IBM® WebSphere®

Leveraging the IBM BPM Coach Framework in Your Organization

Understand how to develop and assemble user interfaces with Coach Views

Investigate three IBM Business Partner offerings from:

Apex Process Consultants, BP3, and EmeriCon

John Reynolds
Mike Collins
Eric Ducos
David Frost
David Knapp
Ivan Kornienko
Bodo Naumann
Pat O'Connell
Paul Pacholski
Gerhard Pfau

Redbooks

ibm.com/redbooks

International Technical Support Organization

Leveraging the IBM BPM Coach Framework in Your Organization

April 2014

SG24-8210-00

Note: Before using this information and the product it supports, read the information in "Notices" on page vii.

First Edition (April 2014)

This edition applies to IBM Business Process Manager v8.5.

Contents

Notices

Trademarks

Preface

The IBM® Coach Framework is a key element of the IBM Business Process Manager (BPM) product suite. With the Coach Framework, process authors can create and maintain custom web-based user interfaces that are embedded within their business process solutions. This ability to create and maintain custom user interfaces is a key factor in the successful deployment of business process solutions. Coaches have proven to be an extremely powerful element of IBM BPM solutions, and with the release of IBM BPM version 8.0 they were rejuvenated to incorporate the recent advances in browser-based user interfaces.

This IBM Redbooks® publication focuses on the capabilities that Coach Framework delivers with IBM BPM version 8.5, but much of what is shared in these pages continues to be of value as IBM evolves coaches in the future. This book has been produced to help you fully benefit from the power of the Coach Framework.

Authors

This book was produced by a team of specialists from around the world working at the IBM International Technical Support Organization, Austin Center.

John Reynolds is a Product Manager for IBM Business Process Manager, focusing to enable business users and authors. His background includes roles as software consultant, solution architect, and product manager. John drives the full lifecycle of product development from gathering and refining requirements, to focusing product vision, to defining product architecture, and managing implementation teams. He holds an M.S. in Computer Science and Engineering from the University of Texas at Arlington, and a Bachelor of Science in Electrical Engineering from Rice University.

Mike Collins is a Senior Software Engineer at IBM. He is currently a lead on the IBM Business Process Manager performance team in the Austin lab, a position he has held for approximately 10 years. He has been a performance analyst at IBM for 17 years, focusing on the Java-based middleware stack. Past performance analysis roles include the IBM JVM and JIT, IBM WebSphere® Application Server, WebSphere InterChange Server, and the WebSphere Adapters. He holds a B.S. in Computer Science from Purdue University. Mike has authored several IBM Redpaper™ publications, notably the series of Business Process Manager Performance Tuning and Best Practices publications.

Eric Ducos is the Chief Technology Officer of EmeriCon, an IBM Smarter Process Business Partner. He currently heads EmeriCon's Research and Development program and has, since 1995, actively participated on many advisory councils and panels related to business process management and operational decision management for IBM. His focus on business process management includes business and technical consulting, management, research, and development. Over his career, Eric has authored a number of frameworks, products, and methodologies in business process management. Recent contributions include co-design and co-authoring of EmeriConVIEWS, this IBM Redbooks publication on BPM Coach Views, and a mobile enablement framework for IBM BPM.

David Frost graduated from Cornell University with a degree in Operations Research and has since worked three years in the BPM space, specifically IBM BPM. David has significant experience in front-end development, including contributing to the BP3 Brazos UI Toolkit for IBM BPM.

David Knapp has 28 years of experience as a Visionary Enterprise Change Agent. After 22 years of developing and implementing Product Lifecycle Management tools and methodologies, in 2006 he introduced IBM BPM (then Lombardi) to Ford Motor Company. He built Ford's BPM Program from the ground up, re-engineering major enterprise processes including Advanced Sourcing, Engineering Release, and Early Bill of Materials. In 2009, he left Ford and founded Apex Process Consultants. David and the Apex team have applied their BPM methodologies and tools to a wide range of industries including health care, product development, and accountancy. Apex is a Premier IBM Business Partner, one of the leading services providers for IBM BPM, and offers a range of IBM BPM products including Apex Coach Views, Apex Repository, and Apex Performance Tools.

Ivan Kornienko is a BPM Architect at BP3. His passion lies in BPM solutions that include a heavy presence from users on non-traditional devices. Taking process solutions from discovery to production, he focuses on building applications that allow people to get their work done regardless of whether they are using desktops, tablets, phones, TVs, or any other network connected devices.

