Leveraging the IBM BPM Coach Framework in Your Organization

Chapter 4: SPARK UI Toolkit – Unabridged

Authors:

Eric Ducos (Salient Process)
Matthew Oatts (Salient Process)
John McDonald (Salient Process)

Reviewers:

Dennis Parrott (IBM)
Claudio Tagliabue (IBM)
Brian French (Salient)

Note: This is the Unabridged version of the SPARK material (Chapter 4) from the IBM BPM Redbook: Deliver Modern UI for IBM BPM with the Coach Framework. We encourage all readers to take advantage of the material in the full Redbook! The information contained in this Unabridged SPARK version includes everything from Chapter 4 of the Redbook plus additional SPARK guidance.
## Contents

4. SPARK UI Toolkit - Unabridged ..................................................................................................... 4

4.1 Preface to Unabridged Edition .............................................................................................. 4

4.2 Introduction ........................................................................................................................ 4

4.2.1 Understanding the value of the SPARK UI Toolkits .......................................................... 4

4.2.2 Developer Experience ................................................................................................... 5

4.2.3 Underlying patterns and principles ................................................................................ 5

4.2.4 Modern, lightweight, consistent across BPM versions ..................................................... 6

4.3 Understanding the IBM and Salient Process partnership ........................................................ 7

4.4 Basic BPM UI concepts with SPARK ....................................................................................... 7

4.4.1 Controls and configuration properties ........................................................................... 7

4.4.2 Methods, events, and addressing .................................................................................. 9

4.4.3 Optional data binding .................................................................................................. 30

4.4.4 Validation .................................................................................................................. 31

4.5 UI layout ................................................................................................................................ 38

4.5.1 Horizontal and vertical layouts .................................................................................... 38

4.5.2 Tabbing or stacking UI content .................................................................................... 44

4.5.3 Displaying modal content ............................................................................................ 45

4.5.4 Wells ......................................................................................................................... 48

4.6 Calling AJAX services .......................................................................................................... 49

4.6.1 Service Call control ..................................................................................................... 50

4.7 Responsiveness ................................................................................................................. 54

4.7.1 Classic responsiveness ................................................................................................ 54

4.7.2 Enhanced Web Component-based responsiveness ....................................................... 56

4.7.3 Coach View responsiveness ......................................................................................... 61

4.8 Working tabular and repeating data .................................................................................... 65

4.8.1 Table and Service Data Table controls .......................................................................... 65

4.8.2 Layouts and repeating data .......................................................................................... 84

4.9 Formulas ................................................................................................................................ 86

4.9.1 Syntax for control references ...................................................................................... 87

4.9.2 Formulas referencing specific controls ......................................................................... 89

4.9.3 Formulas for control aggregates .................................................................................. 90

4.9.4 Working with data directly .......................................................................................... 92
4.9.5 Formulas for non-numerical computations .............................................................. 94
4.9.6 Formulas for control value initialization ................................................................. 95
4.9.7 Circular references .................................................................................................... 96
4.10 Reporting and analytics .............................................................................................. 97
  4.10.1 Charts ..................................................................................................................... 97
  4.10.2 Working with Tables and Charts ............................................................................ 107
  4.10.3 Drill-down behavior .............................................................................................. 110
4.11 Creating SPARK-compatible controls ....................................................................... 120
  4.11.1 Sample "Rating" control ....................................................................................... 121
  4.11.2 Sample Credit Card composite Coach View ......................................................... 128
4.12 Solutions for common patterns and requirements ...................................................... 137
  4.12.1 Working with large forms ..................................................................................... 137
  4.12.2 Server-side paging ............................................................................................... 141
  4.12.3 Cascading Selects ............................................................................................... 147
4.13 Useful Tips .................................................................................................................. 147
  4.13.1 Positioning SPARK UI controls ............................................................................ 147
  4.13.2 When to use diagram logic vs. encapsulated logic ............................................... 148
  4.13.3 Debugging logic .................................................................................................... 149
4.14 More on the SPARK UI Toolkit ................................................................................... 155
4. SPARK UI Toolkit - Unabridged

4.1 Preface to Unabridged Edition

This is the unabridged edition of Chapter 4 of the Redbook “Using the IBM BPM Coach Framework.” This unabridged edition contains approximately fifty more pages than the edited edition which went into the published Redbook. The intended audience for this edition is daily practitioners who want a deeper dive into how the SPARK UI Toolkit works. After reading this, practitioners should have a great understanding of the SPARK UI Toolkit.

4.2 Introduction

The SPARK UI Toolkit is an IBM BPM UI toolkit created by Salient Process, a Premier IBM Business Partner specializing in IBM Smarter Process consulting services and innovation.

IBM and Salient Process have partnered together to make the SPARK UI Toolkit the UI toolkit of choice for IBM BPM customers. There are already efforts underway to incorporate the SPARK UI Toolkit into the IBM BPM product as of the time of this Redbook’s publication. For more information on this partnership and future plans for the SPARK UI Toolkit and IBM BPM UI, see Section 4.3 below.

There is a comprehensive deep dive beginning in Section 4.4 of this Chapter that starts with simple core concepts and builds upon that foundation to explain all there is to know about the developing UI with the SPARK UI toolkit.

**Note:** While UI developers are encouraged to read the Chapter in its entirety to truly become experts, it is recommended that all developers using SPARK should at least read Section 4.4 to take advantage of key benefits of the SPARK UI Toolkit

4.2.1 Understanding the value of the SPARK UI Toolkits

The SPARK UI Toolkit:

- Increases UI developer productivity up to three to four times faster than using traditional methods and decreases maintenance costs by avoiding UI complexity
- Achieves the productivity increase through an efficient and intuitive development experience in combination with reduced skills expectations (primarily JavaScript, limited HTML or CSS, and no Dojo, AJAX, RWD, jQuery, or AngularJS required)
- Provides 90+ responsive and configurable controls, which can adapt to the form factor of the device running the Coach and are suitable for both production and fast-build “proof-of-concept” scenarios
- Includes with every control a simple and powerful event-based framework that creates a consistent approach for validation, formula-based computations, and cross-control interaction
- Optimizes UI performance by using controls that support lazy loading and server side pagination that can support complex UIs and large tabular data sets.

The SPARK External Participant Toolkit can extend the reach of IBM BPM to include external participants (see Chapter 6 of full Redbook)
The SPARK Portal Builder Toolkit provides a set of simple portal controls used to build dashboards and custom portals including a Get Next Task capability (see Chapter 5 of full Redbook)

4.2.2 Developer Experience

IBM BPM UI developers can use the 90+ controls in the SPARK UI Toolkit to address a broad range of UI requirements. While the core set of controls is relatively small (around 25), other more specialized ones (for example Slider, Signature, Video, Popup Menu, Tooltip) help efficiently address more sophisticated UI requirements.

Detailed documentation and how-to articles are available\(^1\) for all SPARK UI Toolkit controls.

UI developers can still take advantage of patterns and capabilities that currently exist as best practices for IBM BPM UI development, as described in Chapter 2. However, the SPARK UI Toolkit extends and streamlines the Coach Framework programming model in significant ways:

- Controls on a page or a view can easily refer to and talk to each other
- Each control can react to all sorts of events (on click, on key press, on tab change, on timeout, on row deleted, etc.) and business logic can intuitively be expressed and attached to those events to create an interactive end-user experience
- Reliance on data binding is optional. This eliminates complexity and "glue" artifacts commonly associated with BPM UIs. A data binding is only needed when a control needs to synchronize with meaningful business data
- The enhanced SPARK UI development approach is designed after a well-known effective and efficient development model of controls composed on a canvas with properties, methods and events. The complexity stops there and all properties, methods, and events are well documented per control.
- When the values of controls/fields on a form are computationally related (whether numerical or not), Excel-like formulas can be used to automatically calculate and update values (in single fields or tables) on a page/view without having to write code

\[\textbf{Note:} \text{BPM UI development can become much more effective and efficient by exploiting the Coach Framework programming model enhancements provided by the SPARK UI Toolkit, and the resulting solutions are consistently more lightweight, more maintainable, and more reusable.}\]

4.2.3 Underlying patterns and principles

A foundational principle underlying the SPARK UI Toolkit is to allow the developer to focus on solving business problems efficiently and with as few technical side-tracks as possible. Technically-speaking, this means there is no need to learn another complex JavaScript library or framework, no need to delve into the internals of controls, duplicate assets, create large amounts of glue constructs, and no need to part with well-understood and intuitive UI development approaches.

\(^1\) See SPARK UI documentation and articles at https://support.salientprocess.com/spark-ui-controls/
Note: To use the SPARK UI Toolkit effectively, a BPM UI developer should become familiar with **four core patterns**: Configuring controls, referencing and calling other controls, attaching logic to events, and using formulas. These patterns are all explained in this chapter.

Once the four core SPARK UI patterns are understood, the extent of a developer's effort is to look up the capabilities (configuration options, events, and methods) of controls that may not have been used before, and exploit those capabilities to implement business requirements. Because all controls follow a strictly consistent approach across the four core patterns, gaining familiarity with a new control is a simple and predictable experience.

4.2.4 Modern, lightweight, consistent across BPM versions

With very few exceptions (for example charting, maps), the SPARK UI Toolkit has no dependencies on extensive libraries and frameworks beyond the Coach Framework, HTML5 and CSS3.

This makes the toolkit extremely lightweight with controls that are inherently optimized to work with IBM BPM and the underlying Coach framework. It also virtually eliminates reliance on sometimes thick layers of additional library or framework-specific processing and legacy cross-compatibility behaviors.

The explicit HTML5 and CSS3 dependency however mandates the use of a modern browser with comprehensive support for HTML5 and CSS3 such as Chrome, Firefox, Internet Explorer 11+, or Safari.

HTML5 and CSS3 support requirements for SPARK are more stringent than IBM BPM's. For example, whereas Internet Explorer 9 support is only deprecated for IBM BPM 8.5.7, it is not supported at all for the SPARK UI toolkit from IBM BPM 8.5.0.1 and up.

NOTE: The SPARK UI Toolkit requires a web browser with comprehensive support for HTML5 and CSS3.

IBM BPM is a constantly evolving platform. The introduction of new features can be valuable to some and disruptive to others, especially when a new feature entails the deprecation of an older one.

While SPARK takes full advantage of the latest IBM BPM product capabilities (for example, SPARK-based UIs in Client-Side Human Services can use the full SPARK programming model right from Client-Side scripts to access page controls and manipulate them), the toolkit also attempts - as much as possible - to normalize the experience across BPM product versions.

It does so by relying on constructs that remain consistent from one product version to the next (for example SPARK-based UIs can be virtually Heritage or Client-Side Human service-agnostic). By design, the toolkit works consistently across BPM 8.5.0.1 through BPM 8.5.7 and is jointly planned by IBM and Salient Process to be fully incorporated into the IBM BPM product.
4.3 Understanding the IBM and Salient Process partnership

In June of 2016, IBM announced a partnership with Salient Process to license the SPARK toolkits with the objective to incorporate the SPARK UI toolkit features into the IBM BPM Platform. More information about that partnership, including how you can order SPARK through IBM, can be found in the IBM announcement:


4.4 Basic BPM UI concepts with SPARK

This section presents key concepts that are specific to the SPARK UI Toolkit and can help a BPM UI developer use the toolkit effectively and efficiently.

4.4.1 Controls and configuration properties

Control configuration, the most basic aspect of SPARK controls, is common with most other Coach Framework-based Coach Views and is accessed in the usual way in IBM Process Designer as illustrated in Figure 1 below:

![Figure 1 General SPARK control configuration](image)
Common configuration categories

Many aspects of a control's appearance, behavior, and other runtime characteristics can be specified through configuration (and can be changed later, at runtime). For consistency in the UI development experience, most configuration options are grouped under consistent categories. Some of the more common ones include:

- Formula: Used for the value computation of the control (more on formulas later in this chapter)
- Behavior: General behavior-related control-specific options
- Appearance: Options including styling, coloring, layout, typography, labeling
- Performance: Options to manage processing-intensive behaviors in repeating controls such as tables and layouts for large data sets
- Responsive: View width-sensitive settings that automatically adjust layout and appearance of a control based on the view that contains it (conventional page width-based adaptive configuration options are also available through the IBM Process Designer Web Editor)
- Events: Used to attached logic to various events specific to a control (more on events in the next Events section)

Figure 2 provides an example of common configuration categories for a Button control:

![Common configuration options organized in consistent categories across controls](image)
4.4.2 Methods, events, and addressing

SPARK UI controls on a page or in views can access each other and call methods on other controls. They can also respond to events triggered programmatically or by user interactions (for example button clicks, tabs changed, keys pressed, data changed). The combination of event triggers, the ability to refer to other controls and to call various methods on those controls, provides a very simple mechanism to create sophisticated behaviors.

Example 1 below shows how a developer could, from a Button control on-click event, change the text content of an Output Text control:

```
// Use button label to set content of the text control
text1.setText("Text set from button: " + button1.getText());
```

The above example should look very intuitive to UI developers because it’s simple and familiar, and it focuses on the problem - namely in this case: In the "on-click" event of a button, access another control and make it do things using calls to methods.

Methods

All SPARK controls have methods. Getters, setters and various action methods (for example to change the color of a button, to set the title of a panel, to expand a collapsible section, to refresh a table backed by AJAX, to make an image visible, and so on)

Example 2 shows how some methods simply abstract certain Coach Framework-specific constructs:

```
button1.setVisible(false)

...has precisely the same effect as
button1.context.options._metadata.visibility.set("value", "NONE")

Or

text1.setText("Text set from button: " + button1.getText())

...behaves exactly like
text1.context.binding.set("value", "Text set from button: " +
button1.context.binding.get("value"))
```

Other methods are specific to particular SPARK controls and have no counterpart in the Coach Framework. As shown in Example 3:
Example 3 Control methods unique to the Coach Framework SPARK UI extensions

\[ \text{table1.search(0, "Acme", false, true)} \]

...displays only rows in a table control where the first column (at index 0) contains the string "Acme".

All SPARK controls provide documented methods and can be accessed from the SPARK support site\(^2\).

Figure 3 shows a few methods for the Button control:

Figure 3 Online documentation for control methods

- Common methods for all controls

Certain methods are common to all SPARK controls, Table 1 provides the list of notable ones:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>boolean</td>
<td>Enabled/read-only flag (true to enable view, false to disable/make read-only)</td>
</tr>
<tr>
<td>required</td>
<td>boolean</td>
<td>Enable/disable required flag for control</td>
</tr>
</tbody>
</table>

Check https://support.salientprocess.com/spark-ui-controls/, then the JSDoc link for each control.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setVisible(visible, collapse)</td>
<td>Sets view visibility. The collapse flag determines whether the space occupied by the hidden view is collapsed or displays as blank space. For controls that have no visual representation (for example the Event Subscription control).</td>
</tr>
<tr>
<td>isVisible()</td>
<td>Self-explanatory</td>
</tr>
<tr>
<td>setEnabled(enabled)</td>
<td>Turns a view’s editable state on or off (assuming the control supports such a state)</td>
</tr>
<tr>
<td>isEnabled()</td>
<td>Self-explanatory</td>
</tr>
<tr>
<td>isBound()</td>
<td>Indicates whether a view is bound to data or not</td>
</tr>
<tr>
<td>setData(data)</td>
<td>Equivalent to context.binding.set(&quot;value&quot;, val). Some controls provide more specialized methods such as setText() for a Text control or setDate() for a Date Picker control - but those methods are merely aliases of setData.</td>
</tr>
<tr>
<td>getData()</td>
<td>Equivalent to context.binding.get(&quot;value&quot;). Some controls provide more specialized methods such as getText() for a Text control or getDate() for a Date Picker control - but those methods are merely aliases of getData.</td>
</tr>
<tr>
<td>addClass(added, replaced)</td>
<td>Adds, replaces or removes a CSS class from the Coach View's top-level DOM element (referenced by context.element). added is a string with 0 or more space delimited CSS class names to add to the view. replaced is a string with 0 or more space delimited CSS class names to remove and replace with the added class names. To remove a CSS class from a view, specify &quot;&quot; for added and the class name(s) to remove for replaced.</td>
</tr>
</tbody>
</table>

**Events**

Events are key to interactivity in the SPARK UI Toolkit. They allow controls to respond to user, device, and various programmatic triggers.

