implementation of the `AdfLogger` class, the log level names have been translated into their respective object representation.

Best Practice 7: No knowledge about framework internal implementation details should be used directly in JavaScript code. Instead, public APIs and constants should always be used.

The managed bean that is referenced from the `af:selectOneChoice` component to provide the default selection is defined in request scope and has the following content

```
public class BrowseDepartmentsEmployeesBacking {
    String loggerDefault = "AdfLogger.OFF";

    public BrowseDepartmentsEmployeesBacking() {
        super();
    }
    public void setLoggerDefault(String loggerDefault) {
        this.loggerDefault = loggerDefault;
    }

    public String getLoggerDefault() {
      return loggerDefault;
    }

}
```

Note: Using client-side logging often proves to be more helpful than using JavaScript alerts or Firebug because the focus is not taken out of the application.

Note: The output device of log information is determined by the `AdfLogWriter` implementation set for the `AdfLogger` instance. Sending log messages back to the server for inclusion in the server-side log files is not a good idea if performance is key.

Note: Oracle JDeveloper 11*g* R1 PS3 (11.1.1.4) is required to view log messages in the Microsoft Internet Explorer 8 native JavaScript console (open with F12).

Assertions

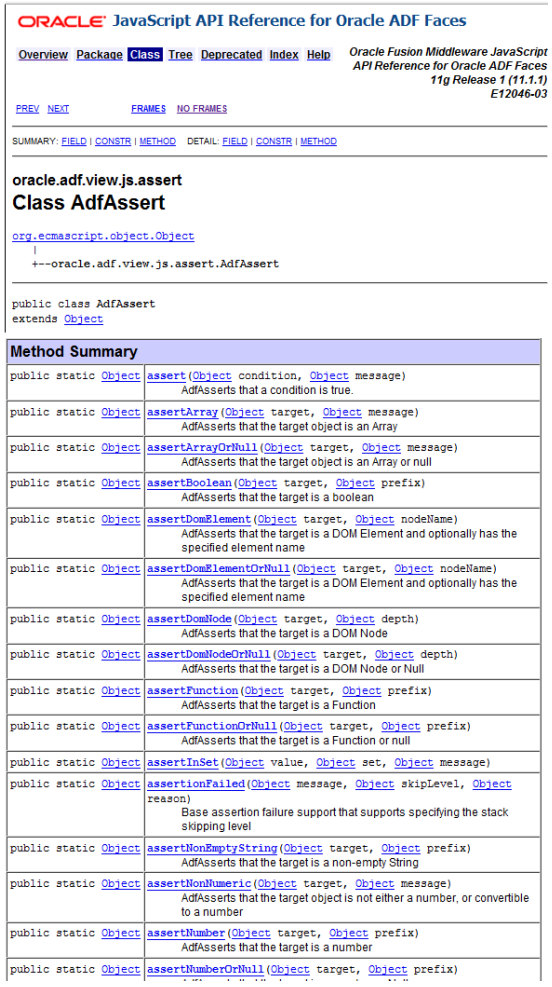
Unlike Java, JavaScript is not a typed language. For JavaScript functions, this means that objects passed as arguments are not required to have a specific data type, which could lead to failure at JavaScript runtime.

To detect cases in which wrong argument types are passed to a function, ADF Faces provides JavaScript assert functions that are used by the framework itself, but that can also be used in custom JavaScript programs. Figure 8 shows the available online JavaScript documentation for the `AdfAssert` object.

If an assertion fails, an error of type `org.ecmascript.object.error.Error` is thrown to display in the browser. The assertion failure always displays as a browser alert for developers to handle the assert problems immediately. At application runtime, assertions should be disabled, which also is the default setting.

To enable assertions in ADF Faces rich client, open the web project's web.xml file and add the following context parameter:

```
<context-param>
  <param-name>
    oracle.adf.view.rich.ASSERT_ENABLED
  </param-name>
  <param-value>true</param-value>
</context-param>
```



ORACLE JavaScript API Reference for Oracle ADF Faces

Overview Package Class Tree Deprecated Index Help

Oracle Fusion Middleware JavaScript API Reference for Oracle ADF Faces 11g Release 1 (11.1.1) E12046-03

PREV NEXT FRAMES NO FRAMES

SUMMARY: FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

oracle.adf.view.js.assert

Class AdfAssert

org.ecmascript.object.Object
|
+--oracle.adf.view.js.assert.AdfAssert

public class AdfAssert
extends Object

Method Summary

public static Object	assert(Object condition, Object message) AdfAsserts that a condition is true.
public static Object	assertArray(Object target, Object message) AdfAsserts that the target object is an Array
public static Object	assertArrayOrNull(Object target, Object message) AdfAsserts that the target object is an Array or null
public static Object	assertBoolean(Object target, Object prefix) AdfAsserts that the target is a boolean
public static Object	assertDomElement(Object target, Object nodeName) AdfAsserts that the target is a DOM Element and optionally has the specified element name
public static Object	assertDomElementOrNull(Object target, Object nodeName) AdfAsserts that the target is a DOM Element and optionally has the specified element name
public static Object	assertDomNode(Object target, Object depth) AdfAsserts that the target is a DOM Node
public static Object	assertDomNodeOrNull(Object target, Object depth) AdfAsserts that the target is a DOM Node or Null
public static Object	assertFunction(Object target, Object prefix) AdfAsserts that the target is a Function
public static Object	assertFunctionOrNull(Object target, Object prefix) AdfAsserts that the target is a Function or null
public static Object	assertInSet(Object value, Object set, Object message)
public static Object	assertionFailed(Object message, Object skipLevel, Object reason) Base assertion failure support that supports specifying the stack skipping level
public static Object	assertNonEmptyString(Object target, Object prefix) AdfAsserts that the target is a non-empty String
public static Object	assertNonNumeric(Object target, Object message) AdfAsserts that the target object is not either a number, or convertible to a number
public static Object	assertNumber(Object target, Object prefix) AdfAsserts that the target is a number
public static Object	assertNumberOrNull(Object target, Object prefix)

Figure 8: Assertions in ADF Faces

The main objective of assertions in ADF Faces rich client is to ensure that the framework is used correctly in framework internal and custom scripting. Assert functions also make sense to use in custom JavaScript functions that are referenced on a page using external JavaScript library files to ensure input arguments are correctly passed.

```
function doSomething(numArg){
  AdfAssert.assertNumeric
    (numArg, "Numeric failure for argument \"numArg\" value
             \" "+numArg+"\"");
  … function code to follow here …
}
```

If an assertion fails, a JavaScript alert like that shown in Figure 9 is displayed for the developer to trace the problem back.