Bodo Naumann started at IBM in 2010 with an internship shortly after Lombardi was acquired by IBM. Since then Bodo worked with IBM Teamworks®, WebSphere Lombardi Edition, and Business Process Manager Standard. In 2011, he joined the ISSW team in the Netherlands as a BPM developer. When version 8 was released, Bodo started building up experience with the new Coach Views. Along the way he shares his findings as samples within the BPM community.

Pat O'Connell has been working in BPM since his graduation from college in 2010. Originally from Chicago, Pat moved to Austin, TX to join the Lombardi team in 2010. In 2012 he joined the BP3 team.

Paul Pacholski has been with the IBM Canada Development Lab for 31 years. Initially, Paul worked as a Senior Developer on several IBM software offerings. For the last 14 years, Paul has held the role of BPM Technical Sales Leader responsible for technical enablement within IBM and influencing BPM product directions. Paul's other responsibilities include helping clients to select the right BPM technology, presenting at technical conferences, publishing technical papers, and filing BPM related patents.

Gerhard Pfau is an IBM Senior Technical Staff Member and a member of the IBM Academy of Technology. He works as product designer on IBM Business Process Manager, IBM's smarter process offering for on-premise BPM. As part of that he has lead the design work for IBM BPM 8.5. Before being a designer, Gerhard was the lead architect for human-centric BPM support in WebSphere. He has worked on architecture and design of the IBM WebSphere-based business process management portfolio since its inception. Gerhard has more than 25 years of experience in commercial software development.

Thanks to the following people for their contributions to this project:

Dave Enyeart, Michael Friess, Werner Fuehrich, Ge Gao, Lorne Parsons, Andreas Schoen, Grant Taylor, Ramiah Tin, Eric Wayne, Jeoff Wilks
IBM

Laura Girodat, Kyle Hoskins, Sethu Katherisan, Rob Robinson, Blake Smith
Apex

Carmen Galicia, Tommy Nguyen, Evan Slate
BP3

John McDonald
EmeriCon

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

1

The IBM Coach Framework and how it can benefit your organization

The IBM Coach Framework is a key element of the IBM Business Process Manager (BPM) product suite. With the Coach Framework, process authors can create and maintain custom web-based user interfaces that are embedded within their business process solutions. This ability to create and maintain custom user interfaces is a key factor in the successful deployment of business process solutions, and this IBM Redbooks publication has been produced to help you fully benefit from the power of the Coach Framework.

The term *Coach* was introduced by Lombardi Software, which was acquired by IBM in 2010. Vince Lombardi was a very successful football coach in the 1960s from whom many have drawn leadership inspiration, and it seemed very natural to call the screens generated by the product *Coaches*. The purpose of these screens is often to *coach* the user through the steps necessary to perform their work.

History aside, Coaches have proven to be an extremely powerful element of IBM BPM solutions, and with the release of IBM BPM version 8.0 they were rejuvenated to incorporate the recent advances in browser-based user interfaces. This IBM Redbooks publication focuses on the capabilities that Coach

Framework delivers with IBM BPM version 8.5, but much of what is shared in these pages continues to be of value as IBM evolves Coaches in the future.

1.1 User interactions with business processes

BPM solutions can help your organization run more smoothly by automating the business logic of your processes and by providing increased visibility and analysis into the performance of your processes. With BPM solutions in place, the execution of your processes can become more predictable and the performance of your processes can improve over time as you incorporate the insights that BPM provides into your process logic.

That is the promise of BPM, and this promise of improved process efficiency has been realized by many organizations over a wide variety of industries, but there is much more to the story that needs to be told.

True process improvement requires much more than getting the right tasks to the right people at the right time. You must also provide the right tools to the people who carry out the work to ensure that they can perform their work in an efficient and error-free manner. Without the right tools for your workers, it is impossible to truly improve process performance.

That is what this paper is really about: Improving user interactions with your organization's business processes. The Coach Framework can help you do that.

Users interact with business processes in various ways, but those interactions can mostly be lumped into a few broad categories:

- Discovering what work needs to be done
- Performing work
- Analyzing the work that has been done in the past
- Reviewing and managing the work performed by a team

User interactions with business processes are all about work, and as BPM solution authors we want to design and implement user experiences that can optimize our users' ability to do their work efficiently and with as few errors as possible.

Users learn about the work that must be performed in various ways. In some cases, users receive notifications via email or SMS. In other cases, users log on to a specific website to view their task list. In either case, it is important for the user to quickly understand the nature of the task and its relative importance to the other tasks that the user is likely responsible for performing. Custom user interfaces are often necessary to deliver the information that a user needs to determine the relative importance of a task.