Examples of actions triggered by events could include:

- An alert displayed when a button is clicked
- Validation performed when an item in a drop-down list is selected
- An AJAX service fetching data when a collapsible panel is expanded
- A chart refreshed when a popup menu item is tapped
- A label updated with a new value every time a timer ticks
- A table filters its content when a user types search text
- An output text displaying a result when an AJAX service returns
All SPARK control events are exposed in a similar way in the Events category of the control's configuration. Figure 4 provides an example of a button that shows an alert saying "Hello" and using the name of a person entered in the text control:

![Diagram of a button with alert](image)

**Figure 4 Specifying logic in a button click event**

For a more compact and convenient notation in inline event logic, SPARK UI controls can be referred to in shorthand using the \${<control-id>} notation. The inline event logic shown in Figure 4 can be rewritten as in the notation shown in Figure 5:

```javascript
alert("Hello " + $\{PersonName\} + ")
```

**Figure 5 Shorthand notation for control references in inline event logic**

**Tip:** Don’t confuse the \${<control-id>} reference lookup notation with the jQuery `$("<selector>")` notation. The SPARK UI Toolkit doesn’t depend on jQuery (though it can peacefully coexist with it).
Note: The SPARK ${ }$ notation is only valid when used in inline event logic. Proper syntax expects curly braces only and does not use quotes for the control id. '${PersonName}' is valid whereas '${"PersonName"}', $(PersonName), or $('PersonName') are not.

Events and Coach Framework boundary events

Buttons, Icons, and a few other controls emit boundary events when clicked or otherwise activated. For most of those controls, the events fired prior to emitting the boundary event (such as on button click) can inhibit the boundary event by explicitly returning `$false$.

In Figure 6, the boundary event inhibitor pattern provides a convenient way to add confirmation prior to navigating away from a Coach.

Figure 6 Boundary event inhibitor pattern (design time)

The resulting runtime behavior is shown in Figure 7:

Figure 7 Boundary event inhibitor pattern (runtime behavior)
Event context variables

Sometimes context is important when handling events. For example, a Text control may need to restrict the length of the content or the input of certain characters or values - to do so the input event logic would examine the attempted input (passed in the "on input" event as a context variable) before allowing the new potential content or not.

Figure 8 illustrates how this scenario could be implemented:

Table 2 shows the context variables that the Text > On Input event provides to the inline event logic:

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>me</td>
<td>Similar to the “this” concept: Refers to the control that emitted the event</td>
</tr>
<tr>
<td>view</td>
<td>Refers to the control's first parent view that is not merely decorative (often this means the composite Coach View that contains the control)</td>
</tr>
<tr>
<td>current</td>
<td>Text content in the control prior to the change from the input</td>
</tr>
<tr>
<td>potential</td>
<td>Attempted content - will become the new text content unless the logic returns false</td>
</tr>
<tr>
<td>selection</td>
<td>Location of cursor in text control or boundaries of the selection if text was selected and replaced by new content</td>
</tr>
</tbody>
</table>

The JS Doc for each control (see the JS Doc links for each control in the online SPARK control reference) details not only the methods but also the events supported by the control and their associated context variables as appropriate.
Note: The "me" context variable is available in all events for all controls. It is a convenient reference to the control that emitted the event. The "view" context variable is also similarly available and points to the view containing the control.

Because these variables are common to all controls, they are never explicitly mentioned in the JSDoc as context variables.

Invoking non-inline event logic

Inline events are simple and convenient, but as long as IBM Process Designer only allows a single line for those events, specifying a lot of inline logic can be cumbersome.

Because inline event logic is purely JavaScript at runtime, functions can easily be called from an event handler. In Figure 9, the `calculate()` function (defined in a script block at the page level) is called on button click:
Figure 9 Calling a page-level function from an event handler

**Note:** There is no limitation to the globally-defined functions and globally-accessible objects that can be called from an inline event handler. If a function requires parameters, they can be passed in regular JavaScript manner. Context event variables can also be passed to externally-defined functions.

A key benefit a reusable Coach Views should include the ability to encapsulate not only the child views laid out on the containing view's canvas, but also their behavior. The
SPARK UI Toolkit allows a composite Coach View to host the business logic that is invoked by the various controls that it contains - thus creating a truly self-contained, fully encapsulated custom reusable Coach View made from smaller parts.

**Hint:** Full Coach View encapsulation is a very significant benefit of the SPARK UI Toolkit. Don't miss the following part!

Figure 10 shows how the "Calculate" button now calls a calculate() function defined in the containing composite Coach View [view.calculate()], not at the page-level:

![Figure 10 Calling a composite Coach View-level function from an event handler](image)

The definition of the "Calculator" composite Coach View's calculate() function is shown in Figure 11:
Figure 11 Defining a method at the composite Coach View level

Note: In a very simple step, non-reusable code (at the page level) was made reusable. The Calculator Coach View relies on no external scripts or components to perform its work.

This is, in the author's opinion, the fastest and most intuitive general approach to building truly reusable Coach Views.
Client-Side script-based event logic

As mentioned previously, SPARK provides various ways of dealing with business logic in response to events - the most portable approach being to encapsulate the business logic in a composite Coach View - as illustrated in the previous section.

If however, encapsulating logic is not important, then Client-Side scripts along with boundary events can be used to achieve a similar effect - in the more conventional Coach Framework-like way, but still using a part of the programming model extended by SPARK, as shown in Figure 12:

![Figure 12 Using the SPARK programming model in a Client-Side Coach](image-url)
Firing boundary events from any event handler

The previous example shows how SPARK can make use of Client-Side scripts in the same way as the classic Coach Framework-based approach. Unfortunately, not all controls fire boundary events, and even for those that do, a boundary event cannot always be fired by the interaction the developer needs.

SPARK overcomes this limitation by providing a way to fire boundary events for any kind of event-triggered interaction, through the **Navigation Event** control.

Figure 13 shows an adapted version of the Calculator scenario with results computed using a Navigation Event control instead of a Button. In that scenario, the calculation is triggered every time a change occurs in the operands or the operation:

---

**Note:** While using many boundary events to execute business logic is easily doable with SPARK (as shown above), boundary events can clutter the Client-Side Human Service.
Flow and boundary events are often not the best way to achieve deep reusability. Handling events at the Coach View level is usually more logical and easier to maintain. In the author’s opinion, the less "mixed" the SPARK approach, the more streamlined the solution, and the more productive the BPM UI developer.

**Events - summary**

Handlers for a multitude of events are available on SPARK controls. Logic can be run from any of those event handlers by calling methods on controls.

Event logic can be inlined or a function defined globally, or in a control's containing composite view, can be called.

Functions to be called in the parent composite Coach View can be defined in the view's Inline JavaScript section (see the Coach View's Behavior tab in IBM Process Designer) as suggested in Example 4:

*Example 4 Event-callable function defined in composite Coach View*

```javascript
this.myFunction = function(<parameters if needed>) {
    /* Business logic... */
}
```

Example 5 shows how myFunction can be invoked from an event handler of the child control:

*Example 5 Calling a function defined in composite Coach View from a control's event handler*

```javascript
view.myFunction(<parameters if needed>)
```

Controls can be referenced from event handlers using:

- `page.ui.get(<control-id>)` at the page level or in Client-Side scripts
- `this.ui.get(<control-id>)` in functions defined in Coach Views (Inline JavaScript) to refer to the child views contained in the composite Coach View

`${<control-id>}` - but only if specifying *inline* event logic (the ${ } notation is translated to a real JavaScript control reference at runtime)

`${<control-id>}` is a convenient way to refer to controls in inline event logic and works consistently at the page level and in a Coach View. UI developers are encouraged to use it.

Lastly, although BPM UI developers may be used to creating Human Service diagrams with dense wiring and client and server scripts for business logic, the SPARK UI event pattern combined with pertinent SPARK control methods helps minimize clutter, increases reusability, and focuses the developer on solving the business problem.
Tip: Using SPARK controls without taking advantage of the Coach Framework programming extensions contributed by SPARK (though quite possible) is forgoing a very significant productivity and simplification advantage provided by the toolkit. BPM UI Developers using the SPARK UI Toolkit should make sure to exploit its extended programming model.

Control referencing

Understanding how control referencing works is an important part of using the SPARK UI Toolkit effectively.

All examples of control referencing seen so far — using the ${<control-id>} notation or page.ui.get("<control-id>") (or view.ui.get("<control-id>")) — have been very straightforward.

A BPM UI developer using SPARK and following standard component encapsulation practices can be productive working with such simple kind of addressing (in addition to working within repeating controls in tables - which is covered later in this section).

Tip: Understanding the structure and implications behind SPARK addressing can help debug BPM UIs and exploit the programming model in creative ways to solve complex requirements.

Control referencing works like a directory tree in the SPARK UI Toolkit (it's a view tree instead) and the "address" of a view can include not only a control's ID but also indicators (such as "/", ".", "[<index>]") that help navigate the view tree like a directory structure using relative or absolute addressing.

Considering the "Simple Calculator" Coach composition in Figure 14:
Addressing in SPARK works as follows:

A parent view that contains children controls can access its *immediate* children using:

```html
<parent view ref>.ui.get("<child view's control id>")
```

A view that needs to access a sibling can do so in two ways

- Directly:
  ```html
  <view ref>.ui.getSibling("<sibling view's control id>")
  ```
Recap: A parent (composite) view "view1" can access an immediate child with
view1.ui.get("<child control-id>"). A view "view2" can access a sibling using
view2.ui.getSibling("<child control-id>").

What really happens in inline event handlers and ${<control-id>} references

When the inline logic of a control's event handler refers to a control id, for example
${Text1}, the query is always done from the context of the parent view (or of the Coach if
there is no parent view), and not from the event's own emitting view.

At runtime, the ${Text1} reference from the event handler's inline logic essentially
translates to me.ui.getParent().ui.get("Text1")

Note: An inline event handler - though associated with a control - runs in the context of
the control's parent composite view (or the Coach if there is no containing composite
view). Use the "me" context variable to refer to the control emitting the event and the
"view" context variable to refer to the parent composite view.

Special case for "cosmetic" controls

If the Coach or Coach View composition is arranged like a tree with parents, and siblings
(and descendants), the addressing scheme could easily break by moving controls around
to different levels of the tree.

For example, after having created logic between a Button and a Text control (Id: Text1),
the reference to "Text1" control would break if it were moved in a Panel control (Id:
Panel1) for cosmetic reasons and the reference would need to be changed from ${Text1}
to ${Panel1/Text1}.

To prevent such problems, most container controls in SPARK whose purpose is to group
or arrange or wrap around other controls (for example panels, layouts, tab section, wells,
and input groups) are purposely not considered in the addressing hierarchy. This flattens
the view hierarchy (more or less depending on how many cosmetic-only controls are
present in a UI and how much they are nested) and solves the problem of potential
reference breakage from visual control rearrangements.

Figure 16 shows the Coach View tree from Figure 15 when adjusted to account for
SPARK's handling of "cosmetic" Coach Views:
Table 3 provides the list of controls that are considered "cosmetic" for SPARK UI addressing:

**Table 3 List of "cosmetic" SPARK controls (not inserted in a control's addressing path)**

<table>
<thead>
<tr>
<th>Control</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caption Box</td>
<td>Caption Box is a container that wraps around a control to add a label or caption around it</td>
</tr>
<tr>
<td>Collapsible Panel</td>
<td></td>
</tr>
<tr>
<td>Deferred Section</td>
<td></td>
</tr>
<tr>
<td>Horizontal Layout</td>
<td><em>Unless bound to a list - which makes it a repeating container</em></td>
</tr>
<tr>
<td>Input Group</td>
<td>Input group is a container that wraps around a control to add a button or icon to the left or right of the control</td>
</tr>
<tr>
<td>Modal Section</td>
<td></td>
</tr>
<tr>
<td>Page Layout Row</td>
<td>Deprecated after IBM BPM 8.5.0.1</td>
</tr>
<tr>
<td>Page Layout Column</td>
<td>Deprecated after IBM BPM 8.5.0.1</td>
</tr>
<tr>
<td>Page Layout Cell</td>
<td>Deprecated after IBM BPM 8.5.0.1</td>
</tr>
<tr>
<td>Panel</td>
<td></td>
</tr>
<tr>
<td>Panel Header</td>
<td></td>
</tr>
<tr>
<td>Panel Footer</td>
<td></td>
</tr>
<tr>
<td>Popup Menu</td>
<td>Popup menu is a container that wraps around a control to add a popup menu to it</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack is like a tab section without any tabs or decorations around it. It only shows one pane at a time</td>
</tr>
<tr>
<td>Status Box</td>
<td>Status Box is a container that wraps around a control to add a status message bubble under it</td>
</tr>
<tr>
<td>Tab Section</td>
<td></td>
</tr>
<tr>
<td>Table Layout</td>
<td>Deprecated after IBM BPM 8.5.6</td>
</tr>
<tr>
<td>Table Layout Row</td>
<td>Deprecated after IBM BPM 8.5.6</td>
</tr>
<tr>
<td>Table Layout Cell</td>
<td>Deprecated after IBM BPM 8.5.6</td>
</tr>
<tr>
<td>Tooltip</td>
<td>Tooltip is a container that wraps around a control to add a tooltip to it</td>
</tr>
<tr>
<td>Vertical Layout</td>
<td><em>Unless bound to a list - which makes it a repeating container</em></td>
</tr>
</tbody>
</table>
The control id of Horizontal and Vertical Layout containers becomes part of the address of the child controls they contain if the layouts are bound to a list (meaning if they are configured as repeating containers).

### Accessing controls in repeating containers

The Table control in the SPARK UI Toolkit is an example of a repeating control. At design time, only the first row is visually modeled. For example, a Table control (Id: Table1) might contain a Text control (Id: Text1) and a Button control (Id: Button1).

In the most use cases, there are three consistent patterns that control referencing needs to support (the examples here refer to a Table control, but control addressing should behave similarly for any repeating container):

1. From outside of a table, a control needs to access a control that is in a table, by control Id and by row index
2. From inside of a table, a control in a table column needs to access a control in another column but in the same row
3. From inside of a table a control in a row needs to update a control outside of the table

Figure 17 shows the inline logic of an on-click event to update the Text control in the fourth row of the table:

[Figure 17 Event referring to nth repeating Text control in table]
This second example in Figure 18 shows how a control can easily access another one in the same row:

Figure 18 Event referring to Text control in table in same row as control emitting event

The runtime behavior of the two previous examples is shown in Figure 19:
Figure 19 Control addressing and events with repeating table - runtime behavior

Note: Achieving the same behavior, as shown in the above example, without SPARK addressing (using classic Coach Framework capabilities) would require a much more complicated and cumbersome approach (using "wrapper" composite coach views).

The third example in Figure 20 shows how a control in a table could refer to another control outside the table using relative addressing:
Figure 20 Event referring to Output Text control outside of table

Figure 21 shows the runtime result for the third example:

Text changed to "2" when button on row at index 2 was clicked

On Button2 click:
$\{ ../Output_Text1 \}.setText( me.ui.getIndex() )$

Figure 21 Control addressing from inside to outside of repeating table - runtime behavior

Tip: In the event handler of a repeating control, `me.ui.getIndex()` provides an easy way to determine the row or list index of the control whose event fired.
Tip: Using relative addressing is a far better practice than using absolute addressing, especially within a composite Coach View. Absolute addressing often requires the logic of a composite Coach View to know details about the naming and structure of its parent that could be altered by other UI developers and break absolute references.

More on tables and repeating containers is covered later in this chapter (see "Working tabular and repeating data") - the purpose of this particular sub-section was only to deal with the topic of typical addressing usage with repeating content.

4.4.3 Optional data binding

Data binding to controls is a valuable and convenient capability which allows controls bound to data either through data binding or through configuration options to automatically synchronize their state based on the bound business data and vice-versa. This capability is provided by the Coach Framework out of the box. The vast majority SPARK UI controls also support this capability.

Most SPARK UI controls however, work almost identically whether they are bound to business data or not. They do not need to be bound to data to be highly functional. This allows the UI developer to only use business data structures for business data purposes. There is virtually no need for data holder structures (Business Objects) whose primary purpose is often to just back UI control states, even when those controls don't represent a meaningful business or process data concept. This can significantly reduce solution clutter and allows reusable Coach Views to have less external dependencies to function properly.

When a SPARK UI control is not bound to data, all its methods work as if it were bound to data, but (obviously) no data synchronization occurs. In either case all SPARK UI controls support the methods shown in Table 4:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getData()</td>
<td>Same as context.binding.get(&quot;value&quot;)</td>
</tr>
<tr>
<td>setData(val)</td>
<td>Same as context.binding.set(&quot;value&quot;, val)</td>
</tr>
<tr>
<td>isBound()</td>
<td>Indicates if the control has a real binding from IBM BPM or a synthetic binding from SPARK</td>
</tr>
</tbody>
</table>

Note on aliases: Many controls provide more intuitive aliases to getData or setData, for example getText or setText for the Text control, getDate or setDate for the Date Picker control, and so on. Regardless, getData and setData always work on controls.