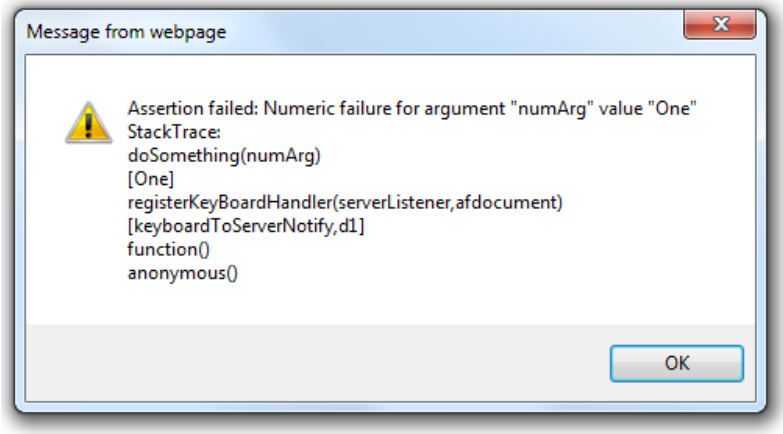


Figure 9: Assertion failure displayed for a JavaScript function

The `AdfAssert` object throws the `org.ecmascript.object.error.Error` after displaying the error message, which also stops the ADF Faces application from further processing.

To continue beyond the first error displayed for an application, developers can surround the assertion check with a try-catch block:

```
function doSomething(numArg){
 try{
   AdfAssert.assertNumeric
    (numArg, "Numeric failure for argument \"numArg\" value\"
             "+numArg+"\"");
   … function code to follow here …
  }
  catch(ex){
    //handle exception here.
  }
}
```

Note: For better performance, JavaScript assertions should be disabled in production environments. This also is the default setting.

Best Practice 8: JavaScript exceptions should be handled gracefully and not just suppressed.

ADF Faces JavaScript Examples

In the following, use cases and their implementation are presented for using JavaScript in ADF Faces. Whether JavaScript is used in an ADF Faces application for implementing a use case is in the judgment of the application developer alone. However, JavaScript should be used sensibly and not only because it is possible.

How to prevent user input during long-running queries

Application users are often impatient when waiting for a task to be completed. To prevent users from pressing a command button twice or changing data input while waiting for a long-running query or process to complete, ADF Faces provides the option to suppress any user input for the duration of an action.

The following example not only shows how to prevent user interaction during the waiting time but also and possibly more importantly, shows how to display a splash screen for the user to know that the application is sound but busy. Figure 10 shows an example of a splash screen that can be launched by the application developer in response to the application busy state change.

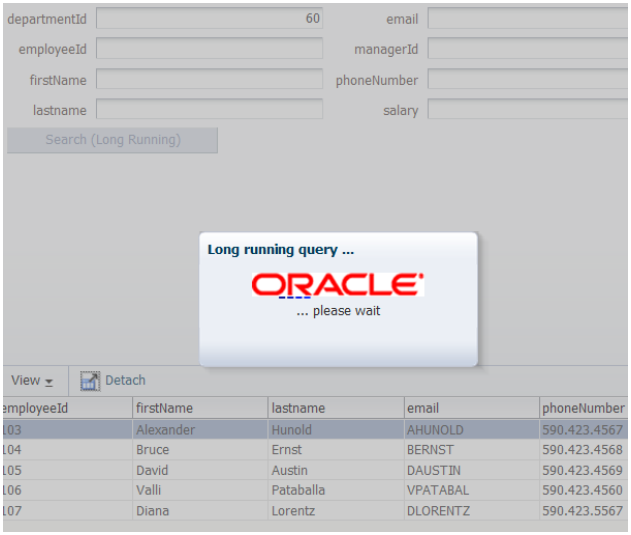


Figure 10: Show splash screen for long running queries

Creating the Splash Screen

The splash screen in this example is built with an `af:popup` component that contains an `af:dialog` component. An important configuration on the `af:popup` is to set its contentDelivery property to "immediate" for the component to render when the page loads.

```
<af:popup id="p1" contentDelivery="immediate">
  <af:dialog id="d2" type="none" title="Long running query ..."
```

```
            closeIconVisible="false">
    <af:panelGroupLayout id="pgl1" layout="vertical">
      <af:image source="/images/codecorner.gif" id="i1"/>
        <af:image source="/images/animbar.gif" id="i2"
                  inlineStyle="width:197px;"/>
        <af:outputText value="... please wait" id="ot11"/>
    </af:panelGroupLayout>
  </af:dialog>
</af:popup>
```

Invoking the splash screen

Using JavaScript to launch the splash screen and to disable user input, no change is required to the command component that invokes the long-running query or process. This is important as it allows developers to almost declaratively add this functionality to their existing applications:

```
<af:commandButton
   actionListener="#{bindings.findEmployee.execute}"
   text="Search (Long Running)" id="cb1" partialSubmit="true">
     <af:clientListener method="enforcePreventUserInput"
                        type="action"/>
</af:commandButton>
```

In this page source example, an `af:commandButton` component has an action listener configured bound to an ADF method binding. The "partialSubmit" attribute is also set to true to issue a partial submit, which is a prerequisite for the `af:popup` component to launch.

An `af:clientListener` component is configured for the button to listen for the button action event, which can be triggered from a key press or mouse click. The listener calls a JavaScript function, "enforcePreventUserInput", to disable user interaction and to show the splash screen.

```
<af:resource type="javascript">
  function enforcePreventUserInput(evt){
    var popup = AdfPage.PAGE.findComponentByAbsoluteId('p1');
    if (popup != null){
      AdfPage.PAGE.addBusyStateListener(popup,handleBusyState);
      evt.preventUserInput();
    }
  }

  //JavaScript callback handler
  function handleBusyState(evt){
```

```
    var popup = AdfPage.PAGE.findComponentByAbsoluteId('p1');
    if(popup!=null){
      if (evt.isBusy()){
        popup.show();
      }
      else{
        popup.hide();
        //remove busy state listener (important !!!)
        AdfPage.PAGE.removeBusyStateListener(popup,
                                             handleBusyState);
      }
   }
 }
</af:resource>
```

In this example, the JavaScript code above has been added to the ADF Faces page source, using the `af:resource` tag. In a real application development project, it would be better to place this code into an external JavaScript file or a page template.