Users perform the work that they are required to do in a number of ways. In some cases, the work is performed outside of the BPM solution using existing systems, and the user simply informs BPM once the work has been completed. In other cases, the user interfaces for performing the work are implemented as part of the BPM solution. In many solutions, both approaches have to be combined. Regardless of the approach, the user must clearly understand the details of the task to be performed, understand the wider business context of the task, and be provided with user interfaces for supplying and modifying the information that the task must gather or manipulate. Custom user interfaces are often necessary to provide users with optimal tools for performing their work.

Users analyze the work that has been performed in a number of ways and for a number of reasons. During the execution of a process, users need real-time feedback to help them guide the subsequent execution of the process. In a similar vein, managers need visibility into the work that their teams are performing, and tools to help them manage that work. Analysis also occurs after-the-fact to determine if the logic of the process is optimal for the business to run smoothly. In each of these cases, the user needs interfaces that supply the business context information that is most relevant for understanding the reality of how well the process is performing. Custom user interfaces are often necessary to provide users with the most relevant information for analyzing process performance.

Some BPM user interfaces can be used out-of-the-box and will be applicable to all of your business processes, but the reality is that most BPM user interfaces need to be customized to present the information that is most relevant to the user in the most effective way. Determining the *relevant information* is very specific to each type of process and each task within the process, and presenting that information effectively is an art that few have mastered. Coaches help you master the art of creating effective business process user interfaces.

1.2 Coaches: Custom user interfaces for business processes

The preceding section made clear that custom user interfaces are often necessary in order to present users with the relevant information that helps them efficiently carry out their tasks. Every process is different, and every activity is different, so to optimize the user's experience you need to build custom user interfaces that present and manipulate the most relevant information in the most effective manner.

This critical need for custom user interfaces for business processes is the driving factor in the development and evolution of the IBM Coach Framework. The Coach Framework's primary focus is to enable process authors to create and maintain custom user interfaces that will improve their users' interactions with their organization's business processes, which in turn can improve the performance of those business processes.

Maintenance of the custom user interfaces in a BPM solution is crucial due to the frequency of changes to the logic of the organization's business processes. As market conditions change, the logic of the business process has to be adapted. As the logic of a business process changes, the users' interactions with the business process must change. Your ability to adapt the user interfaces of a business process has to keep pace with changes to the process logic, or your organization's agility to respond quickly to change is diminished.

Coaches make it much easier to adapt your user interfaces when your business process changes. User interfaces that are created with the Coach Framework are packaged along with the other artifacts that make up a Process Application. When a new version, known as a *snapshot*, of the Process Application is deployed, all of the Coach-based user interfaces are also deployed. This insures that the users are always presented with screens that are in sync with the correct process logic.

Without Coaches, the deployment of the user interfaces for a process is a separate step that must be performed when a new snapshot of the process is deployed. With Coaches, there is never a concern that the process screens might not match the process logic.

1.3 Coach Views: Custom user interface components

Coach Views are the reusable user interface building blocks that authors use to compose their Coaches.

Some examples of Coach Views include components that allow users to edit text, push buttons, select items from a list, and select check boxes and radio buttons. Other Coach Views are used to define the physical layout of a Coach. Examples of layout components include vertical and horizontal sections, tables, and tab controls.

IBM BPM ships with various Coach Views for presenting and manipulating process data, and also supplies tooling for the creation of additional Coach Views. Coach Views are now being created and distributed by IBM, IBM Business Partners, and by clients themselves. This ability to create and distribute additional Coach Views dramatically increases the power of the Coach Framework.

Each Coach View is tailored to present specific information to the user, and most Coach Views also empower users to gather and manipulate specific information.

When added to a Coach, the author configures the Coach View by binding it to the business data that the component will display and to any configuration data that is necessary to control the appearance and behavior of the component at run time.

Most Coach Views respond when any of the data that they are bound to changes. This allows authors to create dynamic user interfaces by binding multiple Coach Views to the same data. When a user manipulates one Coach View, all of the other Coach Views that are bound to the same data can instantly react.

Coach Views can also be bound to Ajax services that are invoked to retrieve information and update systems that are related to the business processes. This ability to bind to Ajax services allows authors to create highly dynamic components, such as components with fields that are updated dynamically while users are typing. This feature can be also used to create Coach Views that directly interact with services outside of BPM.

There are two basic categories of Coach Views; atomic and composite.

1.3.1 Atomic Coach Views

Many Coach Views are atomic in the sense that all of the HTML and JavaScript that makes up the Coach View is contained within a single component. The stock Coach Views that are shipped with IBM BPM are mostly atomic Coach Views. One example of an atomic Coach View is the Text Field that is shipped with IBM BPM.

Atomic Coach Views are authored by programmers who have a deep understanding of HTML, JavaScript, and Ajax services. They often use third party JavaScript libraries to provide sophisticated behaviors.