Tip: Only use data bindings to update or react to changes in legitimate business data that is part of the Human Service.
Data change events without a data binding

Many SPARK controls provide support for a change event. The change event works the same way, even if the control is not bound to data - as shown in Figure 22:

Figure 22 Data change events for bound or unbound controls

4.4.4 Validation

All SPARK UI control types that support validation visually reflect an invalid state. However, SPARK controls that support validation do not strictly depend on the conventional Coach Framework-based validation approach described in Chapter 2 (though they fully behave as expected with conventional validation).

The SPARK UI Toolkit and some of its controls provide support for combinations of the following types of validation behaviors:

1. Incorrect input is prevented in the first place. In this case, the control is never in an invalid state
2. A control allows invalid input but flags the issue while typing
3. A control allows invalid input but flags the issue after it loses focus
4. Some controls contain invalid input, but submission is prevented until errors are resolved
5. Some controls contain invalid input, and issues are flagged after submission

Client-side validation is well-suited for cases 1, 2, 3 and 4 above (assuming the validation needed doesn't rely on server-side rules). Server-side validation (including AJAX-based validation) can also work well for cases 3, 4, and 5.
Client-side validation

Client-side validation means that logic can be triggered to examine the content of one or more controls as content changes, then validation errors can be flagged on one or more controls, or in the composite view containing the controls, or on the entire Coach.

The simplest kind of client side validation (example shown in Figure 23) can give user feedback as content is typed or as the control loses focus:

Figure 23 . Validation error reported on a Text control

Figure 24 shows how this kind of behavior can be modeled with a regular expression on the Text control:

Figure 24 Example of validation through regular expression on Text control

Note: In the context of text validation, a regular expression is a character pattern that defines the allowable format, characters, numbers, or symbols that can be entered in the field.

Other controls such as Masked Text provide build-in capabilities to restrict typed content through input "masks" - as shown in Figure 25:

More information on regular expressions can be accessed through this link: https://en.wikipedia.org/wiki/Regular_expression
Programmatic validation

Although configuration-time validation can be very useful, the most flexible type of validation is programmatic. Table 5 lists the four methods provided by the extended SPARK programming model provides to assist with client-side validation behavior:

Table 5 Client-side validation-related methods for SPARK

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;view&gt;.setValid(flag, errorMessage)</code></td>
<td>Sets the client-side validation state of a control. If not valid, the error to be shown in the validation error tooltip can be specified</td>
</tr>
<tr>
<td><code>&lt;view&gt;.isValid()</code></td>
<td>Queries if a control is in a valid state</td>
</tr>
<tr>
<td><code>bpmext.ui.getInvalidViews(fromView)</code></td>
<td>Retrieves a list of invalid views (optionally under a particular view). If there are no validation errors, the method returns an empty list or array.</td>
</tr>
<tr>
<td><code>bpmext.ui.getRequiredViews(onlyEmpty, fromView)</code></td>
<td>Retrieves a list of required views (optionally under a particular view). The onlyEmpty flag filters out views whose (binding) data is set. If no matching views are found, the method returns an empty list or array.</td>
</tr>
</tbody>
</table>
The `setValid()` method can be used to set or unset the valid state on any view that supports the concept of a "valid" state. In Figure 26, logic runs when the Decimal control value changes and shows an error if the validation test fails:

![Decimal control](image)

**Figure 26 Using the setValid() method for client-side programmatic validation**

**Note:** The decision for where validation logic should run (on the client or on the server) should be carefully evaluated and made by the solution architect. The SPARK UI toolkit provides streamlined support for both scenarios.

The next validation scenario is identical to the previous except that the validation logic runs in an AJAX service. The AJAX service invocation is done through the "Service Call" SPARK control.

Instead of calling the validation logic locally, the "Amount" Decimal's "on change" event invokes the "Validation" Service Call control, which in turn invokes the server-side validation logic. Once the AJAX service returns, the "on result" or "on error" event of the Service Call control is asynchronously triggered and sets the "Amount" Decimal's validation state accordingly - as per the Figure 27 interaction sequence:
Figure 27 Using the setValid() method along with Service Call control and AJAX-based validation

**Sequence recap:**

1. Data changes in the "Amount" Decimal control, which triggers the On Change event
2. The Decimal On Change handler calls execute() on the "Validation" Service Call control
3. The "Validation" Service Call control invokes its associated AJAX service

```javascript
if (tv.local.data < 100) {
    tv.local.error = new tw.object.AjaxError();
    tv.local.error.errorText = "Value must be 100 or higher";
    tv.local.error.errorCode = "00001E";
}
```
4. The associated AJAX service executes the validation logic (server-side obviously) and returns an error if the amount < 100
5. The Service Call control's event handlers call setValid(true|false) on the "Amount" Decimal control

This scenario highlights another key benefit of the SPARK UI Toolkit: Service Call-based invocations don't rely on a Human Service diagram. This means that both client-side logic and server-side invocations can be fully encapsulated inside a view which takes reusability a step further compared to a classic Coach Framework-based scenario.

Lastly, the scenario in Figure 28 explains how to use the bpmext.ui.getInvalidViews() method to easily prevent navigation if any control on a page or in its embedded views is in an invalid state:
Figure 28 Using the bpmext.ui.getInvalidViews() method to inhibit Coach navigation

**Note:** The same principle as above can be applied with bpmext.ui.getRequiredViews() to prevent navigation if not all required views have been filled.
4.5 UI layout

The ability to configure layout behavior to control vertical or horizontal flowing, vertical and horizontal alignment of content - including justification, auto reflowing or wrapping of content is essential to providing a pleasant and naturally flowing user experience.

The ability for that behavior to adapt to different form factors is equally important and is covered in the "Responsiveness" section of this chapter.

As of IBM BPM 8.5.7, Coach content can be laid out using a grid that is configurable at design-time and controls layout behavior for Coaches built in the IBM Process Designer Web Editor (as described in Chapters 2 and 3).

**Tip:** Because Grid responsive behavior is always based on the width of the Coach, grids are usually best to use in a Coach or for Coach Views that are always expected to take up the entire width of the Coach.

SPARK UI layout controls (covered in this section) provide sophisticated and consistent layout support for both Web and Desktop Editors in IBM Process Designer and work well for Coaches and Coach Views, from IBM BPM 8.5.0.1 through IBM BPM 8.5.7.

**Tip:** The responsive behavior of SPARK UI layout controls is not relative to the Coach Page width. This makes SPARK UI Layouts a good choice to lay out controls in a composite Coach View (meaning a view that contains other controls).

4.5.1 Horizontal and vertical layouts

The core layout support provided in the SPARK UI Toolkit is contained in the two Horizontal Layout and Vertical Layout controls. The choice between horizontal and vertical layout is really a design-time consideration since it is fully overridable at runtime (especially for responsive behavior, which is covered later in this chapter). In fact, both Horizontal Layout and Vertical Layout controls are backed by the same logic - the only difference being how IBM Process Designer displays the container controls at design time (using a horizontal or vertical layout).

Figure 29 lists important layout and alignment configuration parameters available for the Horizontal and Vertical Layout controls:
Horizontal layout flow and options

When using horizontal layout, all controls directly contained in the Layout control are displayed next to each other horizontally, as shown in Figure 30:

By default, a margin is included around the layout (whether horizontal or vertical). Often however (especially when a layout is nested inside of another), the UI developer may not want a new margin added with each new level of nesting. The "Horizontal Tight" layout option (shown above) removes the margin to allow for a more esthetically pleasing layout appearance in such cases.
Left, right, center horizontal alignment

Figure 31 shows how layouts configured in a horizontal layout mode can align their controls left, right, or center. Additional horizontal layout configuration flavors also allow content to scroll horizontally or auto-wrap:

- **Layout Flow: Horizontal Inline Scroll**

- **Layout Flow: Horizontal Auto-wrap**

Figure 31 Horizontal layout modes with "Left" horizontal alignment

For inline scroll, content will appear properly aligned until it is compressed enough to cause horizontal scrolling. For auto-wrapping, alignment also works predictably and the wrapped content also aligns correctly (left, right or center).

**Note:** When using the "Horizontal Inline Scroll" layout option, controls that vertically expand (such as a Popup Menu, Tooltip, Status Box) should be used with caution as the expanded content may appear vertically cut off (the scrolling portion of the browser clips vertically overflowing content).

Justified alignment

Justified alignment is the default alignment mode unless otherwise configured (through the "Horizontal Alignment" option). The contained controls in a Justified-aligned layout don't wrap overflowing content or scroll to show the overflow.

Instead, the width of the child controls can become elastic so that the entire row of controls takes up the width of the layout (like table cells in a row). Contained controls can have a width configured - in % or unit-based width (for example 120px, 2.9em). The browser auto-adjusts the width of any contained control that doesn't have a width configuration option specified.

Once the width of controls contained in a layout can be compressed no further, Figure 32 shows how the controls overflow (unless responsiveness settings override this behavior):
Figure 32 Horizontal overflow and "Justified" horizontal alignment

**Note:** "Justified" is not a valid horizontal alignment option for the Inline Scroll and Auto-wrap layout options.

- **Vertical layout flow and options**

  When using vertical layout, all controls directly contained in the Layout control are stacked vertically (meaning they flow top to bottom) - as shown in Figure 33:

  ![](image)

  **Figure 33 . Vertical layout mode with "Justified" horizontal alignment**

  The Left, Center, and Right horizontal alignment options also apply for vertically flowing layout content and the stacked control horizontal align as shown in the center-aligned example in Figure 34:

  ![](image)

  **Figure 34 Vertical layout mode with "Center" horizontal alignment**

- **Width and Height**

  By default - unless the Width configuration option is set - a layout control takes up the entire width of its container (whether in a Coach View or a Coach). As seen previously, a Layout control with a "Horizontal Inline Scroll" layout flow will scroll its child content as soon as the combined (compressed) widths of all child controls exceeds the configured width of the Layout control.

  In general, the widths of immediate child controls (contained in the layout) are configured by setting the Width option on each child control. Child control widths do not need to be set unless the browser's layout behavior needs to be overridden. In the case of *Justified* alignment (covered previously in "Justified alignment"), child controls act like table cells (with one cell for Vertical layout and multiple cells for Horizontal layout). Widths of justified child controls behave like the widths of cells in an HTML table.
Note: Width layout behavior with the SPARK UI toolkit does not use the 12 columns layout model (except for Page Layout controls, which are deprecated after IBM BPM 8.5.0.1). SPARK allows any width specifications and any number of columns. (To use a 12-column model with responsiveness solely based on Coach width, use the Grid in IBM Process Designer’s Web Editor, otherwise use a SPARK UI Layout control).

Unless the Height of a layout is configured, the Layout control expands vertically to accommodate showing its child content. If Height is configured and the content vertically exceeds the height configured, then the layout automatically allows vertical scrolling - as illustrated in Figure 35:

![Figure 35 Vertically scrolling layout](image)

Setting the Height of a Layout control works predictably with any unit (for example 150px, 6.2em). However heights set using % only work if all parent elements (or layouts or views) of the Layout control have a height specified - until either a height is specified in units (not using %), or go all the way to the Coach body element with either % or unit-based height specification. This is a limitation of specifying heights using CSS-based styling.

- **Vertical alignment**

  The ability to configure a Layout control's Vertical Alignment option matters in two situations:
  1. If the height of the Layout is configured and is greater than the vertical space taken by its content (whether the Layout flows horizontally or vertically)
  2. The height of child content flowing horizontally is uneven

  Figure 36 below shows situation #1:
Figure 36 Layout height greater than height of contained controls

Figure 37 illustrates situation #2:

Nesting layouts

Horizontal or vertical layouts controls can (and should) be nested to achieve a variety of layout behaviors. Layouts can contain other nested layouts with their own Layout Flow, Horizontal Alignment, Vertical Alignment, Width, and Height configuration options.

These capabilities (especially when combined with responsive behavior support - covered later) provides very powerful and flexible options for BPM UI layout at the Coach or Coach View level.

The "Tight" (Vertical or Horizontal) Layout Flow options are most useful in nested situations where certain parts of the child layout need to align with the edges of the parent layout. Figure 38 below shows the subtle difference of using a nested layout with and without the Tight option:
4.5.2 Tabbing or stacking UI content

The SPARK UI Toolkit provides a tab control (showing one pane at a time, obviously) - and a Stack control for arranging panes of content that only display one at a time.

- **Tab Section**

An example of design-time and runtime behavior and appearance of the Tab Section control is given in Figure 39:
The Tab Section control provides a "Tab Changed" event which gets fired anytime the current tab is changed (either by the user or programmatically). The binding data (if the control is bound) contains the 0-based integer index of the tab currently showing. The UI developer can also change the current tab programmatically using the setCurrenPane() method.

**Note:** A "0-based" tab index means that the first tab is at index 0, the second at index 1, and so on.

**Stack**

The Stack control is similar in behavior to a Tab Section control but without any visual decorations (no visible tabs or borders).

### 4.5.3 Displaying modal content

When content needs to be displayed modally (meaning in such a way as to prevent input in other parts of the UI), the Modal Section provides a flexible option to do so for any control (for example a Panel - for modal dialog-like behavior, a Well containing a Progress Bar control).
**Note:** A Well is a simple "cosmetic" container that can be colored or left blank and given an optional background icon. See "Wells" for more details.

The Modal Section simply acts as a container. Whatever the control the UI developer inserts in the containment box at design time is shown modally (against a darker backdrop) at runtime, as shown in Figure 40 below containing a Well and a Progress Bar control:

![Figure 40 Modal Section displayed at runtime with a Progress Bar](image)

Figure 41 below shows the same Modal Section containing a Well and a Progress Bar in IBM Process Designer’s Web Editor:

![Figure 41 Modal Section in IBM Process Designer Web Editor](image)
Showing and hiding modal sections

The display of modal sections is based on the standard Coach Framework visibility setting (and can also be controlled by the setVisible() SPARK UI method). For this reason, a Modal Section control's visibility setting must be set to NONE initially, otherwise the section will be displayed as the Coach opens. The visibility of the "Modal Alert" control works the same way.

Once displayed, the Modal Section can be closed in two ways:
By calling setVisible(false) on the Modal Section control
By clicking in the darkened area of the section (assuming the "Close On Click" configuration option has been set at design time, as illustrated in Figure 42 below)

![Figure 42 Close on Click configuration option for Modal Section](image)

Lastly, the "On Close" event on a Modal Section control is available to take programmatic actions when closing and fires when the Modal Section closes from user action or programmatically (from `<Modal Section control ref>.setVisible(false)`).

Displaying dialogs modally

Modal sections can be used equally easily to display modal dialogs. Figure 43 below shows a Panel displayed as a modal dialog (meaning in a Modal Section) and the simple logic to open and close it.
Figure 43 Panel control displayed in a Modal Section as a modal dialog

4.5.4 Wells

Wells provide a simple way to display content in an area that can be colored and can include an icon in its background. The purpose of Wells is mostly limited to making UIs more visually appealing and organized (by separating parts of the UI from their...
background using different coloring). Wells are as appropriate to use by themselves or inside Layout controls or a Grid.

The illustration in Figure 44 below shows a Well configured to display a bank icon in its background (and using the INFO color style) containing a Donut Chart control:

Figure 44 Configuration and runtime behavior of Well control containing a Chart

4.6 Calling AJAX services

AJAX services under the Coach Framework are often called (directly or indirectly) from the diagram of a Human Service. This option is simple to use but provides low reusability as it doesn't allow the service invocation to be encapsulated inside a Coach View.
Another classic option - for Coach Views only - is to create a configuration option of type “Service”, then to call the service through

\texttt{context.options.<serviceName>(serviceArgs)}^{4}.

But again, this can expose as a configuration option a detail about the Coach View which, in some cases, should remain internal to (meaning encapsulated in) the view. It is also a fairly technical exercise.

The SPARK UI Toolkit includes a control that offers streamlined and encapsulated interaction capabilities which work the same way in Coaches and in Coach Views, and in a manner that is consistent with how SPARK events work.