The "enforcePreventUserInput" function is called by the `af:clientListener` and looks up the `af:popup` component to launch. Once the handle to the component is found, a callback method is defined that is called when the application starts getting busy and when it returns from the task. In this example, the busy state information is used to show the popup and, once the task completes, to hide it. After this, the "preventUserInput" method is invoked on the action event to show a glass pane that prevents users from interacting with the application. The glass pane is automatically removed when the application busy state is released.

Keyboard event handling

Typical use cases for client-side keyboard event handling are shortcuts that invoke application functionality and input filters to enforce allowed characters on a UI component. Both use cases can be implemented using JavaScript in ADF Faces.

How-to filter user input on UI input components

A use case for filtering user input is the ADF Faces table. When dragging a collection as a table from the ADF DataControls panel to a page, developers can select the "filter" option to add a filter field on top of the table columns. Application users use the filter fields to further refine the result set returned by a business service query. If a column displays numeric values, then adding alphanumeric characters in the filter may cause errors that developers can avoid by filtering the user input.

The following example illustrates the page source and the JavaScript code used to suppress numeric data entry in a table filter:

```
<af:column sortProperty="FirstName" …
                    filterable="true">
   <af:inputText value="…">
      …
   </af:inputText>
   <f:facet name="filter">
     <af:inputText id="inputText1" simple="true"
                   value="#{vs.filterCriteria.FirstName}">
       <af:clientListener method="blockNumbers"
                          type="keyDown"/>
       </af:inputText>
   </f:facet>
</af:column>
```

Table filters in ADF bound tables are enabled by developers checking the "Filtering" select option in the configuration dialog that opens when a collection is dragged from the DataControls panel in JDeveloper and dropped as a table onto the page.

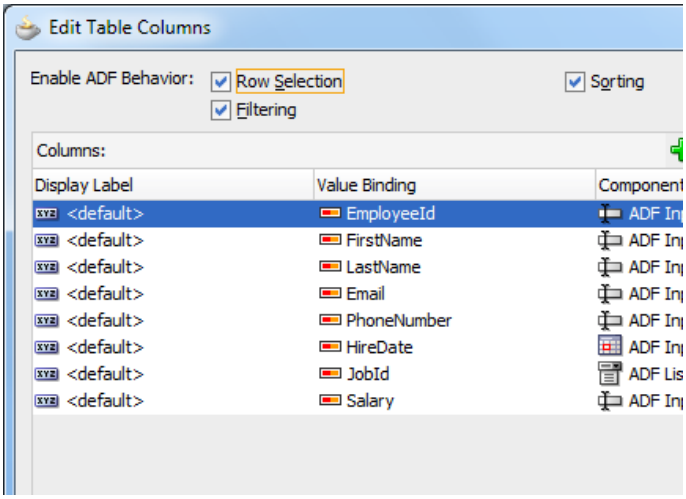


Figure 11: Table edit dialog to enable table filtering

The default filter is internally rendered by an input text field. To customize the table filter, an ADF Faces UI component can be added to the `af:column` "filter" facet. In this example, the table filter component is customized with an af:inputText field. The customized input text component has an `af:clientListener` added that listens for the "keyDown" event to call the "blockNumbers" JavaScript function defined on the page or referenced in a JavaScript library.

```
function blockNumbers(evt){
  var _keyCode = evt.getKeyCode();
  var _filterField = evt.getCurrentTarget();
  var _oldValue = _filterField.getValue();

  //check for numbers
  if (((_keyCode > 47) && (_keyCode < 58))||
     ((_keyCode > 95) && (_keyCode < 106))){
     _filterField.setValue(_oldValue);
     evt.cancel();
  }
}
```

The JavaScript function reads the key code and the input text field reference from the `AdfUIInputEvent` event object that is passed in as the argument. The function checks for numeric input from the keyboard 47 – 57 and the number keypad 95 – 105. If the user key input falls into one of these ranges, the old field value is set back to the input text component, thus undoing the input.

oracle.adf.view.js.base

Class AdfKeyStroke

org.ecmascript.object.Object
  |
  +--oracle.adf.view.js.base.AdfObject
        |
        +--**oracle.adf.view.js.base.AdfKeyStroke**

public class AdfKeyStroke
extends AdfObject

Field Summary	
public static Object	A_KEY
public static Number	ALT_KEY KeyCode representing Alt key.
public static Number	ALT_MASK Modifiers, bit mask value if Alt key is pressed
public static Number	ARROWDOWN_KEY KeyCode representing Down Arrow key.
public static Number	ARROWLEFT_KEY KeyCode representing left arrow key.
public static Number	ARROWRIGHT_KEY KeyCode representing right arrow key.
public static Number	ARROWUP_KEY KeyCode representing Up Arrow key.
public static Number	BACKSPACE_KEY KeyCode representing backspace key.
public static Object	C_KEY
public static Number	CTRL_MASK Modifiers, bit mask value if Control key is pressed
public static Object	DELETE_KEY
public static Object	END_KEY
public static Number	ENTER_KEY KeyCode representing enter key.
public static Number	ESC_KEY KeyCode representing esc key.
public static Number	F10_KEY KeyCode representing F10 key.

Figure 12: AdfKeyStroke JavaScript doc

Figure 12 shows the online JavaScript documentation available for the AdfKeyStroke object.

Note: The `AdfKeyStroke` ADF Faces JavaScript object provides helper constants to test for common keys and masks, like the enter key, F1 – F12, alt key and many more. So instead of using the JavaScript code 13 to test for the enter key, AdfKeyStroke.ENTER_KEY is used.

Note: It is considered good practice to use framework defined constants and functions when possible, instead of the DOM equivalents.