Most atomic Coach Views are general-purpose components that are reused across a wide variety of user interfaces.

1.3.2 Composite Coach Views

Coach Views can be defined by combining pre-existing Coach Views.

Composite Coach Views are authored by business subject matter experts.

Most Coach Views that present and manipulate specific business object types are composite Coach Views.

Composite Coach Views are often created when many of the Coaches of a solution include a common section of user interface components. A common header that contains overview information about a process is an example of a section that is often implemented with a composite Coach View.

1.4 Human Services and Coaches

Coaches do not exist as a distinct library artifact in the Process Center. The library artifact that wraps Coaches is called a *Human Service*.

Human Services are the server-side components that manage the presentation of Coaches. In essence, Human Services are *applications* that users use to interact with business processes.

Human Services can be used to implement activities of a business process, or to implement stand-alone applications that can be invoked independently.

Each Human Service can contain one or more Coaches. The Human Service defines the flow between Coaches, and the integrations and messages that are invoked to support the actions that the user initiates with the Coaches.

When implementing a business process activity, a Human Service is initialized with data from the process when the process flow reaches the activity. This allows the Coach authors to easily build user interfaces within the context of a specific activity of a business process.

1.5 Basic types of Coaches

In version 8.5 of IBM BPM, there are two types of Coaches: Those that are used to complete a specific activity of a process, and those that are used to present overview information about one or more processes. These two types are referred to as Task Completion Coaches and Dashboards.

1.5.1 Task Completion Coaches

Task Completion Coaches are user interfaces that let a user work on a specific activity of a process. The lifespan of these Coaches is tied to the lifespan of an activity in the process. The Coach can only be opened while the activity is active.

1.5.2 Dashboard Coaches

Dashboards are user interfaces that let a user work with a specific process or collection of processes. The Dashboard's lifespan is independent from any process or activity. It can be opened at any time.

There are many uses for Dashboards, such as inspecting performance and managing workloads, and the Coach framework provides the power to build Dashboards that are essentially stand-alone applications.

Much of the IBM BPM Process Portal is implemented using Dashboards, and many clients have used Dashboards to build their own custom portals.

1.6 Using Coaches outside the IBM BPM Process Portal

When users are logged on to the IBM BPM Process Portal, the Coaches that they work with are opened in the browser window of the Process Portal, but Coaches can also be exposed to users who do not use the IBM BPM Process Portal.

1.6.1 Launching Coaches via URLs

Every Coach is embedded in a Human Service, and all Human Services can be launched via a URL. This makes it very easy to incorporate Coaches into a wide variety of web-based interfaces. Links to launch Coaches can be embedded on existing corporate web sites, or when appropriately launched within iFrames on corporate web pages. You can discover human services that are exposed for this purpose via a BPM Representational State Transfer (REST) application programming interface (API).

1.6.2 Coaches within mobile applications

A specific example of using coaches via URLs is by using a Coach for the user experience of a mobile application. The IBM BPM iOS application uses this approach. Other samples that illustrate this approach are available on the IBM BPM Samples Exchange at the following link:

http://bpmwiki.blueworkslive.com/display/samples/Mobile+apps

1.6.3 Coaches within JSR 286 portlets

For companies that use IBM WebSphere Portal, the Coach dashboards can be *wrapped* in JSR 286 portlets. This capability allows dashboards to be wired into WebSphere Portal composite applications in the same manner as other custom Java portlets.

1.7 How Coaches can benefit your organization

Now that you have a better understanding of what Coaches are, we can come back to the question of how Coaches can benefit your organization.

1.7.1 Seamless integration of UIs and process logic

Coaches benefit your organization by seamlessly integrating the development and deployment of the custom user interfaces for a process with the development and deployment of the process logic and necessary integrations that make up the complete solution.

IBM BPM does support the use of *external* user interfaces for interaction with processes via REST APIs, but when external user interfaces (UIs) are used instead of Coaches, the process for developing, testing, and deploying the solutions is disconnected. Without Coaches, explicit coordination and governance is required between the UI and process developers.

In most cases, the seamless integration of UIs with the process solution will make it much easier for your company to implement a business process change that also required a change to the UIs of the process.

1.7.2 Tailored UI components for your business

The ability for each business to create their own Coach Views enables organizations to create their own toolkits of UI components that are tailored for their specific needs.

Experienced developers can create rich user interfaces for the key business data and package those user interfaces as building block components that can be used by casual business programmers.

1.8 Conclusion

Hopefully this chapter has helped make it clear that the IBM Coach Framework can be an extremely valuable resource for helping your business to achieve the process agility. The remainder of this book focuses on specific details and insight for how to best use the Coach Framework within your own organization.