\subsection*{4.6.1 Service Call control}

The Service Call control can be found in the SPARK UI Toolkit palette (shown in Figure 45) under the “System**” category:

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{Service_Call_Control}
\caption{Control palette showing Service Call control}
\end{figure}

\textbf{Configuration}

The Service Call control is configured and works as follows:

A Service Call control must always be associated with a IBM BPM AJAX service - as shown in the configuration example of Figure 46 below:

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{Service_Call_Config}
\caption{AJAX Service association with a Service Call control}
\end{figure}

---

\footnote{See an example of calling an AJAX service in the classic Coach Framework manner at: http://www.ibm.com/support/knowledgecenter/SSFPJS_8.5.7/com.ibm.wbpm.wle.editor.doc/topics/tajaxservice.html}
The service call control provides the `execute()` method to trigger the invocation of the associated AJAX service. `execute()` accepts a single input which can be a simple or complex type. For example:

```javascript
${Service_Call1}.execute("abc")
```

...to specify a string as input, or:

```javascript
${Service_Call1}.execute({"prop1": "value1", "prop2": 2})
```

...to specify an object with a `prop1` string property and a `prop2` integer property

If the single parameter for `execute()` is omitted, then the value associated with the "Input Value" configuration option is used.

The control method `setInputData()` provides a programmer-friendly way of updating the value of the "Input Value" configuration option.

If the "Auto Run" configuration option is checked, then the Service Call control calls `execute()` automatically every time the value associated with the "Input Value" configuration option changes (and also when the control is first loaded).

The Service Call control can show a visual progress indicator (see Figure 47) which is displayed while the AJAX invocation is in progress.

![Service Call busy indicator configuration](image)

### Invocation sequence

Figure 48 below explains the invocation sequence needed for the Single Select control containing shipping modes to use the Service Call control to trigger a shipping cost query which in turn sets the retrieved cost in a Decimal control:
Figure 48 Usage example for the Service Call control

**Sequence recap:**

1. Selected item changes in the Single Select control, which triggers the On Change event
2. The Single Select On Change handler calls execute() on the "ShippingRateQuery" Service Call control
3. The "ShippingRateQuery" Service Call control invokes its associated AJAX service
4. The associated AJAX service executes its server side logic and returns the shipping cost (or an error for standard shipping mode)
5. The Service Call control's event handlers call setValue(<returned AJAX result>) on the "Shipping Rate" Decimal control
The resulting behavior at runtime - given the sequence, logic and configuration above - is as follows for a successful invocation (Figure 49):

![Figure 49 Runtime example for Service Call control - successful invocation](image)

Figure 50 shows a failed invocation:

![Figure 50 Runtime example for Service Call control - AJAX service error](image)

- Handling server logic errors

AJAX service logic error information is accessible from the Service Call control through the getLastError() method which contains an error object with two properties:

- `errorText`
- `errorCode`
When the "On Error" Service Call event fires, using getLastError().errorText and getLastError().errorCode can be used to retrieve the message from the AJAX service logic that is associated with the error.

- Reporting errors from AJAX service logic

Errors in AJAX service logic can be reported in three ways inside the AJAX Service associated with the Service Call control:

1. By initializing `tw.local.error` to `new tw.object.AjaxError()` and setting the `errorText` and `errorCode` property of the error. With this approach, an error event or exception should *not* be thrown. When the AJAX invocation returns, the Service Call control automatically activates its "On Error" event.

2. By using an Error end event. The error code and the data from the error mapping on the Error Event map to `errorCode` and `errorText`.

3. By throwing a server-side JavaScript error - for example `throw new Error("Invalid account number")`. The exception Java text (including the location of the offending statement) is mapped to the `errorText`. The value of `errorCode` in this case is determined by IBM BPM, not by the service developer.

4.7 Responsiveness

Responsiveness in the SPARK UI Toolkit works on two levels:

- **Classic (IBM BPM-based) responsiveness:** The basic responsive behavior exposed in the IBM Process Designer Web Editor for controls that provide adaptive properties.

- **Enhanced Web Component-based responsiveness:** Using the SPARK responsive capabilities that behave consistently since IBM BPM 8.5.0.1, work with both Web and Desktop Editors of IBM Process Designer, and allow greater flexibility than Coach form factor-based responsive triggers.

4.7.1 Classic responsiveness

Most SPARK UI Toolkit controls provide adaptive configuration options that can be exploited through IBM Process Designer's Web Editor.

The illustration provided in Figure 51 shows an example of the Note control "Label Style" adaptive property, which allows the IBM Process Designer Web Editor to specify different configuration option values across the three IBM BPM-supported screen or form factors (Small, Medium, and Large):
At runtime, the control auto-adjusts the configuration option value of its adaptive property based on the Coach Framework's screen factor-based trigger, as shown in the Figure 52 runtime example for the previously-configured Note control:
Using this approach (which only works with IBM BPM 8.5.5 and higher and requires using IBM Process Designer’s Web Editor) a large number of SPARK UI controls provide adaptive properties that can be configured, as appropriate, to provide this type of behavior.

4.7.2 Enhanced Web Component-based responsiveness

By combining layouts and the Responsive Sensor control (discussed in this section) the SPARK UI Toolkit provides the following added benefits in addition to classic responsiveness support in IBM BPM:

Responsive behavior is provided and works consistently across product versions from IBM BPM 8.5.0.1

Coach Views can predictably determine their responsive behavior even when they don't take up the entire width of the Coach in which they are placed.

Coach Views with SPARK behave like "Web Components" and as such do not depend on the overall Coach form factor (from the IBM Process Designer Web Editor) to trigger layout changes.

Layout changes can instead be triggered by the size change of the Coach View itself, independently from the Coach that contains it.

SPARK UI responsiveness principally relies on three controls:
Respnsive Sensor
Horizontal Layout
Vertical Layout
Responsive Sensor control

The Responsive Sensor is a container control. It wraps itself around its contained child controls (usually a Horizontal or Vertical Layout) and acts as a size-sensing "rubber band". It is configured with "box factors" that help determine when the sensor activates responsive behavior in its contained Layout control(s).

An example of the Responsive Sensor control at design time is shown in Figure 53:

![Responsive Sensor control/container](image)

Responsive Sensor control/container

Control ID: Sensor1

Horizontal Layout control/container

Box factors:
- Narrow: 0 up to 480px
- Wide: > 480px

Figure 53 Response Sensor and box factor configuration

Once box factors are specified, a Layout control (Horizontal in the previous example) inside the Responsive Sensor can specify layout-related behavior based on one or more of the box factors configured. In the example below, the Layout is configured with the following options by default (settings in Appearance category):

- Layout Flow: Horizontal
- Horizontal Alignment: Justified

But the settings in the Responsive category override the default configuration as follows:

If Sensor1’s box factor name is "Narrow":

- Layout Flow: Vertical (option labeled as "childLayout")

Figure 54 provides the full design-time configuration:
Responsive Sensor control/container
Control ID: Sensor1

Horizontal Layout control/container

Default Layout configuration. These options are used if no box factor is matched

Use “Sensor1” as box factor sensor (optional setting – uses nearest parent sensor if blank)

If Narrow box factor is active, display child content Vertical, otherwise, use configured defaults (horizontal layout, justified alignment, etc…)

Figure 54 Responsive behavior overriding default layout configuration

Note: The name of the responsive sensor to be used by the Layout control (see Responsive > Responsive Sensor configuration option) is not a mandatory setting. If it is not provided, the Layout control listens to its nearest Responsive Sensor parent.

Box factor change event

At times it may be useful to take programmatic action whenever a box factor is activated. The Responsive Sensor control provides a special event (Events > On Responsive Boundary) that fires every time a box factor changes (including the first time a box factor is activated).

The Events > On Responsive Boundary event handler provides an “event” context variable that contains at runtime the properties shown in Table 6:
### Table 6 Responsive Sensor's On Responsive Boundary event handler context

<table>
<thead>
<tr>
<th>event context variable property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boxFactor</td>
<td>The name of the Responsive Sensor box factor that became active</td>
</tr>
<tr>
<td>lowerBound</td>
<td>The lower bound in pixels for the current box factor</td>
</tr>
<tr>
<td>upperBound</td>
<td>The upper bound in pixels for the current box factor</td>
</tr>
<tr>
<td>width</td>
<td>The current width (in pixels) that triggered the box factor change</td>
</tr>
</tbody>
</table>

The code in Example 6 (in the "On Responsive Boundary" event handler of a Responsive Sensor control) logs the current box factor name to the web browser console when a responsive boundary has been crossed:

```javascript
console.log("Active box factor: " + event.boxFactor)
```

**Example 6 On Responsive Boundary event handler code sample**

**Reminder:** The On Responsive Boundary event doesn't fire every time a resize occurs. It only fires when a new box factor becomes active.

- **Layout controls - responsive settings**

  When a box factor is active and a (Horizontal or Vertical) Layout's responsive behaviors option uses that box factor, then any "Behaviors" configuration setting corresponding to the box factor entry is applied. See configuration setting example in Figure 55:
Figure 55 Layout configuration with responsive settings

**Note:** The example above assumes a Responsive Sensor contains the Layout and the Responsive Sensor is at least configured with box factors named "Medium" and "Narrow".

Table 7 below details the meaning of the Behaviors configuration settings for a Layout control:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boxFactorName</td>
<td>The name of the Responsive Sensor box factor that activates the configuration for the row of settings</td>
</tr>
<tr>
<td>childLayout</td>
<td>Same as Appearance &gt; Layout Flow: The Layout Flow applied to the child content of the Layout control (Horizontal, Horizontal Inline Scroll, Horizontal Tight, Horizontal Auto-wrap, Vertical, Vertical Tight)</td>
</tr>
<tr>
<td>childAlign</td>
<td>Same as Appearance &gt; Horizontal Alignment: The Horizontal Alignment applied to the child content of the Layout control (Justified, Left, Center, Right)</td>
</tr>
</tbody>
</table>
4.7.3 Coach View responsiveness

The purpose of this section is to help the BPM UI developer decide when to use the Responsive Sensor in Coach Views instead of relying on the built-in Web Editor Grid in IBM Process Designer or using adaptive properties based on form (Coach) factors.

- Responsive behavior using built-in IBM BPM capabilities

The example in Figure 56 below shows two composite Coach Views ("Contact Information" and "Billing Information") stacked vertically on a Coach. The responsive behavior of the views is based on adaptive properties that adjust the arrangement of input controls as the form factor changes from Medium to Small:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>childWidth</td>
<td>Width(s) of child controls contained in this Layout.</td>
</tr>
<tr>
<td></td>
<td>If 1 width is specified (for example 100%), all children controls have that</td>
</tr>
<tr>
<td></td>
<td>width. If 2 widths are specified (for example 64% 33%) the 1st child has</td>
</tr>
<tr>
<td></td>
<td>the 1st width, the 2nd child has the 2nd width, the 3rd child has the 1st</td>
</tr>
<tr>
<td></td>
<td>width, the 4th child has the 2nd width, and so on.</td>
</tr>
<tr>
<td></td>
<td>As many widths as needed can be specified - they are applied as per the</td>
</tr>
<tr>
<td></td>
<td>algorithm described above. They must be separated by a space. Units can be</td>
</tr>
<tr>
<td></td>
<td>for example %, px, em.</td>
</tr>
<tr>
<td>width</td>
<td>The Width of the Layout control (relative to its container if using %)</td>
</tr>
<tr>
<td>height</td>
<td>The Height of the Layout control (relative to its container if using %)</td>
</tr>
<tr>
<td>cssStyle</td>
<td>CSS settings to be applied to the &lt;div&gt; element representing the</td>
</tr>
<tr>
<td></td>
<td>Layout control (same div as the context.element div reference from the</td>
</tr>
<tr>
<td></td>
<td>Coach Framework API)</td>
</tr>
<tr>
<td>cssClass</td>
<td>CSS class to be added to the &lt;div&gt; element representing the Layout</td>
</tr>
<tr>
<td></td>
<td>control (same div as the context.element div reference in the Coach</td>
</tr>
<tr>
<td></td>
<td>Framework API)</td>
</tr>
</tbody>
</table>
This use case works equally well using the IBM Process Designer Web Editor Grid, the layout or alignment-related adaptive properties of the SPARK Layout controls, or the Responsive Sensor.

However, when a UI developer expects that a composite Coach View may be placed on a Coach UI in such a way that it will not take up the entire width of the page, the Responsive Sensor control (used inside the Coach View) is the best option to predictably control the Coach View content's layout behavior.

Without a Responsive Sensor, when the two coach views are laid out horizontally (side by side), the layout cannot accurately determine when to reflow. This is because, since each Coach View in this case takes up 50% of the Coach's horizontal space, a Coach View will be twice as narrow by the time the Coach-driven form factor takes effect - as shown in Figure 57 below:

---

Figure 56 IBM BPM-based responsiveness - correct behavior
Responsive behavior using SPARK UI capabilities

When using a Responsive Sensor, the width of a Coach is not considered (unless the Responsive Sensor is configured as wide as the page, obviously). This means that Coach View responsiveness is truly an aspect that is controlled by and encapsulated in the Coach View. The same example as before is provided for contrast in Figure 58, using a Responsive Sensor and a Horizontal Layout configured with a single Responsive > Behaviors configuration option for a narrow (≤ 420px) form factor.
Figure 58 SPARK UI-based responsiveness
4.8 Working tabular and repeating data

The SPARK UI Toolkit provides several ways of displaying repeating data, whether it comes from IBM BPM (through Human Service Data represented as local variables) or from an AJAX service, and whether it is a list of simple types (such as string, number) or of complex types (defined in IBM Process Designer as Business Objects).

4.8.1 Table and Service Data Table controls

The two tabular data controls provided by the SPARK UI toolkit are the Table control (for working with IBM BPM Human Service data) and the Service Data Table control (meant to display data returned by a backing AJAX service).

Table control

The Table control can be bound to a list of complex objects. It is a high-performance control to display data in cells as Coach Views or using other cell-rendering options. Figure 59 below provides key feature and capability highlights for the SPARK UI Table:

![Table control options](image)

**Figure 59 High-level Table control options (asterisk denotes optionally enabled features)**

To configure the columns in a Table, various controls must be added to the columns of the table (dragged & dropped from the palette) and - when needed - the individual controls for each column can be bound to `<list>.currentItem.<property>` as needed and as shown in the example from Figure 60:
Figure 60 Binding table and column controls to BPM data

**Note:** Tables can only be bound to lists of complex types. For lists of simple types (such as strings, numbers, and so on) use a Layout control instead of a Table. Horizontal and Vertical Layouts can be bound to lists of simple types and thus become repeating containers. This is covered later in the "Layouts and repeating data" section of this chapter.

- **Column configuration**

  By default, a table with no column configuration renders all columns containing a control as a Coach View. Other rendering options (shown in Figure 61) are available from the "Columns" table configuration option:
To fully configure columns in a table, a configuration entry must be explicitly added for each column mapping to a control in the table. The first control corresponds to column configuration entry 0, the second to entry 1, and so on.

When displaying in a table, the borders on an "input" type Coach View control such as Text, Integer, Decimal, Date Picker, can be distracting and add unnecessary clutter. The Seamless Coach View option renders the Coach Views but automatically removes the border styling on input controls, as illustrated in this Figure 62 example:
Table with no column configuration: All columns rendered as Coach Views

<table>
<thead>
<tr>
<th>Text</th>
<th>Decimal</th>
<th>Integer</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value 0</td>
<td>$0.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value 1</td>
<td>$0.333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value 2</td>
<td>$0.667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value 3</td>
<td>$1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value 4</td>
<td>$1.333</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Same Table rendering “Text” and “Decimal” columns as Seamless Coach View

Active input control for data entry

Figure 62 Comparison between Coach View and Seamless Coach View rendering options

**Note:** The Table control respects tab index on data entry. A control in the Table that is rendered as a Coach View can be tabbed to naturally without using a mouse click or an explicit finger tap. A currently active input control in the table shows the cursor and is styled with a slight grey background as shown in the previous illustration.

- **Simple HTML column rendering**

  Coach Views are always more expensive to render than a simple HTML Document Object Model (DOM) element as they incur the management and lifecycle overhead of the Coach Framework. When Table data shown in a column can be displayed statically, the UI developer should consider using Simple HTML rendering.

  With Simple HTML rendering, the control dragged & dropped in the Table column at design time *is still necessary* as it gives the column a hint about how the data should be rendered (for example a Checkbox control might hint to the Simple HTML rendering mode that a boolean `true` value should be rendered as a “check”, not as the text “true”).

  At runtime, instead of the table cell containing a live Coach View, it contains very a lightweight HTML representation. For instance, a simple check mark (for a `true` value) would be rendered in a cell in lieu of an actual live Checkbox Coach View. The illustration in Figure 63 below shows how Simple HTML-rendering (for all columns in this particular example) is minimally configured and how it is rendered at runtime:
Figure 63 Basic Simple HTML column rendering for a Table (no format options specified)

The next illustration in Figure 64 (using the exact same example as in the previous figure) shows how data formatting options in Simple HTML rendering can be specified to influence the rendering of cell data:
Figure 64 Basic Simple HTML column rendering for a Table (with data formatting options)

**Note:** Do not omit quotes around the option keywords or they will not load properly!
For example "dateFormat": "MMM yyyy" works but dateFormat: "MMM yyyy" does not.