How to implement global function keys

The Oracle Forms product allows developers to define global function keys that help users to access application functionality easier. For example, developers may define a function key to commit or roll back transactions, to add or delete form or table rows, to duplicate table or form rows, to launch dialogs, or to logout. The same global key functionality, though not provided as a native feature in ADF Faces, can be built in JavaScript.

Figure 13 shows a sample in which the **ctrl+k** key combination opens a dialog showing all available keyboard shortcuts.

Note: Ctrl+k is the keyboard short cut Oracle Forms users are used to.

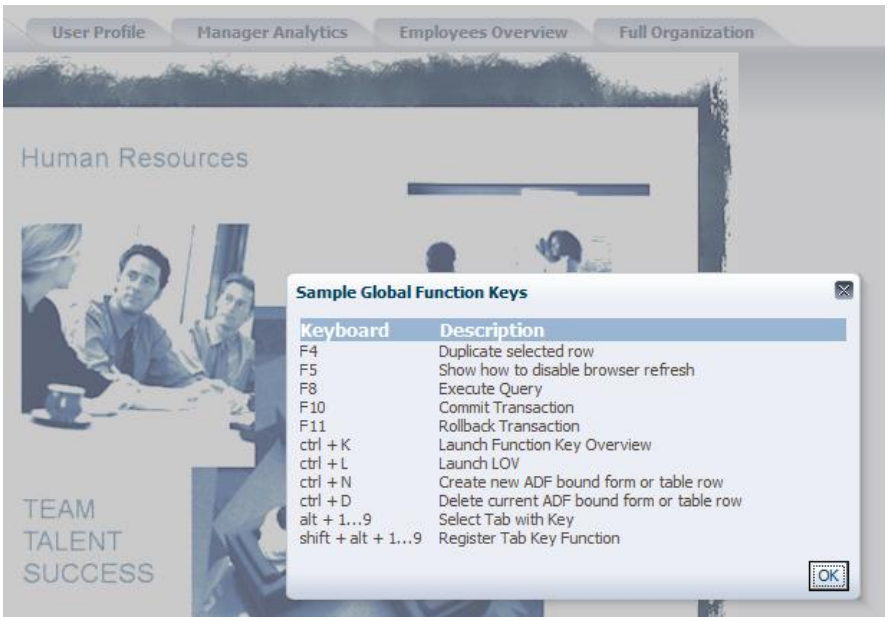


Figure 13: Sample use of global keyboard short cuts

The ADF Faces client architecture exposes the static function "registerKeyStroke" on the `AdfRichUIPeer` client object that developers can use to register function keys with a target component and a JavaScript callback. In addition, the `AdfRichUIPeer` object provides

encoding and decoding functionality to marshal, and unmarshal, the `AdfKeyStroke` object representing a specific keyboard shortcut.

Note: As mentioned earlier, `AdfRichUIPeer` is the only public class that has the "Peer" string in its name. All other classes with this string are internal and should not to be used in custom application development.

The remainder of this section describes and explains an advanced JavaScript example that show how to build a keyboard registry for ADF applications like the one that exists for Oracle Forms. Several techniques and components introduced in this paper come into play:

- An `af:serverListener` to invoke a server-side Java method in response to a client keyboard event
- The *`af:resource`* tag to load JavaScript from external libraries
- A managed bean that holds the client event method for the `af:serverListener` to read and handle the payload
- The execution of JavaScript from Java
- A "beforePhase" phase listener method on the *f:view* tag to load the JavaScript registry when the page renders

The registry function example consists of the following parts:

- A keyboard registry function
- A callback function that notifies the server about keyboard invocations
- A keyboard registry that allows developers to define the keys they want to respond to in a server-side method
- Phase listener implementation

Creating the Keyboard registry function

The keyboard registry function is responsible for registering the defined key with the callback handler. The function is invoked from a phase listener in the before RENDER_RESPONSE phase. For the example it is sufficient if the listener is defined using the beforePhase property of the `f:view` tag:

```
function registerKeyBoardHandler(serverListener, afdocument) {
 _serverListener = serverListener;
 var document =
          AdfPage.PAGE.findComponentByAbsoluteId(afdocument);
 _document = document;
 //iterate over an array of keyboard shortcuts to register with
 //the document
```

```
 for (var i = keyRegistry.length - 1; i >= 0; i--)
 {
   var keyStroke =
   AdfKeyStroke.getKeyStrokeFromMarshalledString(keyRegistry[i]);
   AdfRichUIPeer.registerKeyStroke(document,
                        keyStroke, callBack);
 }
}
```

This code references an array of registered function keys stored in an externally referenced JavaScript library. All shortcuts are registered for a single "callBack" function that is called when a registered key is used in the scope of the `af:document` component.

The callback function

The callback function is invoked for all of the registered keyboard keys. It uses client-side logging to indicate the keyboard combination a user pressed before building the marshalled key code, a string representation like "alt F4", of the pressed key combination.

Note: Different callback functions can be defined for different keyboard shortcuts. This makes sense, for example, to define some function keys to execute on the client and some function keys to execute logic on the server. The keys for client-side functions could be registered with their own callback handler to prevent unnecessary server roundtrips.

```
function callBack(adfKeyStroke) {
  var activeComponentClientId =
                  AdfPage.PAGE.getActiveComponentId();

  //send the marshalled key code to the server listener for the
  //developer to handle the function key in a managed bean method

  var marshalledKeyCode = adfKeyStroke.toMarshalledString();

  //{AdfUIComponent} component
  //Component to queue the custom event on {String} name of
  //serverListener
  //{Object} params
  //a set of parameters to include on the event.
  //{Boolean} immediate
  //whether the custom event is "immediate" - which will cause
  //it to be delivered during Apply Request Values on the
  //server, or not immediate, in which case it will be
  //delivered during Invoke Application.
```

```
  //Note that if one of the keyboard functions new rows
  //in an ADF iterator, immediate must be set to false.

   AdfCustomEvent.queue(_document,
               _serverListener,
               {keycode:marshalledKeyCode,
               activeComponentClientId:activeComponentClientId},
               false);

   //indicate to the client that the key was handled and that
   //there is no need to pass the event to the browser to handle
   //it

   return true;
}
```

To obtain the string representation of the pressed keys, the "toMarshalledString" function of `AdfKeyStroke` object is called. The key string is then passed as the payload to the server-side managed bean method referenced by the `af:serverListener`.