Chapter 2, “Assembling user interfaces with Coach Views” on page 11 covers the topic of assembling your custom user interfaces from existing Coach Views. This chapter helps you understand how to connect the user interface components to your underlying business data, and how to make UI components react to changes in related UI components.

Chapter 3, “Building Coach Views” on page 57 covers the topic of developing custom Coach Views. This chapter provides all of the details that are necessary for constructing new UI components when an existing component does not meet your needs.

Chapter 4, “Advanced performance considerations” on page 137 covers the topic of performance considerations when crafting your Coaches and custom Coach Views. Included are many practical tips to help you avoid introduction performance problems in your solutions.

The final three chapters of the book focus on sharing real world lessons that have been learned by a trio of IBM Business Partners who have extensive experience with developing custom Coach Views for their clients. Each chapter offers insights from the unique perspective of a specific partner, Apex Process Consultants, BP3, and EmeriCon. Topics such as dealing with large amounts of data, dealing with mobile UI considerations, and crafting a matched toolkit of Coach Views are presented by folks with extensive experience in building Coach-based solutions for their clients.

2 Assembling user interfaces with Coach Views

In Chapter 1, “The IBM Coach Framework and how it can benefit your organization” on page 1 we learned about Human Services, Coaches, and Coach Views, and how they can be used to build user interfaces. Now it is time to take a closer look at Coach Views and how they can be used to build rich, coach-based user interfaces. In this chapter, we are concentrating on the capabilities that are provided by the product. Those are sufficient for playback scenarios, when you quickly have to mock up a user interface, as well as for many production use cases. If you have user interface (UI) requirements that go beyond that you can build your own coach views, which are explained in the next chapter. The overall chapter is organized into the following sections:

- Getting started
- Building a simple coach
- Configuring coach views
- Advanced configuration
- Coach view toolkits
- Creating Coach Views for business objects
- Further information
- Performance considerations

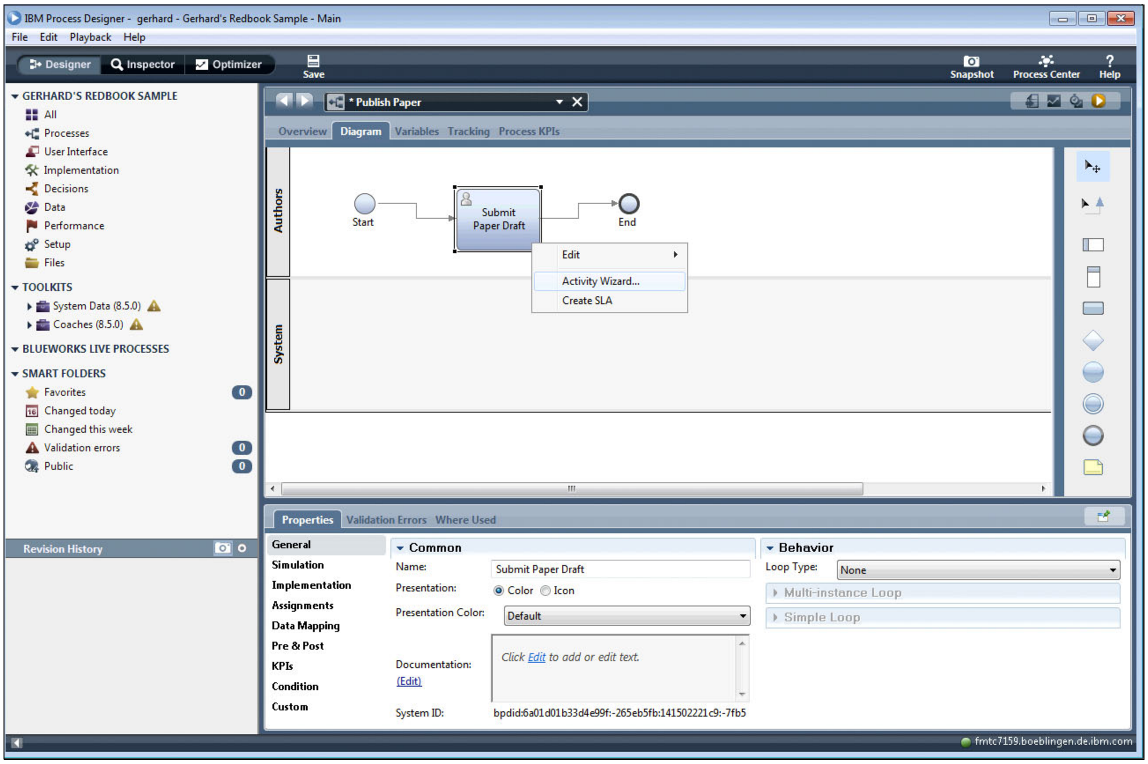
2.1 Getting started

As introduced in Chapter 1, “The IBM Coach Framework and how it can benefit your organization” on page 1, the primary use cases for human services, coaches, and coach views are when building task completion UIs or dashboards. In this chapter, we are focusing on task completion UIs. Task completion UIs are the user interfaces used by a person who participates in a business process by performing human activities. Human activities are used to allow people to enter data, verify data returned by backend systems, give their approval, and so on. All the techniques that are discussed in this chapter also apply when modeling dashboards.