Table 8 below provides a summary of the supported data formatting options if Simple HTML or Custom rendering is used for a Table column (the column containing the appropriate associated “hint” control):
### Table 8 Simple HTML and Custom format options

<table>
<thead>
<tr>
<th>Hint Control</th>
<th>Possible options</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Date Picker</strong></td>
<td>&quot;datePattern&quot;: &quot;&lt;format spec&gt;&quot;</td>
<td>Quick reference: &lt;br&gt;<strong>yy</strong> or <strong>yyyy</strong> for year (for example 16 or 2016) &lt;br&gt;<strong>MM</strong> or <strong>MMM</strong> or <strong>MMMM</strong> for month (for example 06 or Jun or June) &lt;br&gt;<strong>dd</strong> for date - meaning day - in month (for example 14) &lt;br&gt;<strong>EEE</strong> or <strong>EEEE</strong> for day of week (for example Fri or Friday) &lt;br&gt;<strong>z</strong> for timezone (for example Mountain Daylight Time) &lt;br&gt;<strong>hh</strong> for 0-12 hour in day (for example 08) &lt;br&gt;<strong>a</strong> for AM/PM indicator (for example PM) &lt;br&gt;<strong>HH</strong> for 0-24 hour in day (for example 20) &lt;br&gt;<strong>mm</strong> for minutes (for example 54) &lt;br&gt;<strong>ss</strong> for seconds (for example 42) &lt;br&gt;Refer to <a href="http://salientprocess.com">dojo/date/locale::format()</a> for a complete reference</td>
</tr>
<tr>
<td><strong>Checkbox</strong></td>
<td>&quot;type&quot;: &quot;&lt;check&gt;&quot;&lt;icon&gt;&quot;text&quot;&gt; &lt;br&gt;&quot;trueValue&quot;: &quot;&lt;text or icon name if true&gt;&quot; &lt;br&gt;&quot;falseValue&quot;: &quot;&lt;text or icon name if false&gt;&quot;</td>
<td>If type is &quot;check&quot; no other options are needed. A true value renders as a check while false renders as blank. If type is &quot;icon&quot;, any icon name available in SPARK will work. For example &quot;times&quot;, &quot;check-circle&quot;, &quot;battery-full&quot;, &quot;comment&quot;, and so on.</td>
</tr>
<tr>
<td><strong>Decimal</strong></td>
<td>&quot;decimalPlaces&quot;: &lt;integer&gt; &lt;br&gt;&quot;decimalSeparator&quot;: &quot;&lt;text&gt;&quot; &lt;br&gt;&quot;thousandSeparator&quot;: &quot;&lt;text&gt;&quot; &lt;br&gt;&quot;prefix&quot;: &quot;&lt;text&gt;&quot; &lt;br&gt;&quot;postfix&quot;: &quot;&lt;text&gt;&quot;</td>
<td>A decimal can be formatted as an integer by specifying 0 decimal places</td>
</tr>
<tr>
<td><strong>Integer</strong></td>
<td>&quot;thousandSeparator&quot;: &quot;&lt;text&gt;&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Custom column rendering

The Table control also allows the developer to fully control the rendering of a particular column, using the "Custom" rendering option.

When "Custom" is selected for a column, the "On Custom Cell" event is fired for every cell in the given column. The developer can then alter the style of the table cell or can provide the entire HTML content (meaning a DOM element or structure), or both, to be used for the content of the cell.

The Table > On Custom Cell event provides the "cell" context variable to the event inline logic. The cell variable contains a number of properties and method to facilitate the custom rendering effort. Those are detailed in Table 9 below:

Table 9 Detail of "cell" context variable in the Table > On Custom Cell event

<table>
<thead>
<tr>
<th>Property or method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFormattedValue()</td>
<td>If the purpose of the custom rendering isn’t to format the data itself in a particular way, then this method can be used to spare the developer the effort of formatting data. This method returns the formatted value that Simple HTML rendering would have output (including any configured formatting options).</td>
</tr>
<tr>
<td>setSortValue()</td>
<td>Sometimes the displayed content of a cell is not consistent with how the data should be sorted. For example, the list of values “High”, “Medium”, “Low”, when lexically sorted would be ordered as “High”, “Low”, “Medium”. The setSortValue() method tells the table to disregard the displayed value and use a different - developer-provided - one instead (for example “3”, “2”, “1” for “High”, “Medium”, “Low” respectively)</td>
</tr>
<tr>
<td>colIndex</td>
<td>The index of the column containing the cell to be rendered. This is especially useful when more than one column in the table is configured to be custom-rendered</td>
</tr>
<tr>
<td>td</td>
<td>The HTML DOM “td” element corresponding to the current cell. This allows quick styling of the TD element itself</td>
</tr>
<tr>
<td>row.index</td>
<td>The 0-based index of the current row containing the cell to be rendered</td>
</tr>
<tr>
<td>row.data</td>
<td>The &quot;record&quot; (meaning the list item) associated with the current row containing the cell to be rendered. If the table is bound to a list of business objects with properties &quot;lastName&quot;, &quot;creditScore&quot;, for example, those properties can be retrieved as: cell.row.data.lastName and cell.row.data.creditScore.</td>
</tr>
<tr>
<td>varName</td>
<td>The name of the variable associated with the current cell and column (for example lastName)</td>
</tr>
<tr>
<td>value</td>
<td>The value of the current variable associated with this cell. This is exactly equivalent to using cell.row.data[cell.row.varName], but much more convenient.</td>
</tr>
</tbody>
</table>
The "On Custom Cell" event also provides an additional convenience to the UI developer through its returned result:
If the returned result is an object, a DOM element is assumed and is appended to the TD element representing the table cell.
If the returned object is a string:
  o If "H" is returned the table renders the cell as if it had been configured with the "Simple HTML" rendering option
  o If "V" is returned the table renders the cell as if it had been configured with the "Coach View" rendering option
  o If "S" is returned the table renders the cell as if it had been configured with the "Seamless Coach View" rendering option
  o If any other string value is returned an error is logged to the console

Reminder: In the On Custom Cell event, either a DOM element is returned (which will become the content of the cell) or a single character: "H", or "V", or "S".

The example in Figure 65 shows how the column displaying decimals (first column from the left) can style its content red for value less than 400:
Figure 65 Custom rendering for Table cells - conditionally styling a cell

This next example in Figure 66 changes the custom-rendering logic of the previous example to create an actual DOM element for the cell and includes "in-context" on-click logic to the created links in the table:
Rows in the Table control can be searched either by matching content in a specific column or across all rows. The Table control itself does not provide a search text field out of the box. Instead a simple and flexible method is provided to allow searching in different ways.

The `Table:search()` method allows 4 parameters:

- `columnIndex` (0-based integer): Only match content in a particular column. If null, matches across all columns.
searchText (text): search expression to matched
caseInsensitive (boolean): If true, performs a case insensitive search
regularExpression (boolean): If true, consider searchText as a regular expression

The example in Figure 67 below shows a Text control used for a simple search:

Figure 67 Table-wide text search example with Text search field
Performance options

The Table control provides several features that help preserve the user experience even with large numbers of rows and columns. Three key capabilities allow tables to display large amounts of tabular data:

Rendering options: Displaying columns as Simple HTML (or as Custom-rendered) only costs a small fraction of the processing otherwise needed to create and manage the lifecycle of even the simplest of coach views.

On-demand and cached rows: When paging, a table only creates rows (and associated content and views) at the time they are displayed. This reduces the cost of creating rows to the number of rows displayed on a page. Once a row is created it is cached (even if it is no longer displayed because of paging, sorting, or filtering) to avoid the overhead of recreating it.

Record management: The control methods to add or remove records to the table (appendRecord, appendRecords, removeRecord) are significantly very efficient and do not trigger a complete table reload.

Asynchronous loading: Even the lightest of tables can lock up a browser while rendering an extremely large number of rows (because JavaScript is single-threaded), which can be disruptive to the user experience. The SPARK UI Table provides an asynchronous loading feature that keeps the browser fully responsive regardless of how many rows need to be loaded. The trade-off of course is that the rows take longer to be loaded in the background. Asynchronous loading can even be tuned so that - for example - batches of 5 or 10 rows can load at the same time to accelerate background loading while still preserving the user experience.

Figure 68 below highlights the performance-related features that can be used to manage the processing overhead and the user experience when displaying a large number of rows or columns:
Table events

Table 10 lists the events supported by the Table control:

**Table 10 List and detail of events supported by the Table control**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Custom Cell</td>
<td>Covered in detail in the &quot;Custom column rendering&quot; section</td>
</tr>
</tbody>
</table>
| On Rows Loaded      | This event is especially useful when asynchronous loading is on and the developer can therefore not exactly predict when rows in the table finish loading.  
                       If the table uses paging, this event also fires whenever all rows on a page finish loading. The following context variables are available:  
                       all (boolean): true if all rows in the table are completely loaded (if paging is on "all" is true when all pages have been displayed) |
<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
</table>
| **On Row Selected By User** | When a user selects a row through the selection widget to the left of the table (assuming the Table's selection mode is Single or Multiple) this event is fired and provides the following context variables:  
  
  **row.data** (object): Contains the record or item in the list backing the table at the index corresponding to the selection  
  
  **row.recordIndex** (integer): The 0-based index corresponding to the selection  
  
  **row.tr** (HTML element): The TR element associated with the current selection |
| **On Deleting Record**      | When a user selects the deletion widget to the right of the table (assuming the Table's "Show Delete Button" option is on), this event is fired and provides the following context variable:  
  
  **item** (object): Contains the record or item in the list backing the table at the index corresponding to the row containing the delete button clicked  
  
  If the logic in this event handler explicitly returns false, the record deletion is canceled. |

- **Additional capabilities**

  The Table control provides a number of methods to control selection, record insertion and deletion, paging, sorting, and data retrieval. Check the Table control reference for additional information.

- **Service Data Table control**

  The Service Data table control retrieves data from an AJAX service instead of from Human Service data.

  Because Coach Views are tied to Human Service data, the Service Data Table cannot render cells as Coach Views (only as Simple HTML and Custom). Instead of using a list-based data binding as its data feed, the Service Data Table control includes an AJAX service configuration option as its data source, as shown in Figure 69 below:

---

5 See https://support.salientprocess.com/docs/enterprise/Table.html
Unlike the Table control, the Service Data Table must be configured for each column it needs to display. Column configuration include the mapping of the data returned by the service and the type of rendering to use for the column, as shown in Figure 70:
Querying and re-querying data

Figure 71 suggests how a Service Data Table that needs to re-query its associated AJAX service with the same or with new input data (to refresh its content or to issue a new query) can do so by calling the setQueryData() method (passing in new input data), and then by calling the refresh() method:

If the associated AJAX service accepts a simple data type for input (such as an integer), the setQueryData() can be invoked as shown in Example 7:

Example 7 Invoking Service Data Table:setQueryData() with simple input
Example 8 shows how to pass a complex type as input to the AJAX query:

Example 8 Invoking Service Data Table:setQueryData() with complex input

```javascript
$(Table1).setQueryData({"custName":"Smith", "accType":"C"});
```

Lastly, after setting the query data, the refresh() method must be called to invoke the AJAX service with the new input data - as shown in Figure 72 below:
Figure 7.2 Example of Service Data Table re-query based on text update

- Service Data Table events

Table 11 shows the Service Data Table-specific events that are provided in addition to the events supported by the Table control:
Table 11 List and detail of events supported by the Service Data Table control

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Service Data Loaded</td>
<td>Called when a service invocation successfully returns and after the table displays its rows.</td>
</tr>
<tr>
<td>On Service Data Error</td>
<td>Called when a service invocation failure returns.</td>
</tr>
</tbody>
</table>

Additional capabilities

The Service Data Table control provides special capabilities in addition to the Table control (including the ability to create and populate columns dynamically through the setColumns method). Check the Service Data Table control reference6 for additional information.

4.8.2 Layouts and repeating data

When layouts are bound to a list, the Coach Views contained in the Layout which are bound to the current list item ("currentItem") are automatically repeated at runtime.

In the example of Figure 73 below, the Layout is bound to a list of strings containing user ids:

---

6 See https://support.salientprocess.com/docs/enterprise/ServiceDataTable.html
Layout flow and alignment with repeating content

The Layout options (layout flow, horizontal alignment, and so on, including responsive behaviors) are applied to the repeating content just as they are to any other child content. In Figure 74 below, the same layout as in the previous example displays user ids, but this time using Layout Flow "Horizontal Auto-wrap" and Horizontal Alignment "Right":

Performance options

As with the Table control, Layouts provide asynchronous loading capabilities to display very large lists without disrupting the user experience, through the following configuration options in the Performance configuration category:
Async Loading (same behavior as Table control option)
Async Batch Size (same behavior as Table control option)

**Note:** Contained Coach Views can only rendered as "Coach Views" (no "Seamless Coach Views", "Simple HTML", or "Custom" rendering options are currently available for a Layout).

The (Horizontal/Vertical) Layout control also provides methods to efficiently append and remove list elements (instead of causing the reload of the entire list), namely:
- appendElement(<simple or complex object>)
- removeElement(<0-based index>)

### 4.9 Formulas

Formulas are a simplifying and streamlining feature of the SPARK UI Toolkit. They can significantly reduce the need for creating code to maintain computational relationships between controls on a form.

Formulas provide support for simple expressions that prescribe how the value of a control is the result of an operation involving other controls (for example, the ${Total} Decimal control is the sum of the ${Sales} Decimal control and the ${TaxAmount} Decimal control.

When the value of a control can be computed through a formula, the control provides a special option in the Formula category of its configuration options - as shown in Figure 75 containing a Decimal control's configuration example:
Once the formula is specified, the behavior of the control regarding the computation of its value is "maintenance-free": Any updates to the values of any of the controls that are referenced in the formula causes the expression to recalculate automatically.

When a formula references a control whose value is itself computed through a formula, the cascade of updates occurs efficiently and in the correct order.

**Note:** As is usually the case for SPARK controls, data binding for controls is optional even when using formulas. If a control computed through formula has a data binding, the bound value is automatically updated every time the formula result changes.

### 4.9.1 Syntax for control references

The same syntax and addressing behavior that works with event handlers works with formulas, and the notations seen previously work as expected.

For example:

A "Total" Decimal control refers to a sibling "Sales" control using the ${Sales} reference.

A "LineTotal" control in a table refers to a "Quantity" control in its same row as ${Quantity=} (the equal sign meaning "same row as me")
A “Price” control in a Table refers to a “Margin” control that is a sibling to the table containing Price as ${../Margin} (because Margin is one level up in the view tree from Price)

**Note:** While the ${ } syntax is an optional convenience and can be replaced in event handlers with the less compact page.ui.get("<control-id>") or view.ui.get("<control-id>") alternative, it is *not* optional with formulas.

The ${ } notation (or the compact @{ } notation explained below) are *mandatory* in formulas, otherwise the result of the formula will not be updated after its initial value.

---

**Compact references**

Because formulas are meant to be as simple to use as possible, SPARK supports a more compact notation than the ${ } reference syntax. The following expressions (in Example 9) are fully equivalent (and resolve to the same logic at runtime), but the last is visibly simpler to use than the first:

```javascript
${Sales}.getValue() + ${TaxAmount}.getValue()
```

...is equivalent to:

```javascript
@{Sales} + @{TaxAmount}
```

**Example 9 Compactness comparison between ${ } and @{ } notations**

The @{ } notation shortcuts the various implied control-specific methods calls (such as getText() for a Text control, getValue() for a Decimal or Integer control, getProgress() for a Progress Bar, and so on) and requires less programming skills to express.

That said, the formula expression becomes a JavaScript expression at runtime. This means that the sophistication of a formula is only constrained by the facts that:

- The formula must be a single JavaScript statement (but ternary operations such as `<condition> ? <statement 1> : <statement 2>` are allowed)
- At runtime, the formula (after references have been translated to JavaScript) is placed inside a return() statement

For example, if the result of the formula "@{Sales} + @{TaxAmount}" needs to be fixed to 2 decimals, it can simply be altered as follows in Example 10:

```javascript
(@(Sales) + @{TaxAmount}).toFixed(2)
```

**Example 10 Using JavaScript in formulas**
4.9.2 Formulas referencing specific controls

Formulas between scalar (meaning non-repeating) controls simply need to reference the control(s) involved in the calculation and add whatever needed logic in the expression to produce the result. The example shown in Figure 76 below computes the TaxAmount control's value:

![Figure 76 Formula expression with two specific control references](image)

References between controls in a table

A special case of control-to-control references involves control in a table row referencing another control in the same row. The example below (in Figure 77) shows how the "LineTotal" Decimal control references its peer (meaning same row, same index) controls "Price" and "Quantity" through the @(<control-id>=) notation:
Formulas for control aggregates

Formulas can also be used to compute results from repeating controls in a table. For example, all repeating instances of a "LineTotal" control in a table could be summed up into a grand total.