With the keycode, the function also passes the client Id of the currently active client UI component for the server to determine the key function to process and, if needed, access the component on which the key was pressed.

Note: Keyboard shortcuts that should invoke server-side functions conditionally can be intercepted in the callback handler. The callback function would return false for events that should be further handled by the browser.

Managed bean method to listen for the custom client event

The `af:serverListener` references a managed bean method to handle its payload. The managed bean accesses the `ClientEvent` object argument to read the key code string and the clientId of the current component. It then determines the pressed function key to execute the associated logic.

```
public void handleKeyboardEvent(ClientEvent clientEvent) {
  String keyCode =
       (String) clientEvent.getParameters().get("keycode");
  String clientId =
  (String) clientEvent.getParameters()
                      .get("activeComponentClientId");


  logger.log(ADFLogger.TRACE, "key code : "+keyCode);
```

```
  if(keyCode.equalsIgnoreCase("F4")){
    handle_f4(clientId);
  }
  if(keyCode.equalsIgnoreCase("F5")){
    handle_f5(clientId);
  }
  …
```

The "handle_f4" method for example is a helper method to be defined by the application developer to execute the task mapped to the F4 key.

Key Registry

The keys that are registered on the `AdfRichUIPeer` are defined in an array structure in an external JavaScript file. The JavaScript file is referenced from an `af:resource` tag in the JSPX page to load with the page.

```
/*
 * The array below defines the keys that are registered for
 * client to server notification. The map helps to avoid
 * unnecessary server round trips and preserves the native
 * browser functionality for keys that are not used in the ADF
 * application
 *
 * modifiers: "shift", "ctrl", "alt"
 * keys:      "TAB", "DOWN", "UP", "RIGHT", "LEFT", "ESCAPE",
 *            "ENTER", "SPACE", "PAGE_UP", "PAGE_DOWN", "END",
 *            "HOME", "INSERT","DELETE", "F1", "F2", "F3", "F4",
 *            "F5", "F6", "F7", "F8","F9", "F10", "F11", "F12"
 * Examples:  ctrl F10   --> Intercepts the ctrl key and F10
 *                           pressed in combination
 *             ctrl K    --> Intercepts the ctrl key and K key
 *                           pressed in combination
 *             F9        --> Intercepts the F9 key
 */
var keyRegistry = new Array();

//Only add keys that are actually use in the application. Don't
//add keys that have no action attached as it only produces
//unnecessary traffic
//
//GENERIC FUNCTION KEYS
//----------------------
```

```
//Generic keys operate on the ADF binding to create a row, delete
//a row, duplicate a row, show LOV, commit or roll back the
//transaction

//The keys work with all ADF applications and the source code,
//and therefore can be copied and pasted into custom applications
//
keyRegistry[0]="F1";
keyRegistry[1]="ctrl K";
keyRegistry[2]="ctrl L";
keyRegistry[3]="ctrl N";
keyRegistry[4]="ctrl D";
keyRegistry[5]="F4";
keyRegistry[6]="F5";
keyRegistry[7]="F8";
keyRegistry[8]="F10";
keyRegistry[9]="F11";
```

As shown in the registry key definition, the marshalled string version of the keys is used to define the key or key combination the client-side should listen for. All keys that are not registered on the `AdfRichUIPeer` object either perform the ADF Faces default functionality or browser native operations.

Note: Any combination of keyboard short cuts that include the numbers 8 and 9 ("alt 9", "alt 8" …) incorrectly causes assertion errors if assertion is enabled for Oracle JDeveloper 11*g* releases prior to 11.1.1.4. At runtime, however these combinations work without problems.

PhaseListener

The phase listener is implemented by referencing the managed bean method shown in the following code from the *BeforePhase* property of the `f:view` tag. The `f:view` tag is only available for pages, not page fragments. The `AdfRichUIPeer` object can used to register keyboard functions for any UI component. However, for global keys, the JSPX document and its `af:document` tag appear to be the best place to put this.

```
public void registerKeyboardMapping(PhaseEvent phaseEvent) {
  //need render response phase to load JavaScript
  if(phaseEvent.getPhaseId() == PhaseId.RENDER_RESPONSE){
    FacesContext fctx = FacesContext.getCurrentInstance();
    ExtendedRenderKitService erks =
                Service.getRenderKitService(
                          fctx,
                          ExtendedRenderKitService.class);
    List<UIComponent> childComponents =
```

```
                                fctx.getViewRoot().getChildren();
    //First child component in an ADF Faces page - and the
    //only child - is af:document. Thus no need to parse the child
    //components and check for their component family type
    String id =
        ((UIComponent)childComponents.get(0)).getClientId(fctx);

    StringBuffer script = new StringBuffer();
    //build JS string to invoke registry function loaded to the
    //page
    script.append("window.registerKeyBoardHandler
                  ('keyboardToServerNotify','" + id +"')");
    erks.addScript(fctx, script.toString());
   }
}
```

Handling double-click on a table

A common user requirement is to perform a task on a tree, tree table or table row in response to a mouse double click. The ADF Faces table does not provide a native event listener to handle the mouse double click. However, this use case can be easily implemented with JavaScript.

Use case: open edit form on table

The frequent use case asked for on the OTN help forum for Oracle JDeveloper is how to open an edit form in response to users double clicking a row or node in a table, tree or tree table component. Figure 14 shows this use case implemented for an `af:table` component.

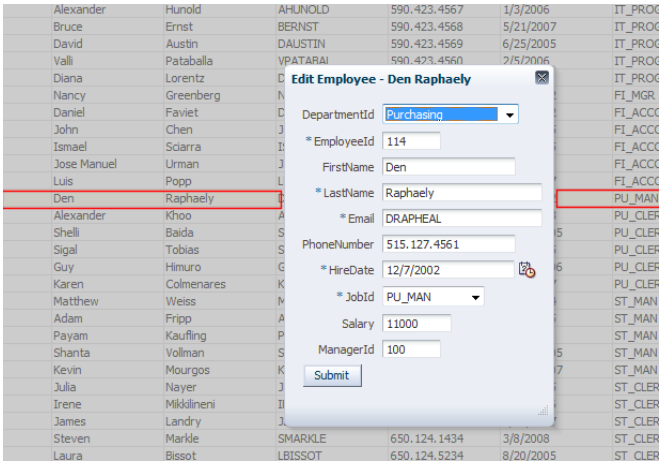


Figure 14: Open an edit dialog on table double click to edit the selected row

The solution to implement this use case is a hybrid approach of JavaScript and server-side Java. The reason for this is that this use cases requires component and ADF binding access, which is easier to do on the server than the client. In addition, this example illustrates that a JavaScript only implementation is not required and instead JavaScript is only used for what JavaServer Faces or ADF Faces don't provide.