We look at more sophisticated scenarios later, but now we quickly look at how to get started. In this chapter, we explain what you need to know to build coaches by example. If you want to do some up-front reading, see the information center chapter “Building Coaches” at this site:

`http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/buildcoach/topics/tbuildviewcoaches.html`

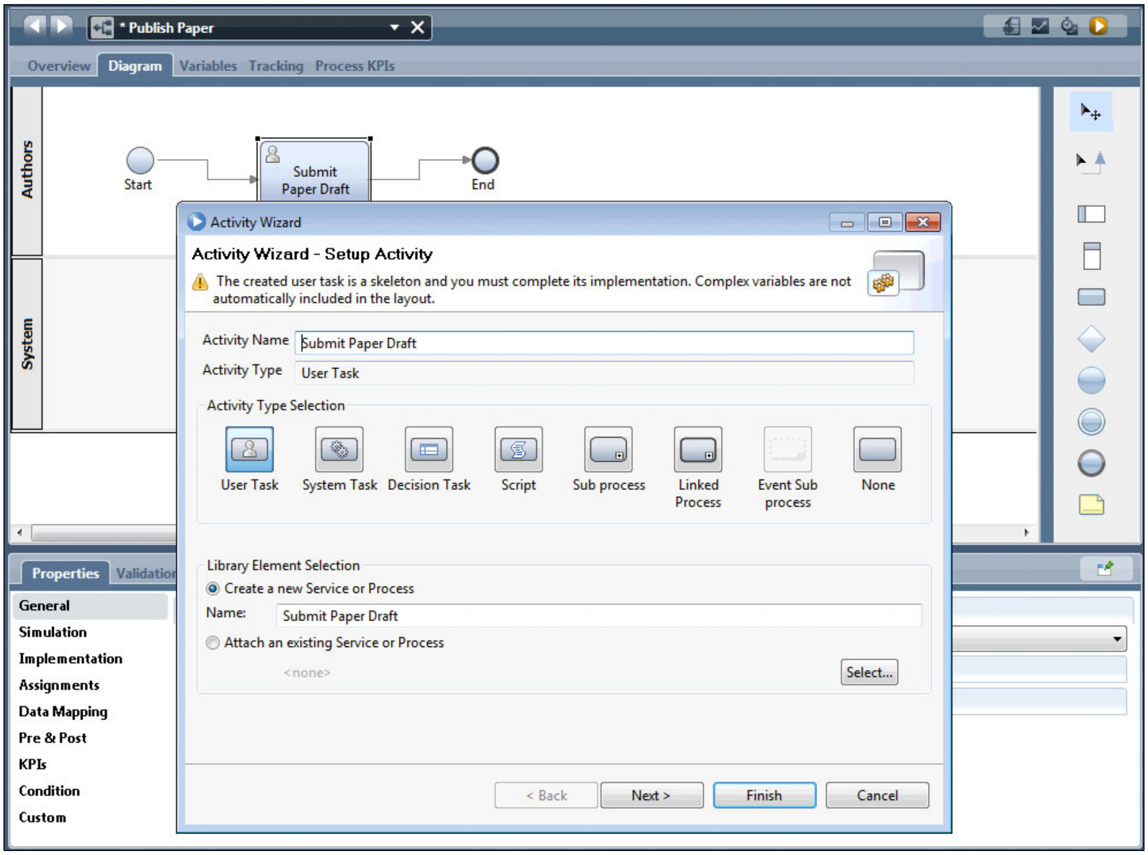
Now we get started with the sample. We created a business process definition (BPD) named *Publish Paper* in Process Designer and we added a single activity *Submit Paper Draft*. By default, that activity has a standard implementation. We do not want to stick with the standard implementation and therefore launch the Activity Wizard to create an implementation for the activity as shown in Figure 2-1 on page 13.