The @{} notation allows a formulas to refer to all controls named <control-id> as suggested in Example 11:

\[@\{ItemsTable/LineTotal*\}\]

"*" at the end of the reference means "all"

Example 11 Using the aggregate reference notation in formulas

The above reference means all controls named "LineTotal" under the Table control named "ItemsTable".

Functions used with aggregate references

The functions currently available to work with aggregate references are:

SUM( )
COUNT( )
MIN( )
MAX( )
AVG( )

Because those functions work with aggregate references, they are referred to as aggregate functions in the SPARK UI Toolkit.

The illustration in Figure 78 below shows the use of the SUM() function combined with the aggregate notation for all controls named "LineTotal" under the Table named "ItemsTable":

As is the case for all other formulas, value changes in any field directly or indirectly referenced by the SUM() function causes the formula using the SUM() function to be recalculated.

The runtime behavior for the above UI composition is shown in Figure 79 as follows:
4.9.4 Working with data directly

So far, all formula examples given have depended on controls on a Coach or in a Coach View. This works well as long as the controls are in existence (even if they are hidden). Formulas are also able to handle delayed instantiation of controls, and results are automatically updated as the controls come into existence (even asynchronously).

There are specific situations however - for example when using paging with tables - when a portion of the controls in the table may never come into existence (because the user may never display a particular table page). In such cases, the formula doesn't have access all the controls in needs to consider for the computation to show the expected result.

To illustrate the use case, the Figure 80 example below shows a grand total computation based on a $[@{ItemsTable/LineTotal*}] formula with a paging table - before and after the second page is displayed:
For such cases, the SPARK formula framework provides support for a special notation to work directly with the data backing the Table control instead of relying on control references inside the table. This special notation must be used in conjunction with an aggregate function, as shown in Example 12:

\[
\text{SUM}(${\text{ItemsTable}}, \text{FOR\_EACH}(${\text{lineTotal}}))
\]

Example 12 Direct data reference in a Table control

Assuming in the example above, that the ItemTable control would be bound to a list of objects containing the following attributes:

price
quantity
lineTotal

...with such an approach, the total would always be correct since the formula doesn't depend on the existence of the controls in the table and directly uses the table's backing data.

The FOR_EACH expression part, however, provides one more option: Since the SPARK UI Toolkit doesn't make data binding mandatory, one might expect that a Line Total control would have no backing data (since it is always derivable from calculation and could therefore be considered redundant). In such a case, the UI developer needs the sum of price x quantity calculated for each row.

Accordingly, the following Example 13 expression is supported:

\[
\text{SUM}(${\text{ItemsTable}}, \text{FOR}_\text{EACH}({\#\text{price}} \times {\#\text{quantity}}))
\]

Example 13 Using FOR_EACH{} to compute sub-expressions in data-backed formulas

**Note:** The content of the FOR_EACH{} clause can be any valid JavaScript expression that returns a result.

**Tip:** Use the proper syntax with data-backed expressions. Note where parentheses are used and when curly braces are used. They are currently *not* interchangeable!

### 4.9.5 Formulas for non-numerical computations

Formulas do not need to produce numerical results. They work just as well with string data (or other types of data). For example, formulas can also be used to set the following:

- The text value of an Output Text control
- The title of a Panel or Collapsible Panel control
- The text value of a Note control
- The content of a Tooltip control
- The data for a Data control
- The text content of a QR Code control to be graphically represented
- The label of a Button control

The Figure 81 example below shows a formula used to set the title of a Panel control to "Hello " then the name entered in the Text control, or if no text is entered, "Hello user":
4.9.6 Formulas for control value initialization

A very convenient use of formulas (particularly for controls that are not bound to data) is for control value initialization. The example in Figure 82 below shows the value of an Output Text control initialized to a specific value:
In general, formulas do not need to contain references to other controls, a simple JavaScript expression works just as well. Remember however, that the content of the formula expression is JavaScript and therefore, a string specified for initialization must be contained in quotes.

4.9.7 Circular references

With formulas, developers may introduce dependency cycles where a formula in a control directly or indirectly refers back to itself. Sometimes such cycles may exist out of convenience. For example, two formula-based controls may represent two different ways of entering the same data and when the value of one control changes, the other needs to adjust itself to also reflect the new value.

The SPARK UI formula framework quietly tolerates such dependency cycles as long as the change in one control doesn't trigger a different value change in another, which escalates into yet another change in the first, and so on. Such "runaway" change situations are flagged by the formula framework at runtime as soon as they are detected.

The example below in Figure 83 shows a browser log for a Coach that contains a circular reference with a runaway change situation:
As correctly reported in the log, the circular dependency chain is as follows:

```
/A → /B → /C → /A
```

The control references in a circular reference log report are always fully qualified to help the developer locate the source of the problem more quickly.

### 4.10 Reporting and analytics

The ability to visualize data in charts, to render tabular data and to provide analytics capabilities for BPM and business data in general is built into the SPARK UI toolkit through three key enabling aspects:

- **Controls**: 7 chart types represented as 7 different controls
- **Drill-down framework**: An XML-configurable and customizable drill-down framework that automatically gives charts drill-up and drill-down capabilities
- **AJAX service-backed tables (Service Data Table control)** that can be combined with charts and the events they provide to zoom in on data represented in charts

This section covers each key aspect of the SPARK UI reporting and analytic features.

#### 4.10.1 Charts

SPARK UI Charts are highly configurable and interactive controls that can represent data from a backing AJAX service or from Human Service Data. The SPARK UI Toolkit includes seven chart controls that can be categorized in three broad classifications:
Composition: Pie Chart, Donut Chart
Comparison or Evolution: Bar Chart, Area Chart, Line Chart, Step Charts
Profiling: Multi-purpose chart

Figure 84 below provides a sampling at a glance of many of the charts types included in the SPARK UI toolkit:
Connecting charts to data

Charts can visualize data retrieved through a configured associated AJAX service or (less commonly) through data from the Human Service. The illustration in Figure 85 below shows a chart control retrieving data through an AJAX service:

![Chart illustration](image.png)

**Figure 85 Bar Chart backed by AJAX service**

**Note:** The Chart AJAX service developer only needs to work with the input variable named "input" and the "dataSeries" output variable. All other inputs and outputs are automatically populated by the Chart control or by the Drill Down service.
The next illustration in Figure 86 shows the AJAX service sample code used to populate the previous bar chart example:

In summary, to make a single data series chart work, the UI developer needs to complete the following steps:

1. Place a chart on the Coach or Coach View canvas
2. Configure the chart to use an AJAX service (create a chart service based on the AJAX service named "Default Chart Single Data Series Service")
3. Create logic in the AJAX service to populate the tw.local.dataSeries object
Configure the chart’s visual appearance if needed
Configure the chart’s behavior (such as bounds for X or Y axis or both, data tooltips) if needed
Playback the chart and adjust appearance and behavior if needed

Working with charts programmatically

Charts provide a number of methods to influence the behavior, visual representation, and data (re-)query.

The first example (in Figure 87) shows how to influence what a chart displays based on input data that is used in the chart’s AJAX query. The illustration below shows a media consumption chart that adjusts its data based on the region selected:

![Figure 87 Example media consumption data represented in bar chart based on region](image)

The above behavior is modeled with a Bar Chart SDS control, a Single Select control containing regions (for example Europe, US), and an AJAX service backing the chart. The bulk of the logic is in the AJAX service performing the actual query (this is usually the case and is not really specific to using the SPARK UI toolkit).

On the client side, the Single Select “On Change” event simply needs to set the chart’s query data to the current region selected, then refresh the chart - as illustrated here in Figure 88:
Figure 88 Modeling and development detail for refreshing charts after setting input data
In addition to influencing query behavior, charts provide a number of methods to control visual and data representation behavior.

In Table 12 below (followed by matching illustrations of the behavior of the chart in Figure 90), several charts methods are successively called to show, hide, disable, enable, group data series and to annotate the charts with labeled horizontal and vertical lines. The chart starts off in the state depicted in Figure 89:

```
Figure 89 Initial chart state before transformation sequence
```

```
Table 12 Sequence of chart data representation and annotation method calls

<table>
<thead>
<tr>
<th>Step</th>
<th>Logic</th>
<th>Step</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;Initial chart state&gt;</td>
<td>9</td>
<td>chart.removeVerticalLine();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>chart.transform(&quot;area-spline&quot;,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;Sample 3&quot;);</td>
</tr>
<tr>
<td>2</td>
<td>chart.groupSeries(true, [&quot;Sample 0&quot;, &quot;Sample 1&quot;, &quot;Sample 2&quot;, &quot;Sample 3&quot;]);</td>
<td>10</td>
<td>chart.transform(&quot;line&quot;, &quot;Sample 2&quot;);</td>
</tr>
<tr>
<td>3</td>
<td>chart.addVerticalLine(2, &quot;Line 2&quot;);</td>
<td>11</td>
<td>chart.showSeries();</td>
</tr>
<tr>
<td></td>
<td>chart.groupSeries(true, [&quot;Sample 2&quot;, &quot;Sample 3&quot;, &quot;Sample 4&quot;]);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>chart.groupSeries(true);</td>
<td>12</td>
<td>chart.focusSeries([&quot;Sample 0&quot;, &quot;Sample 1&quot;, &quot;Sample 2&quot;]);</td>
</tr>
<tr>
<td>5</td>
<td>chart.defocusSeries([&quot;Sample 0&quot;, &quot;Sample 3&quot;]);</td>
<td>13</td>
<td>chart.addHorizontalLine(60, &quot;Line 1&quot;);</td>
</tr>
<tr>
<td>6</td>
<td>chart.groupSeries(false);</td>
<td>14</td>
<td>chart.groupSeries(true);</td>
</tr>
<tr>
<td></td>
<td>chart.transform(&quot;bar&quot;);</td>
<td></td>
<td>chart.transform(&quot;bar&quot;);</td>
</tr>
<tr>
<td>7</td>
<td>chart.transform(&quot;line&quot;, &quot;Sample 0&quot;);</td>
<td>15</td>
<td>chart.focusSeries([&quot;Sample 0&quot;, &quot;Sample 1&quot;, &quot;Sample 2&quot;, &quot;Sample 3&quot;, &quot;Sample 4&quot;]);</td>
</tr>
<tr>
<td>8</td>
<td>chart.hideSeries([&quot;Sample 1&quot;, &quot;Sample 4&quot;]);</td>
<td>16</td>
<td>chart.transform(&quot;spline&quot;);</td>
</tr>
</tbody>
</table>
```

The matching successive runtime representations of the charts (for each of the 16 calls) is shown in Figure 90 below:
Figure 90 Chart transformation sequence using 16 successive chart method calls
Events

Charts can respond to lifecycle events (after load), user events (when a clicking or tapping the chart) and backing AJAX service events (when the data series is returned from the service). Table 13 lists events common to chart controls:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Load</td>
<td>Fired immediately after a chart is loaded (but it is not necessarily populated with data at that time). Useful to take initializing steps (such as adding a menu with static menu items for the chart - see Menus section for more information)</td>
</tr>
<tr>
<td>On Refreshed</td>
<td>Fired every time a chart has loaded and rendered new data from its associated AJAX service (or when the chart loads its bound data if not using an AJAX service).</td>
</tr>
<tr>
<td>On Click</td>
<td>Fired when the user clicks or taps on a data point in the chart. The data point that was selected can be retrieved through the chart’s <code>getSelectedDataPoint()</code> method. The On Click event is also the best place to add context menu items - see Menus section for more information)</td>
</tr>
<tr>
<td>On Menu Action</td>
<td>Fired when a menu item has been clicked (see the Menus section below for how to create context menu items). The action context variable provides <code>action.name</code> to determine what menu item was clicked.</td>
</tr>
</tbody>
</table>

Table 13 Chart control events

Menus

Menus are automatically displayed on charts when:
- The Behavior > Enable Menu configuration option is set on the chart
- A data point (showing as a bar in a Bar Chart, a slice in the Pie Chart, and so on) is clicked by a user
- There is at least one menu item defined for the chart (through `addStaticMenuAction()`, `addTempMenuAction()` or because the chart's drill-down behavior is active)

The example below adds two menu actions (through the `addStaticMenuAction()` method) to a Bar Chart. Menu items are associated with action names, the action name is then checked in the "On Menu Action" event to help decide what to do when a menu item is clicked:
Sometimes it is useful to add a menu item that is only valid given the context of the click on the chart. This next example shows how to create a temporary menu item that is only shown when a bar labeled “Category 0” is clicked. Figure 92 below shows how the `getSelectedDataPoint()` chart method is used in the On Click event to examine what the user has clicked on (in this particular case the label of the select data point is checked):
Figure 92 Using temporary menu items for context-specific chart clicks

**Note:** Temporary menu items are best created in the Chart’s On Click event. They disappear as soon as the menu closes. Temporary items are listed in a separate (top) section of the context menu.

4.10.2 Working with Tables and Charts

Reporting requirements often include both displaying summary data and providing details regarding some aspects of a chart in tabular form. Charts controls and the Service Data Table control can work together to provide such an experience. In Figure 93 below, the user taps in the category of a chart (for example News) to request more detailed data on News media consumption to be displayed in a table under the chart:
Figure 93 Runtime example of chart and table controls working together

This example relies on a chart menu item click to 1) request a table query corresponding to the selected data point label ("News" in this case) and 2) set the title of the collapsed panel containing the table to "Details - " <data point label>.

Once the AJAX query for the table returns, the collapsed panel expands (triggered by the table's "On Service Data Loaded" event) and the populated table is revealed.

The design-time detail for this interaction is provided in Figure 94 below:
Figure 94 Design-time composition for chart, table, panel interaction
4.10.3 Drill-down behavior

SPARK UI charts provide data drill-down support. The following prerequisites must be completed to fully enable this capability:

The chart must be a Single Data Series (“SDS”) chart. All such chart controls are named with an “SDS” ending in the control palette.

The chart must have its Behavior > Enable Menu option set (otherwise the popup menu that makes the drill-up and drill-down options will not be available).

The chart must use an AJAX service to retrieve its data (bound data does not allow drilling down).

The associated AJAX service must follow a particular drill-down pattern requiring an XML document to describe the drill-down structure and a simple mapping of data to the AJAX service's input and output data (more detail in the next section).

Optionally, the Appearance > Show Breadcrumbs option can be set for easier drill-down tree navigation.

Once the drill down behavior is enabled, charts can work as depicted in the Figure 95 Donut Chart example below:

![Figure 95 Drill-down-enabled chart behavior](image)
Creating drill-down services

The first step to creating a drill-down service is to plan out the drill-down tree. The drill-down tree creates the drill down structure, or in other words, the dimension paths or branches through which drill-down will occur.

Drill-down tree

The drill down tree describes all possible paths of data drill-down allowed along desired dimensions. Creating a drill-down tree is an exercise in articulating the main branches and dimensions needed to realize specific data analysis requirements.

The example in Figure 96 shows a particular drill down tree for the visualization of media consumption from a general categorization (for example News, Entertainment, Living, Education, Style) to narrower topics under each categorization (for example Money or Politics under News, Sciences or Arts under Education), to the geographical region associated with media consumption (for example Europe, U.S.). Note in this simple case that the drill-down tree only has a single branch from the "Type" root dimension:

![Figure 96 Single branch drill-down tree example](image)

The drill down tree is expressed as an XML document. Each node in the tree is represented by a `<drilldown>` element. The `<drilldown>` element contains the query associated with the node as well as any child `<drilldown>` element that make up the drill-down path. The first Figure 97 example below shows the drill-down tree with the "Type" root node only:
Figure 97 Simple drill-down tree XML document - root node

This next Figure 98 example adds the "Sub-type" node to the tree in the XML document:
Lastly, the Region node (child of the Sub-type node) is added in Figure 99 to complete the drill-down tree all the way to the Region dimension:
Figure 99 Simple drill-down tree XML document - root (type), subtype & region nodes

The next Figure 100 illustration shows the drill-down tree built in Figure 99 above in context with the behavior of the chart at runtime. The dotted lines show how each
element of a chart (title, breadcrumbs, drill-down choice(s)) relates to the information on the drill-down tree and vice-versa:

```
<drilldown name="Type">
  <!-- Drill-down/Breadcrumb label -->
  <jq-query description="Media Consumption">
    <script>
      <!-- Chart title -->
      <script>
        <drilldown name="Sub-type">
          <!-- Drill-down/Breadcrumb label -->
          <jq-query description="Media Consumption by sub-types">
            <script>
              <!-- Chart title -->
              <script>
                <drilldown name="Region">
                  <!-- Drill-down/Breadcrumb label -->
                  <jq-query description="Media Consumption sub-types by region">
                    <script>
                      <!-- Chart title -->
                      <script>
                        <drilldown>
                          <!-- Chart title -->
                          <drilldown>
                        </drilldown>
                      </script>
                    </script>
                  </jq-query>
                </drilldown>
              </script>
            </script>
          </jq-query>
        </script>
      </script>
    </script>
  </jq-query>
</drilldown>
</script>
```

Figure 100 Chart behavior super-imposed with drill-down tree XML document
Tip: Sometimes the JavaScript drill-down query logic may also need to use input data specified on the chart (the data specified through `<chart>.setQueryData()` method). The input data is referenced in the js-query script as `tw.local.input`.