The edit dialog sample makes use of server-side Java for opening and closing the dialog and for the cancel button in the popup. The latter accesses the ADF binding layer for the table.

```
<af:table  value="#{bindings.allEmployees.collectionModel}"
           selectionListener="
            #{bindings.allEmployees.collectionModel.makeCurrent}"
           rowSelection="single" id="t1">
  <af:column ...>
      ...
  </af:column>
  <af:column ...>
      ...
  </af:column>
  ...
  <af:clientListener method="handleTableDoubleClick"
                     type="dblClick"/>
  <af:serverListener type="TableDoubleClickEvent"
                     method="#{empbean.handleTableDoubleClick}"/>
</af:table>
```

The page source in this example defines the table with its `af:clientListener` and `af:serverListener` child tags. The `af:serverListener` is called from the JavaScript function that is invoked by the `af:clientListener`. The client listener responds to the mouse double-click on the table.

Handy for this use case, the double-click marks the clicked row as current, set by the selection listener referenced from the table's "selectionListener" attribute.

To show the selected row in an edit form, application developers only need to to drag the same collection, like an ADF Business Components View Object, as a form into a popup. In the sample, the popup is added as a child of the `af:form` tag.

```
<af:popup id="p1" binding="#{allEmployeesBean.editEmployees}"
          contentDelivery="lazyUncached">
  <af:dialog id="d2" type="none" title=" …" resize="on">
    <af:panelFormLayout id="pfl1">
```

```
      <af:selectOneChoice ...>
        ...
      </af:selectOneChoice>
      <af:inputText value= ... >
        ...
      </af:inputText>
        ...
    </af:panelFormLayout>
  </af:dialog>
</af:popup>
```

The JavaScript

The JavaScript function gets the table component handle from the event object. The table is passed as a reference in the custom server event, issued by the `af:serverListener` on the page. The double-click event itself does not need to be handled on the server, which is why it is cancelled. The `AdfCustomEvent` object message payload can be used to open a popup to align with a specific component on a page. In this example, the message payload is not used and thus left empty.

```
<af:resource type="javascript">
 function handleTableDoubleClick(evt){
   var table = evt.getSource();
   AdfCustomEvent.queue(table, "TableDoubleClickEvent",{}, true);
   evt.cancel();
  }
</af:resource>
```

The managed bean code

The method that handles the server listener event opens the popup dialog. No popup arguments are defined, in which case the dialog is launched in the center of the page:

```
public void handleTableDoubleClick(ClientEvent ce){
  RichPopup popup = this.getEditEmployees();
  //no hints means that popup is launched in the
  //center of the page
  RichPopup.PopupHints ph = new RichPopup.PopupHints();
  popup.show(ph);
}
```

Note: To be able to open a popup from Java is a new features added in Oracle JDeveloper 11.1.1.3. The `PopupHints` class used to set the popup alignment hints is an inner class of the `RichPopup` class, which is not obvious.

The onSubmitPopup method hides the displayed popup dialog and refreshes the table component to show the user edits:

```
public void onSubmitPopup(ActionEvent actionEvent) {
   RichPopup popup = this.getEditEmployees();
   popup.hide();
   //refresh the table
   AdfFacesContext adfctx = AdfFacesContext.getCurrentInstance();
   adfctx.addPartialTarget(getEmployeesTable());
}
```

The "onCancel" action listener hides the popup dialog but also resets the user changes by calling "refresh" on the current selected row.

```
public void onCancel(ActionEvent actionEvent) {
  //undo changes
  RichTable  table = this.getEmployeesTable();
  CollectionModel model = (CollectionModel) table.getValue();
  JUCtrlHierBinding treeBinding =
                    (JUCtrlHierBinding) model.getWrappedData();
  DCIteratorBinding iterator =
                                 treeBinding.getDCIteratorBinding();
  Row rw = iterator.getCurrentRow();
  rw.refresh(Row.REFRESH_UNDO_CHANGES);

  RichPopup popup = this.getEditEmployees();
  popup.hide();
}
```

Clearing error messages

ADF Faces provides client validation for component constraints like required fields.

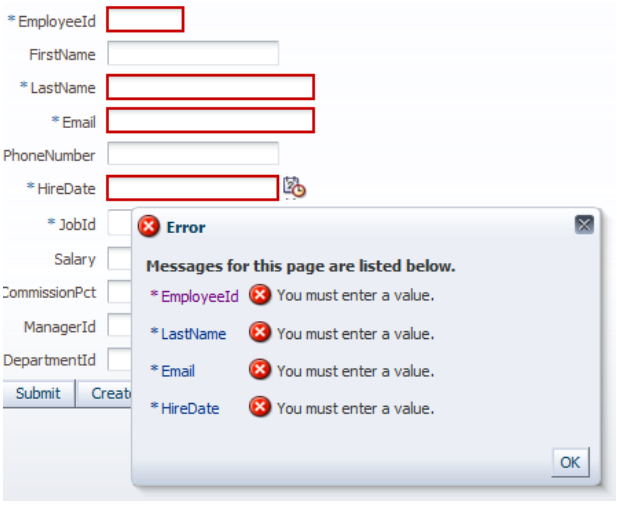


Figure 15: Input field constraint validation causing client-side errors to be displayed

If input fields in a form fail client-side validation, then the form data is not submitted to the server and an error message is displayed for the individual fields causing the error.