Name: Submit Paper Draft
Presentation: [✓ Color] Icon
Presentation Color: Default
Documentation: (Edit)
System ID: bpdid:6a01d01b33d4e99f:-265eb5fb:141502221c9:-7fb5
Loop Type: None

Figure 2-1 Launching the Activity wizard

The Activity wizard opens. Ensure that the *Activity Type* is *User Task*. That Activity Type is used for human steps in a business process. Also choose *Create a new Service or Process* for the activity. Press **Finish** to close the wizard as shown in Figure 2-2 on page 14.



Activity Wizard - Setup Activity

The created user task is a skeleton and you must complete its implementation. Complex variables are not automatically included in the layout.

Activity Name: Submit Paper Draft

Activity Type: User Task

Activity Type Selection: [✓ User Task] System Task Decision Task Script Sub process Linked Process Event Sub process None

Library Element Selection

[✓ Create a new Service or Process]

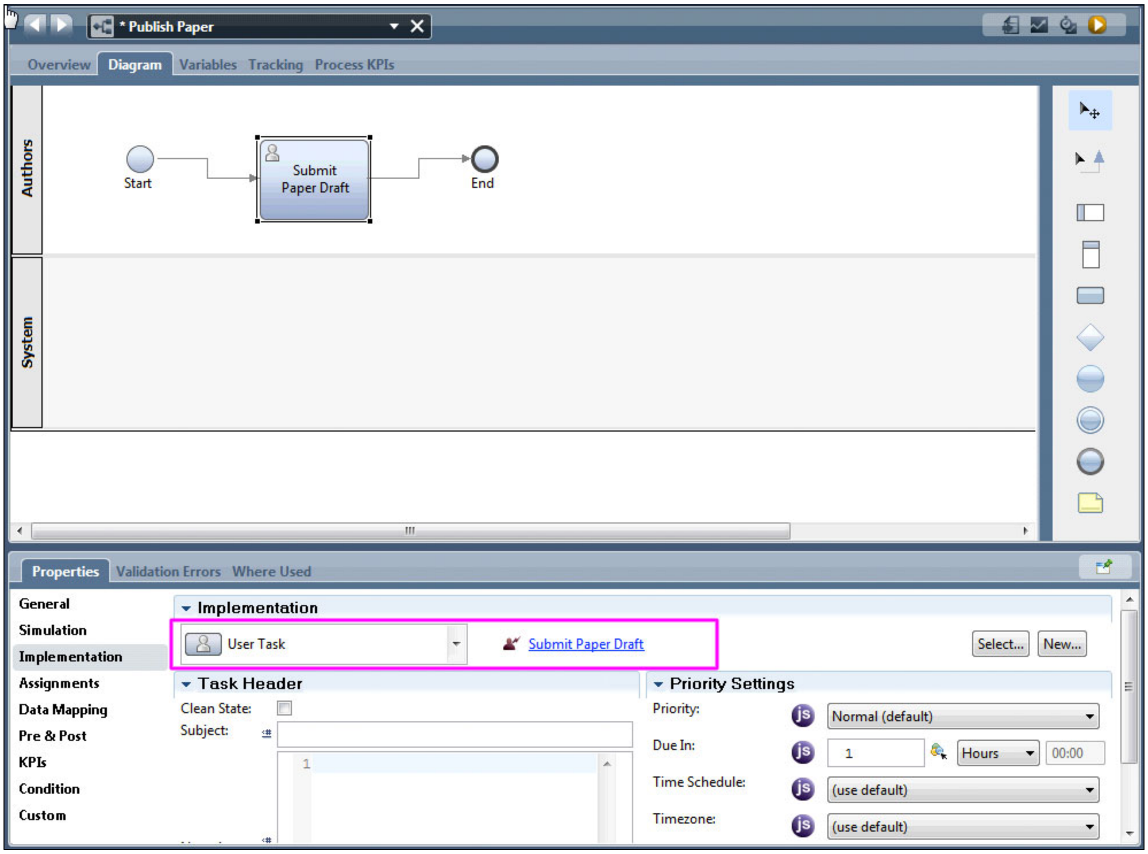
Name: Submit Paper Draft

Attach an existing Service or Process

<none>

Figure 2-2 Activity wizard for Submit Paper Draft activity

As a result, a Human Service named Submit Paper Draft is created for you and it is associated with the Submit Paper Draft activity of the process. Verify that everything worked as expected by selecting the Submit Paper Draft activity and by looking at its *Implementation* definition as shown in Figure 2-3 on page 15.



Implementation: User Task — Submit Paper Draft

Task Header
Clean State:
Subject:

Priority Settings
Priority: Normal (default)
Due In: 1 Hours 00:00
Time Schedule: (use default)
Timezone: (use default)

Figure 2-3   Implementation definition of Submit Paper Draft activity

Double-click the **Submit Paper Draft** link to open the Submit Paper Draft human service in the service editor. That human service contains the task completion UI of the activity. Initially it consists of a single Coach with a single button as shown in Figure 2-4 on page 16.