Drill-down trees structures can be more comprehensive than the one illustrated above. Figure 101 below shows a structure that could provide additional drill-down options, which would then be reflected at the right drill-down level on the chart's context menu (with multiple drill-down options when prescribed by the tree):

![Figure 101: More complex drill-down tree more than one branch](image)

Note: To create a fork in the drill-down tree, just add another `<drilldown>` element as a sibling to an existing one. A parent `<drilldown>` element can have as many direct `<drilldown>` child elements as needed. Also, note that not all branches need to have the same length if the data analysis requirements do not necessitate it.

Lastly, the `<js-query>` element in a drill-down tree is not the only way to specify query logic. SQL can be used instead of JavaScript with a `<sql-query>` element. The illustration in Figure 102 below shows how the Type > Sub-Type > Region drill-down tree might be implemented with SQL queries:
Figure 102 Drill-down tree using SQL queries

Unlike with `<sql-query>`, output mapping is necessary to the value and the label of a `DataPoint`.

Instead of mapping JavaScript variables to parent value reference, `?` query placeholders are mapped.

The order of the mapping is important. The 1st mapping is for the first `?` placeholder and the 2nd for the second placeholder.
Sometimes the SQL drill-down query may also need to use the input data specified (through the setQueryData() method) on the chart. The input data is referenced in sql-query by adding a special `javascript:tw.local.input` field reference, as shown in Example 14 below:

**Example 14 Referencing AJAX service's tw.local.input in drill-down tree's sql-query**

```
<sql-query description="Product sales by store">
  <in>
    <!-- Store id -->
    <field ref="../@value" type="VARCHAR"/>
    <!-- Brand -->
    <field ref="javascript:tw.local.input" type="VARCHAR"/>
  </in>
  <sql><![CDATA[
    SELECT
      SUM(QTY) AS TOTAL_QTY, PRODUCT
    FROM
      STORE_SALES
    WHERE
      STORE = ? AND BRAND = ?
    GROUP BY
      PRODUCT
  ]]></sql>
  <out>
    <label field="PRODUCT"/>
    <value field="TOTAL_QTY"/>
  </out>
</sql-query>
```

**Note:** The elements `<js-query>` and `<sql-query>` are mutually exclusive under a `<drilldown>` element, but in the same drill-down tree, some `<drilldown>` elements can contain JavaScript queries while others can contain SQL queries.

**Drill-down AJAX service**

Once the drill-down tree XML is created, implementing the AJAX service associated with the chart is straightforward and entails the following steps:

- Copy the "Drill Down Query Service Template" AJAX service from the SPARK UI toolkit to the current Process App or Toolkit.
- Rename the copied AJAX service (for example "Media Consumption Drilldown Service")
In the AJAX service, select the Server Scriptlet activity and replace its Implementation with the XML Drill-down tree created.

On the Variables tab of the AJAX service, ensure that none of the input variables have a default (the template might have defaults). If they do, unset the default.

The Figure 103 illustration below shows the Drill-down AJAX service with the replaced XML drill-down tree as per the previous instructions:

**Figure 103 AJAX Drill-down tree for single data series charts**

- **Associating the Drill-down service with the Chart**

  The last step entails associating the AJAX service with the chart control and ensuring its menu configuration is enabled, as shown in Figure 104 below:
4.11 Creating SPARK-compatible controls

BPM UI Developers can create their own Coach Views with SPARK capabilities including:
- Addressing scheme-readiness
- Optional data binding support
- Extended event support
- Extended validation support
- Formulas
- Methods
Some Coach Views may be built from scratch, while others may be composite views (for example Contact Information, Billing Address) that already contain several SPARK UI controls. This section covers both examples starting with a simple "built-from-scratch" scenario.

4.11.1 Sample "Rating" control

The Rating control example (in Figure 105) shows a simple rating-like control with 5 stars where the user can select 0 to 5 stars, which yields an integer rating value from 0 to 5:

![Figure 105 Rating control - high-level requirements](image)

The runtime example in Figure 106 below shows how the control might behave when using SPARK capabilities such as eventing and formula support to work with other controls (an Output Text to display the rating in text form (for example 3/5) and a Text Area control to request details if a rating is less than 4/5):
Note: Creating a control from scratch is a somewhat more technical activity since it requires basic UI programming (or possibly "wrapping" a control from a library) and connecting the control to the SPARK UI runtime.

General behavior

As is the case for most SPARK controls, basic script, Asynchronous Module Definition (AMD) and styling dependencies are required to set up the Rating control. These include:

- Scripts and CSS files:
  - spark-fonts.css (from BPMExt-Style.zip)
  - spark.css (from BPMExt-Style.zip)
  - BPMExt-Core.js (from BPMExt-Core.zip)

- AMD dependencies
  - com.ibm.bpm.coach.util/itilitys (IBM utilities to control visibility)
  - com.salientprocess.bpm.views/bpmext (the core SPARK API)

Additional styling is also defined to create the orange or gray star visual aspect of the Rating control. The control's behavior configuration and variable mapping details are provided in Figure 107 and Figure 108 below:
Figure 107 Rating Coach View’s “Behavior” tab details in IBM Process Designer

Figure 108 Rating Coach View’s Variable tab details in IBM Process Designer
Control load logic

A Coach Framework control becomes a SPARK control at "load" time. The logic sequence in Figure 109 below shows the steps (method calls) required to set up and connect a control such as "Rating" to all the SPARK UI features needed for addressing, event, and formula support:

Note: The setRating and getRating() methods that are referred to in the load logic below are defined later. They are created for convenience and merely "wrap" context.binding.get or context.binding.set into intuitive methods for the UI developer.

Note: The order in which SPARK control methods are called - as suggested in the above load logic - should be followed if possible as it represents the most logical sequence for when SPARK features for the control should be initialized at load time.
Control inline logic

The Inline JavaScript section of the Rating control example (shown in Figure 110 below) contains basic HTML DOM and JavaScript logic (for visual control updates) and definitions for control methods. The only two methods defined in this case are merely for convenience, but the control developer may choose to include many more methods to make the control more powerful and usable.

![Custom HTML DOM/JavaScript UI logic to update the rating stars with the proper coloring when the rating is updated: (Logic not particularly specific to the SPARK UI toolkit)](link)

Usability/programmer-friendly methods to set and get the rating. These methods are arguably optional.

**Figure 110 Rating control inline JavaScript logic**

Control view logic

Figure 111 below shows how the view logic for the Rating control simply updates the visibility status and draws the initial coloring for each star (see `updateStars()` in Figure 110) according to the value of the rating associated with the control:
Control change logic

Because visibility status and rating value can change during the lifetime of the control instance, the Coach Framework "change" event is used to detect such changes and allow the control to react appropriately - as illustrated in Figure 112:

Figure 112 Rating control change logic

Note: To keep the Rating control example to a minimum, the Read-Only visibility status is not implemented.
Note: For the `executeEventHandlingFunction()` SPARK call shown in the change logic above, the event name is the same name used with the `registerEventHandlingFunction()` call issued at load time. It is also the name of the control's configuration property that represents the "On Change" event.

Control unload logic

Lastly, the SPARK resources associated with the Coach View should be freed when the Coach Framework unloads the view. Figure 113 below shows the SPARK-related unload method to be called at Coach View unload time:

Figure 113 Rating control unload logic

Recap of important SPARK methods

The methods recapped in Table 14 below are instrumental in enabling a Coach View to play in the extended Coach Framework programming model provided in the SPARK UI toolkit:

Table 14 Key SPARK feature enablement methods for Coach Views

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bpmext.ui.registerEventHandlingFunction</code></td>
<td><code>this</code> (the reference to the coach view being loaded)</td>
</tr>
<tr>
<td><code>Used for event handling</code></td>
<td><code>eventName</code> (name of the configuration property for the particular event)</td>
</tr>
<tr>
<td></td>
<td><code>&lt;var name 1…&gt;</code> (name of context variable to pass to event when fired)</td>
</tr>
<tr>
<td></td>
<td>- optional</td>
</tr>
<tr>
<td></td>
<td><strong>Up to 9 context variable names can be specified</strong></td>
</tr>
<tr>
<td><code>bpmext.ui.executeEventHandlingFunction</code></td>
<td><code>this</code> (the reference to the coach view)</td>
</tr>
<tr>
<td><code>Used for event handling</code></td>
<td><code>eventName</code> (name of the configuration property for the particular event)</td>
</tr>
<tr>
<td></td>
<td><code>&lt;var 1…&gt;</code> value or reference mapping to the first context variable name</td>
</tr>
<tr>
<td></td>
<td>- mandatory if specified at registration time</td>
</tr>
<tr>
<td></td>
<td><strong>Up to 9 context variables names can passed</strong></td>
</tr>
<tr>
<td>Method</td>
<td>Parameters</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>bpmext.ui.setupFormulaTriggeredUpdates</td>
<td>this (the reference to the coach view)</td>
</tr>
<tr>
<td>Used to setup formula support</td>
<td>callbackFn (function to be called whenever the formula associated with the control computes a result)</td>
</tr>
<tr>
<td></td>
<td>valueFn (method to be called on the control when the @ctl-id notation is used in a formula)</td>
</tr>
<tr>
<td>bpmext.ui.broadcastExpressionTrigger</td>
<td>this (the reference to the coach view)</td>
</tr>
<tr>
<td>Used to alert other formula-dependent</td>
<td></td>
</tr>
<tr>
<td>controls that this control’s value has</td>
<td></td>
</tr>
<tr>
<td>changed (triggers formula recalculations in</td>
<td></td>
</tr>
<tr>
<td>other controls)</td>
<td></td>
</tr>
<tr>
<td>bpmext.ui.loadView</td>
<td>this (the reference to the coach view)</td>
</tr>
<tr>
<td>Initializes the SPARK features for the</td>
<td></td>
</tr>
<tr>
<td>control</td>
<td></td>
</tr>
<tr>
<td>bpmext.ui.unloadView</td>
<td>this (the reference to the coach view)</td>
</tr>
<tr>
<td>Removes the SPARK features for the control</td>
<td></td>
</tr>
</tbody>
</table>

### 4.11.2 Sample Credit Card composite Coach View

The Credit Card control example (illustrated in Figure 114) shows a mini form-like view where a user can enter credit card information and where self-contained server-side validation of the credit card information (meaning in the view, not in a diagram) can be triggered through an AJAX service:

**Credit Card Coach View:**
- 5 card-related input fields
- AJAX validation when all fields are filled out
- Event for card validation success/failure
- Method to clear/reset fields
- Configurable Input Group button colors based on coach view color property

![Credit Card Coach View](image)

*Figure 114 Credit Card control - high-level capabilities*
The runtime example in Figure 115 below shows how the control behaves with correct and incorrect data entry. Validation for the card number and CVV (Card Verification Value) is simulated in the example by verifying that the card number specified starts with the same digits as the CVV:

Figure 115 Credit Card control - runtime behavior

**General Behavior**

A Credit Card control can be placed in a Coach or Coach View. When a user completes data entry on all the fields in the Credit Card control, an AJAX validation is triggered to validate the CVV with the credit card number provided (as explained previously, by checking that the credit card number starts with the same digits as those provided for the CVV).

The Credit Card control provides an "On Validation" event which is fired whenever CVV validation asynchronously returns a passed or failed result. In the previous example, the containing page uses this event to display an alert for a failed result, or clear any previous alerts for a passed result. The control also provides a clear() method which is called when the "Reset Card Info" button is clicked. The scenario, as composed in IBM Process Designer, is shown in Figure 116 below:
Unlike for non-composite views, it is not strictly required for a composite Coach View to load the "bpmext" (SPARK UI) namespace through the AMD. (This is because the Coach Framework loads SPARK child controls first and because SPARK controls already load the SPARK API and "bpmext" in the global namespace).

However, it is still good practice to use AMD-loading including to load Coach utilities (e.g. to control visibility). The scripts and dependencies are as follows:

Scripts and CSS files:
- BPMExt-Core.js (from BPMExt-Core.zip - optional in this case)

AMD dependencies
- `com.ibm.bpm.coach.utils/utilities` (IBM utilities to control visibility)
- `com.salientprocess.bpm.views/bpmext` (the core SPARK API - optional in this case)

**Note:** Composite Coach Views are generally simpler to create than basic (non-composite) views because they usually require much less "low-level" HTML DOM programming and CSS development - often they require none at all.

- **Composite control structure**

As a "composite" Coach View, the Credit Card control contains the following controls:

- **Text:** For name on the card
- **Masked Text:** For the card number
- **Integer:** For CVV and billing ZIP code
- **Date Picker:** For the expiration date (showing only month and year)
- **Input Group:** To add icons to a few input fields (purely used to improve visual aspect)
- **Horizontal Layout:** Holds horizontally-displayed fields and reflow layout when directed responsive sensor
- **Responsive Sensor:** Placed at the top level of the view to sense the width of the Coach View and trigger layout reflows when the view width is <= 420px

Figure 117 below provides the structure of the Credit Card view in IBM Process Designer, the variable and configuration options associated with the view and how the "colorStyle" configuration option relates to the color of its contained Input Group controls:
Coach view-based responsiveness

The Responsive Sensor control, when used in combination with the Horizontal Layout control in this case, allows the coach view layout to reflow predictably regardless of how the Credit Card control is laid out on a page or on another view, whether taking the full width of the page or not.

Figure 118 below shows how view-level responsiveness is configured between the responsive sensor and the layout controls:
Control load logic

The load logic required for the Credit Card composite view allows the registration of the "On Validation" event and the initialization of the view as a SPARK UI view, as shown in Figure 119 below:
Figure 119 Credit Card control load logic

**Note:** Context variables names specified with `bpmext.ui.registerEventHandlingFunction()` become available in the inline event corresponding to the event registered. However `bpmext.ui.executeEventHandlingFunction()` is responsible for actually passing in the proper order the values or references that map to those context variables (see how `bpmext.ui.executeEventHandlingFunction()` is used in the `_setCardValid()` method detailed in the Inline JavaScript logic of the Credit Card coach view below).

### Control inline logic

This section contains details for both the Credit Card coach view's Inline JavaScript logic and for the Credit Card Validation Service logic. They are shown together to provide better context between the client and server-side interactions.

The three coach view method detailed are as follows:

- **_checkCard:** Called from the "On Change" events of all input controls (Text, Masked Text, Integer, Date Picker contained in the Credit Card view). This method checks if all input controls have content and if so, invokes the AJAX service through the Service Call control.

- **_setCardValid:** Called from the "On Result" and "On Error" events of the "CCValidationSvc" Service Call control. Fires the "On Validation" event of the Credit Card view, passing in a valid and error context variables required by the event.

- **clear:** The only explicitly created public function for the Credit Card control. Used to blank out or reset all the input controls in the view.

**Note:** By convention, SPARK control methods and attributes that start with "_" are not considered public and should not be used by developers using these controls.

The Credit Card Inline JavaScript and associated AJAX service logic is provided in Figure 120:
Figure 120 Credit Card control inline and AJAX service logic

**Note:** Because the AJAX service invocation is done through the Service Call control, the Credit Card view has no dependency on diagram logic for server interactions, which fully encapsulates the invocation of its business logic behavior, thus making the view readily reusable.

- **Event flow detail**

  The Figure 121 event sequence below ties together the various events inside the Credit Card coach view with the _checkCard() and _setCardValid() methods already explained:
Control unload logic

Lastly, the Credit Card control follows the regular pattern for unloading a SPARK UI control, as shown in the control's unload event (Figure 122) below:
SPARK UI pattern for composite Coach Views

The Credit Card example is by no means comprehensive and could easily be enhanced to provide better support for visibility settings, to change certain aspects of the control at runtime, to support more fine-grained validation capabilities, and to potentially provide a more extensive set of configuration options.