The error messages are displayed until the user corrects the input field values and resubmits the form, which could be irritating to users. To get rid of error messages of a failed previous form submit, application developers can use JavaScript as shown below

```
<af:resource type="javascript">
  function clearMessagesForComponent(evt){
    AdfPage.PAGE.clearAllMessages();
    evt.cancel();
  }
</af:resource>
```

The JavaScript client function is called from the focus event of a client listener added to the input fields in a form

```
<af:panelFormLayout id="pfl1">
  <af:inputText value="#{bindings.EmployeeId.inputValue}" …>
     …
    <af:clientListener method="clearMessagesForComponent"
                       type="focus"/>
  </af:inputText>
  <af:inputText value="#{bindings.FirstName.inputValue}" … >
    …
    <af:clientListener method="clearMessagesForComponent"
                       type="focus"/>
  </af:inputText>
  …
</af:panelFormLayout>
```

Note: This JavaScript function is not dependent on a specific instance of an ADF Faces page and therefore is a good candidate to be contained in an external JavaScript library file.

Don’t let JavaScript spoil the Performance Soup

Adding or using JavaScript on a page always adds overhead in the form of increased download size, additional client processing time or network round-trips. This section discusses ADF Faces client framework features and custom JavaScript development practices that ensure good client performance.

ADF Faces JavaScript download performance

One of the hot topics in Ajax application development is how to get JavaScript sources loaded to the client without blocking the page from rendering or causing delays due to the number of client server roundtrips, which is where network latency adds to the cost. The solution implemented by many Ajax libraries is to load resources on demand and to obfuscate scripts to reduce the size of JavaScript that gets downloaded for a request.

To address the initial script loading performance, the ADF Faces UI framework uses JavaScript partitioning, a feature discussed in the following.

The JavaScript library partitioning feature in ADF Faces allows developers to customize the way component scripts are loaded when requested by a page. With JS library partitioning, script files are bundled based on their component relationship and/or frequency of use without creating a dependency between ADF Faces application and the physical script files. Developers who don't build a custom library partition for their application leverage the smart default configuration in ADF Faces.

As explained in the architecture overview earlier in this paper, the smallest script source unit is the JavaScript component class, which could be a peer object or a public component interface. JavaScript library partitioning in ADF Faces allows developers to register component classes into logical units called *feature*.

A feature manages the script dependencies of an ADF Faces component and ensures that all required resource files are available on the client when a component is contained in a view. For example, the Panel Stretch Layout component in ADF Faces is defined in two client-side classes:

oracle/adf/view/js/component/rich/layout/AdfRichPanelStretchLayout.js
oracle/adfinternal/view/js/laf/dhtml/rich/AdfDhtmlPanelStretchLayoutPeer.js

The two files are referenced by the "AdfRichPanelStretchLayout" feature, allowing application developers and component developers to reference the component without knowing about the physical files and file locations involved. Features are usually defined by component developers and seldom by application developers.

To further reduce network round-trips when downloading component features, ADF Faces bundles a set of features on partitions. A partition combines features that are commonly used in conjunction. For example, tables have columns and columns usually render their cell content as output text or text input. To further improve the loading performance of the initial application page, developers may keep the page complexity to a minimum by grouping all its components into a custom partition.

To configure features and partitions, two XML formatted configuration file types are used: the JavaScript feature configuration file and the JavaScript partition configuration file. For example, the Panel Stretch Layout component feature, which is one of the default definitions in ADF Faces, is defined as follows:

```
<feature>
  <feature-name>AdfRichPanelStretchLayout</feature-name>
  <feature-class>
     oracle/adf/view/js/component/rich/
     layout/AdfRichPanelStretchLayout.js
  </feature-class>
  <feature-class>
     oracle/adfinternal/view/js/laf/dhtml/rich/
     AdfDhtmlPanelStretchLayoutPeer.js
  </feature-class>
</feature>
```

Component developers who want to leverage the ADF Faces library partitioning feature for their component add the feature definition into the META-INF directory of the component JAR file. For ADF Faces to find the feature definition at runtime, the configuration file must have a name of "adf-js-features.xml".

To create a partition, developers edit the partition configuration file, referencing the logical feature names to include:

```
<partition>
  <partition-name>AllThatStretches</partition-name>
  <feature>AdfRichPanelStretchLayout</feature>
  <feature> … </feature>
</partition>
```

In contrast to feature definitions, which are defined in multiple configuration files located in the component JAR, partitions are configured using a single file, "adf-js-partitions.xml", located in the web application's WEB-INF directory. If no "adf-js-partitions.xml" file is found in the WEB-INF directory, then ADF Faces falls back to its default configuration that contains a sensible set of partitions that usually ensure good application JavaScript loading performance.

At runtime, ADF Faces looks at the required features and the partitions those features are contained in to then dynamically add a script reference to the end of the dynamically generated HTML page.

For more information about the ADF Faces JavaScript library partitioning feature see the Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development.

Note: The ADF Faces default setting is chosen sensibly. It requires a good understanding of JavaScript library partitioning and the features used on a page to build a partition that performs better than the default. For this reason it is advisable to treat library partitioning as an advanced topic that developer should look at after completing application developments and performance testing.

Use external JS Libraries

Footprint matters. Oracle recommends keeping custom JavaScript code in external source files that are referenced from `af:resource` tags in the pages. This strategy improves performance for JavaScript code that is reused on many ADF Faces pages and stored in JavaScript files of a reasonable size. Code that is used on all pages should be referenced from a page template that then is applied to ADF Faces pages. This way the browser cache is leveraged, so that source files are not repeatedly downloaded.

Disable debugging and diagnostic settings

The ADF Faces component framework provides settings for developers to debug their client application code and to analyze the generated page output. For best performance, ensure that the following context parameters are set in web.xml or not included, in which case the Oracle smart defaults are used.