That said, the example provides the basics of creating composite Coach Views that contain SPARK UI controls. By following the patterns explained, composite views can "play" well in the SPARK-extended Coach Framework programming model, providing methods, events, configuration options, responsiveness capabilities, and encapsulation characteristics that are consistent with the SPARK UI pattern.

Following these patterns consistently can increase the power, flexibility, reusability, maintainability, and ease of use of controls (whether simple or composite) created for IBM BPM UIs.

4.12 Solutions for common patterns and requirements

4.12.1 Working with large forms

Without necessarily considering good User Experience (UX) guidelines (which tend to suggest limiting the amount of content presented to a user all at once), it is sometimes necessary to mitigate the performance impact of loading a very large number of controls on a single Coach (the result of extreme cases being a web browser that "freezes" or becomes unresponsive for several seconds).

An effective performance impact mitigation approach can include a combination of some or all of the suggestions below:

- If displaying data in a Table control, change the rendering option of one or more table columns to "Simple HTML", which can drastically reduce the cost of rendering the table (the tradeoff being that those columns cannot be edited/updated since they are not Coach Views at runtime).

- For Table, Service Data Table or (Horizontal and Vertical) Layout controls, the following loading options can completely eliminate a browser freezing problem by loading content asynchronously while still providing an option to fine-tune the overall content load time:
  - Configuration > Performance > Async Loading: Check to enable
  - Configuration > Performance > Async Batch Size: Number of rows (or controls for a Layout) loaded at the same time as part of an asynchronous batch. The larger the batch, the faster the overall load time, at the cost of the browser acting less responsive.
To suspend the instantiation of one of more controls until explicitly requested, use a Deferred Section control. For example:

- A Tab Section control containing Deferred Sections directly under each of its tabs might delay the instantiation of the content of each tab until the particular tab is selected (using the lazyLoad() method on the deferred section control when the On Tab Changed event is fired), or over several seconds, in the background.
- A Table control containing potentially large Coach Views on each row may instead display a few light columns including a button to request the display of the heavy Coach View only if the button is clicked. The heavy Coach View would be contained in a Deferred Section which would in turn only load the processing-intensive content of the corresponding row when or if requested.

Because the Table and Service Data Table controls have already covered in previous sections, only the scenarios using Deferred Sections are provided here.

- **Tab Section with on-demand content instantiation**

The scenario in Figure 123 below shows how a Tab Section and a Deferred Section under each tab can work together to delay the loading of Coach Views until the tab containing them is selected:

![Figure 123 Example of deferred content loading in a Tab control](image-url)
Under such a scenario, (other than the template HTML which the Coach Framework inserts into the page) the Billing Info view does not exist at all until the Deferred Section lazy-loads it.

**Note:** Because the Billing Info Coach View is small (contains ~10 controls) in the particular example shown above, using a Deferred Section in a production scenario would likely be overkill (thought it would not hurt either because a Deferred Section has a small footprint). But the larger the view (for example 40 or 50+ contained controls), the greater the benefit.

---

**Table with on-demand content instantiation**

This next scenario shows how a table that would otherwise need to display a lot of content on each row can instead display a few light controls per row (including a deferred section) and only load the full Coach View in the row when the user clicks on the "View…" button to show more detail. The in-table defer-loaded Coach View scenario might behave as shown in Figure 124 at runtime:

![Figure 124 Deferred or on-demand loading of Coach Views in table rows](image-url)
The advantage of this approach is that the performance of a Table that might otherwise be expensive to render (the larger the views contained on each row the more expensive) is managed by only incurring the cost of instantiation if or when the user requests to see more detail. The scenario (shown in Figure 125) can be composed as follows:

In the above composition, the Coach View is contained inside a Modal Section. Because both the Modal Section and the "Contact Info" Coach View are contained under the Deferred Section, even the instantiation overhead of Modal Section is deferred until the user requests to load the content. In fact, it is really the Modal Section that's lazy-loaded. The Coach Framework takes care of loading the view contained in the Modal Section when the Modal Section instance comes into existence.
Note: The Deferred Section's lazyLoad() method takes 2 parameters. The first is a delay in milliseconds to wait before the lazy-loading takes place. The second parameter is for convenience. Calling lazyLoad() on Deferred Sections whose content was already-loaded does not load it again, but - if the 2nd parameter is set to true - it fires the On Lazy-Loaded event again so the logic it contains can execute again.

4.12.2 Server-side paging

Whereas client-side paging loads all records in browser memory but only displays a portion of the in-memory data set (page by page), server-side paging only retrieves in browser memory a small subset of a potentially very large data set, then it displays in the table the entire data subset that is in memory.

Server-side paging is a common requirement for the Service Data Table control but it works very differently from the built-in client-side paging feature of the table controls. The paging behavior is effectively emulated by tracking a cursor or offset in the remote data set and requesting a batch of records (the page size is essentially the maximum batch size, or in other words the maximum number of records to be returned in the query).

For example, with a 250-remote record data set and a page size of 100:
A query of up to 100 records at offset 0 returns the first page of 100 records
A query of up to 100 records at offset 100 returns the second page of 100 records
A query of up to 100 records at offset 200 returns the last page containing the remaining 50 records

Note 1: Server-side paging is most easily achieved with a Service Data Table control since it is the only table control capable of fetching data through an AJAX service.

Note 2: Another more complex and less reusable pattern could be implemented through a Table control with Human Service data fetched progressively through diagram logic and "Stay On Page" interactions, but the pattern presented in this scenario is much simpler and can be readily reused and encapsulated.

The example illustrated below uses a Service Data Table to query (completely artificial) data for Media Consumption volumes. The runtime behavior for the sample server-side paging scenario to be implemented is shown in Figure 126:
The Service Data Table-based pattern has two halves:
The client side, with the Service Data Table control and the logic required to set up the
query (including page size and offset), then refresh the table.
The server side, which includes the table's AJAX Service, with its associated query and
resulting data

- **Client-side implementation**

The sample scenario implementation shown here uses a Coach View which contains the
Service Data Table and the paging navigation controls ("previous" and "next" Button
controls, and an Output Text control to show the current page number).

**Note:** Using a Coach View to contain all the controls and associated business logic is not
at all mandatory for the server-side paging pattern, but it helps keep the implementation
self-contained and reusable.

- **Overall structure**

The overall structure and the logic defined for the "Server Side Paging Table Example"
Coach View are given in Figure 127 below:
Note: The page size in the example provided is arbitrarily set to 5. In a more comprehensive example, the page size could be set from a configuration option, or from a "Page Size" Integer control in the Coach View.

Service Data Table configuration

The Service Data Table is configured with the "Media Consumption Query" AJAX service. The Figure 128 illustration below shows key details of the Service Data Table configuration used for the example:
By default, the Service Data Table control automatically invokes its associated AJAX service at load time for the table. In this example however, the initial automatic query is prevented through the "Start Empty" setting. The initial query is instead done explicitly and with the desired initial offset (0) at load time for the "Server Side Paging Example"
Coach View by calling `this.query(0)` from the view's load handler (see load logic in Figure 127).

### Paging navigation buttons

Since the Service Data Table pager widget in the table footer can only be used for client-side paging, the footer and pager widget are not shown. Instead the server-side paging pattern uses controls such as a Buttons or Icons for the user to page forward or backward.

The illustration in Figure 129 below shows the two paging buttons (labeled "<" and ">") that are used to trigger the AJAX re-query with a new offset. This is done by invoking the `previous()` and `next()` Coach View methods from the On Click event of the "<" and ">" buttons respectively:

![Server-side paging navigation through buttons](image)

**Figure 129 Server-side paging navigation through buttons**

### Server-side implementation

The template for the AJAX service associated with the Service Data Table does not prescribe inputs used for server-side paging. It simply allows for input one data element of type ANY. To combine server-side paging parameters along with other query data, a "MediaConsumptionQuery" complex type is created with the following server-side paging-related attributes (in addition to other attributes business case-specific attributes):

- offset (Integer)
- maxCount (Integer)
The offset and maximum record count are then used in the database query to determine the "page" (meaning the slice of records) for which to return results. The server-side logic for the AJAX service is shown in Figure 130 below:

Figure 130 Server-side paging AJAX service implementation

Pattern summary

The list recaps the pattern-specific aspects of server-side paging with a Service Data Table control:
1. Create an AJAX service associated with the Service Data Table
2. Ensure the input data for the AJAX service includes attributes for an offset and a maximum record count
3. Implement the AJAX Service query logic to take the server-side paging parameters into account
4. Ensure that client-side paging aspects are turned off for the Service Data Table configuration
5. Add controls (such as Buttons, Icons, Links) that can be used to attach table re-query logic to on-click events to allow previous page and next page navigation.

6. Implement simple client-side logic to manage the offset to be specified whenever a re-query is triggered.

**Note:** The example provided was purposefully simplistic. Additional sophistication could include retrieving the full record count upfront to not only display the current page, but also the maximum number of pages available. More variability (to control the page size and other query and sorting criteria) could easily be introduced by extending the pattern presented in this section.

### 4.12.3 Cascading Selects

The "Cascading Selects" pattern describes the situation where the items available in a Single Select control depend on what was selected in a previous Select control. For example the list of counties in a Single Select control is dependent on which state was picked in a previous Select control. The dependency cascade may even span more than two Select controls.

### 4.13 Useful Tips

#### 4.13.1 Positioning SPARK UI controls

In IBM BPM 8.5.5, the IBM Process Designer Web Editor introduced a "Positioning" configuration tab for controls instead of relying on the HTML Attribute tab (to add a CSS style attribute or CSS classes, or both to the control).

For historical and backward compatibility reasons however, some size or positioning-related attributes of certain SPARK controls are exposed through configuration options (using the Configuration tab instead of the Positioning tab).

The SPARK UI Toolkit works consistently across IBM BPM versions starting at 8.5.0.1 (where there was no Web Editor for IBM Process Designer and no explicit Positioning configuration tab). Under 8.5.0.1, configuration options were the simplest and most intuitive way to expose such attributes. The toolkit retained these options after 8.5.0.1 to preserve backward compatibility for customers.

Although size-related configuration options and the "Positioning" tab can often be used interchangeably, *UI developers should use configuration option-based positioning attributes whenever they are available*. If no such configuration options are available for a particular SPARK UI control, the Positioning tab can be used.

**In short:** Use SPARK control positioning settings (such as width, height, padding) from the Configuration tab whenever such options are available. Otherwise use the Positioning tab in IBM Process Designer’s Web Editor.
4.13.2 When to use diagram logic vs. encapsulated logic

When developing a Coach View (which by definition is a reusable asset), the behavior in methods, event handling and the view in general should - to the extent possible - be self-contained to reduce the exposure of the view's internals to external factors and to maximize component reusability. If the view relies on certain AJAX interactions with a specific service, then those interactions should also be contained and modeled in the view instead of relying on boundary events and externally managed data updates.

That said, a Human Service is not the same as a Coach View and the SPARK UI toolkit is very accommodating to developers who want to use a lot of classic Coach Framework capabilities, including with SPARK UI controls (through the use of extensive data binding and boundary events). SPARK even provides the "Navigation Event" control, which allows boundary events to be emitted from any kind of interaction for which a SPARK control fires an event.

➢ Best use of diagram-level logic

The most effective use of diagram logic (when combined with SPARK UI capabilities) is most often one where it is limited to:

- High-level client-side business logic at the page level that adds meaningful context to a service diagram
- Page to page interactions
- Page-level AJAX interactions and other service integrations

Since SPARK doesn't require large quantities of boundary events (and therefore associated diagram-based scripts for each boundary event), Human Service diagrams can become clutter-free and focus on expressing business logic at the right level.

**Note:** Client-side scripts can even make full use of the SPARK API to retrieve and manipulate SPARK controls on the page that emitted the boundary event to which they are attached.

➢ Internal Coach-page level logic

The most convenient way to express simple internal "page-level" business logic with SPARK is with inline events, accessible through the controls' Configuration > Events category.

For more complex business logic (for example more than 3 or 4 statements), one or more `<script>` blocks can be defined on the page through a Custom HTML component and can hold function definitions that are called from the inline events of various control contained by the page.

Lastly, logic and groups of controls that show reusable potential in a page should be rolled into SPARK-enabled composite Coach Views providing the proper level of usability and encapsulation through appropriate configuration options, events, and methods.
4.13.3 Debugging logic

BPM UI developers often need to debug and log business logic, whether in inline events, custom control methods, functions called from events, or client side scripts using the SPARK API. Sometimes, certain parts of the UI must be exercised programmatically to conveniently trigger certain behaviors during a test. Debugging may also be more challenging on large UIs whose many loaded controls might create “noise” that detracts from the problem analysis.

This section provides useful hints to help developers maximize the benefit of the debugging process while using the SPARK UI Toolkit.

- Using the Configuration control

Using the Configuration control is a good first step to debugging as it allows the developer to turn on SPARK logging and provides useful detail which the developer can use to exercise various parts of the SPARK-based UI.

The following Figure 131 provides an example of SPARK logging in a web browser showing, among other things, the load sequence of controls and their respective absolute addresses:
Figure 131 Using the Configuration control to show SPARK debug logging

Note 1: Configuration controls have a page-wide effect and should therefore only be placed in Coaches, not in Coach Views.
Note 2: When a Coach contains repeating content (such as Tables and repeating Layouts) the SPARK log shows the fully qualified and indexed address of the control. For example, the reference /Table1/Price1[1] points to the Price1 control under the Table1 control, in the second row.

Viewing the log on mobile devices

The Configuration control also provides on-screen logging that can be used when testing a Human Service on a mobile device. This can be enabled by selecting the Configuration > Logging > Show Log option of the Configuration control - as shown in Figure 132:

Note: SPARK log entries are only shown if the Configuration > Debugging On option is selected.

Exercising SPARK from the web browser console

Every control on a Coach, whether at the Coach level or in composite Coach Views can be accessed by their absolute address using the page.ui.get() method. Figure 133 shows
that once a control reference is retrieved, methods can be called on them through the web browser console:

![Figure 133 Calling SPARK control methods from a web browser console](image)

**Note:** Features such as code completion allow the UI developer to not have to remember the exact method names and signatures for the SPARK API or for SPARK UI controls. Some web browsers might not support this feature.

If using code completion, the developer should not assume that a control method or SPARK API method is publicly available just because it is visible or accessible from the console. Only the usage of methods published in the SPARK online documentation is supported.

- **Debugging inline event logic**

  It is often useful to be able to debug event logic, even if the logic is very simple. When testing a Coach under a browser that supports the "debugger" directive, "debugger" can be placed in the inline event to pause execution and step through logic from that point - as illustrated in Figure 134:
Figure 134 Debugging inline event logic

Tip: Adding “debugger” in event logic also provides a quick way to discover the context variables available for a particular event.

Forcing Right-to-Left behavior

The Configuration control provides a shortcut to turn on Right-to-Left behavior for a Coach without having to switch the locale for the IBM BPM user or for the web browser.
This is done by setting the Configuration control's Internationalization > Global Text Direction to "Right to Left", as shown here in Figure 135:

![Figure 135 Global text direction setting through the Configuration control](image)

**Note:** The text direction can also be switched at runtime by calling the `setGlobalTextDirection()` method on the Configuration control, passing in the value "DEFAULT", "LTR" (Left-to-Right), or "RTL" (Right-to-Left).
Filtering out noise at debugging time

When a developer needs to test a Coach incrementally it is sometimes useful to turn off parts of the UI to focus on a particular problem. A Deferred Section is a convenient way to do this by moving parts of the UI into it and leaving the "Configuration > Lazy-load Automatically" setting unchecked.

This effectively prevents any content under it from instantiating at runtime.

4.14 More on the SPARK UI Toolkit

From a BPM UI developer's perspective, the SPARK UI Toolkit represents a deliberate compromise between simplicity and flexibility.

The benefit of this approach is that the level of UI development expertise required to be effective is low once the core SPARK UI patterns are understood. At the same time the toolkit provides many extension and customization capabilities to implement very advanced and complex production requirements and scenarios.

Additional content on the SPARK UI toolkit is available and provides a solid supplement to this chapter - exploring several additional aspects of the toolkit in context of the foregoing material, including the following topics:

- Controls: Creating simple and composite Coach Views with SPARK capabilities
- Tables: Custom rendering and performance options
- Charts: Implementing drill-down behaviors
- Useful tips on debugging, deciding when to use diagram-based or encapsulated control logic