Test automation

Test automation generates a client component for every component on the page, thus increasing the number of JavaScript objects executed on the browser client. The default content rendering in ADF Faces is optimized in that ADF Faces pages render in a mix of server-side created HTML and client-side JavaScript objects. Test automation suppresses this optimization and should be disabled. To disable test automation, the following context parameter should be set to false or not included in the web.xml file:

```
<context-param>
  <param-name>
    oracle.adf.view.rich.automation.ENABLED
  </param-name>
  <param-value>false</param-value>
</context-param>
```

Assertions

Assertions are not only used in custom code but throughout the ADF Faces client platform. Running an application with the assertion feature enabled will reduce performance even if no error is detected. To disable assertions, the following context parameter should be set to false or not included in the web.xml file:

```
<context-param>
  <param-name>
    oracle.adf.view.rich.ASSERT_ENABLED
  </param-name>
```

```
  <param-value>false</param-value>
</context-param>
```

JavaScript profiling

Profiling allows developers to gain information about JavaScript function compilation, execution time, and size. When the JavaScript profiler is enabled, an extra roundtrip occurs on every page in order to fetch the profiler data. To avoid this extra roundtrip, the profiler should be disabled. To disable JavaScript profiling, the following context parameter should be set to false or not included in the web.xml file:

```
<context-param>
  <param-name>
    oracle.adf.view.rich.profiler.ENABLED
  </param-name>
  <param-value>false</param-value>
</context-param>
```

Debug mode

When resource debug mode is enabled, no HTTP response headers are set telling the browser or Oracle WebCache that resources like JavaScript libraries, CSS, or images can be cached. Caching, however, is key to improving client performance, which is why the debug mode must be disabled at runtime. To disable debug mode, the following context parameter should be set to false or not included in the web.xml file:

```
<context-param>
  <param-name>
    org.apache.myfaces.trinidad.resource.DEBUG
  </param-name>
  <param-value>false</param-value>
</context-param>
```

Enable JavaScript obfuscation

For better performance, ADF Faces framework libraries are obfuscated, which reduces them in their size. For diagnostic reasons, it may be useful to disable obfuscation. It is recommended to keep obfuscation enabled for production systems since the reduction of the JavaScript content size is up to 50%, thus reducing the time to download. To enable JavaScript obfuscation, the following context parameter should be set to false or not included in the web.xml file

```
<context-param>
  <param-name>
```

```
    org.apache.myfaces.trinidad.DEBUG_JAVASCRIPT
  </param-name>
  <param-value>false</param-value>
</context-param>
```

Enable Library Partitioning

JavaScript partitioning is enabled by default to enforce loading on demand of JavaScript component objects and their peers. To enable JavaScript partitioning, the following context parameter should be set to false or not included in the web.xml file

```
<context-param>
  <param-name>
    oracle.adf.view.rich.libraryPartitioning.DISABLED
  </param-name>
  <param-value>false</param-value>
</context-param>
```

What else?

Less JavaScript means better performance. However, avoiding the use of JavaScript often is only wishful thinking when building browser-based rich Internet applications.

Therefore, if the use of JavaScript cannot be avoided, it should be used with care so custom JavaScript code does not break ADF Faces framework functionality. Developers should always work with the framework, not against it.

For example: Instead of issuing custom XMLHTTP requests to the server, developers should check using the `af:serverListener` and custom events can provide the same functionality.

Custom XMLHTTP requests run outside of the ADF Faces and ADF request lifecycle, which means there is no guarantee that the environment is prepared for the call. Also, XMLHTTP requests are not an option for implementing improved parallel processing in ADF Faces, especially not when the ADF binding is involved.

Another hint to prevent performance problems with custom JavaScript code is to read 3rd party documentation and articles written about JavaScript programming. Sometimes little details make a big difference in client-side performance. For example, though it is possible to define variables in JavaScript function without declaring them with a leading "var", this leads to bad performance because the variable is attempted to be found in memory before it is created.

Best Practice 9: The framework default behavior should be respected. The rich client component architecture and its APIs are performance optimized. Component functionality therefore should not be changed by call to the client component *prototype* handler.

Security

Just for developers to be aware: The only option for secure JavaScript is no JavaScript! There is no such thing as safe or undiscoverable JavaScript. Scripts may be obfuscated to make them harder to read, but not impossible to read. With DOM inspection tools like Firebug in Firefox it's easy to debug and manipulate scripts on the client. So don't rely on security implemented in JavaScript alone.

Component security

To harden the ADF Faces client architecture, not all ADF Faces component properties are exposed to JavaScript. For example, the "disabled" property of the command button cannot be accessed and changed on the client.

If a use case requires that a protected component state is changed on the client then this needs to be implemented by a call to server-side logic using the `af:serverListener` component.

Application Security

Reliable authorization cannot be implemented with client-side JavaScript alone. All authorization must be re-enforced on the server. Though the ADF Faces client component API does not expose sensitive properties, such as the component setDisabled() property, all UI components have a browser DOM representation for their rendered markup, which can be used by a hacker to enable a component that initially renders disabled. If a user is not supposed to read or access components, then best practice is to set the component "rendered" property to false. Everything else, including hidden fields, shows in the DOM.

Best Practice 10: JavaScript should not be used for implementing application security.

Conclusion

JavaScript in ADF Faces should only be used if it has to be used and never only because it can be used. The use of JavaScript in ADF Faces applications should be by exception. Solutions to a problem should be first looked for in the JavaServer Faces and ADF Faces API set and components.

With this in mind and the rules of best practice at hand, JavaScript is good to use in the context of ADF Faces applications.

JavaScript Best Practices at a Glance

- Developers should understand the ADF Faces client architecture before using JavaScript. JavaScript doesn't make things better without knowing how to use it and where to put it.
- ADF Faces protected or private JavaScript functions and variables shown by browser DOM introspection tools should not be used.
- Developers must ensure a component to exist before accessing it from JavaScript. For a component to exist, the component's clientComponent property must be set to true or an `af:clientListener` tag must be added.
- No component id that appears in the generated HTML page output should be hardcoded into JavaScript functions. Client id values in the HTML markup may change, even between runs of the very same view.
- All events that don't need to propagate to the server should be cancelled.
- Client to server calls must be used sensibly to prevent unnecessary network roundtrips.
- No knowledge about framework internal implementation code should be used in custom JavaScript code. Only public APIs and constants should be used.
- Exceptions should be handled gracefully and not just suppressed.
- The framework behavior must be respected. JavaScript should not be used to change and override component functionality definition by calls to the JavaScript *prototype* handler.
- Application security cannot rely on authorization implemented in JavaScript. ADF Security should be used instead.

Using JavaScript in ADF Faces Rich Client Applications

January 2011
Author: Frank Nimphius
Contributing Authors: [OPTIONAL]

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Oracle is committed to developing practices and products that help protect the environment

0109