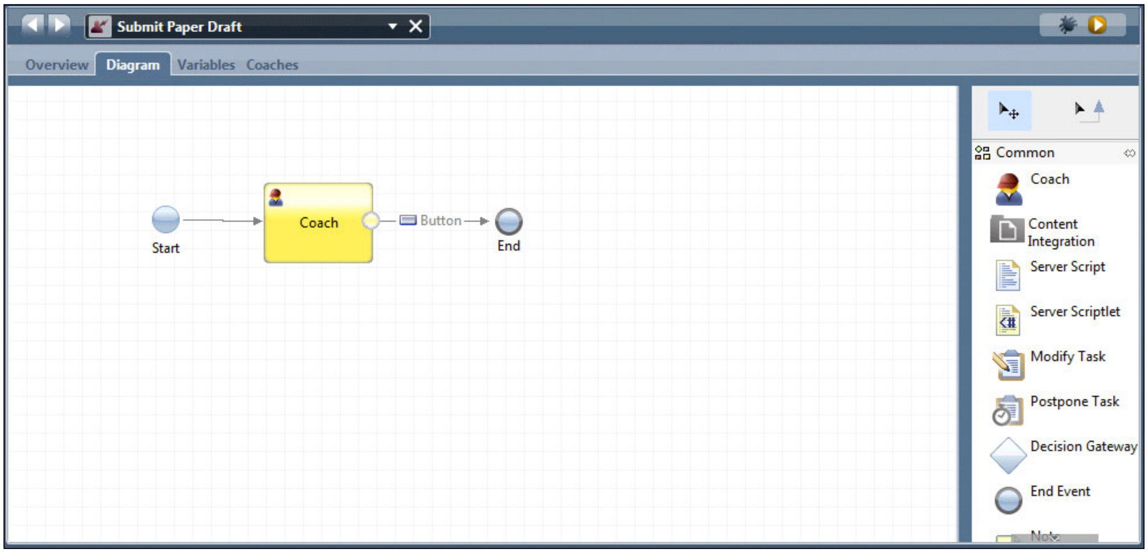


Figure 2-4 Default implementation of Submit Paper Draft Human Service

You can now press the run button to run the service as shown in Figure 2-5.

Figure 2-5 Run button

A browser window opens and shows an empty form with a button as shown in Figure 2-6 on page 17.

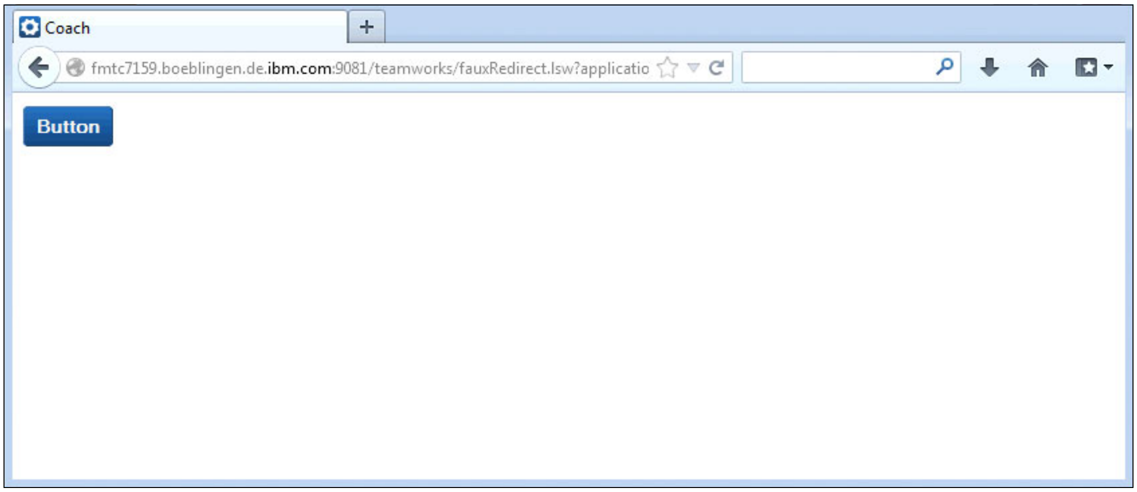

Coach
fmtc7159.boeblingen.de.ibm.com:9081/teamworks/fauxRedirect.lsw?applicatio
Button

Figure 2-6 Default coach implementation

Clicking the button closes the form and the text “The service has finished.” is displayed.

You have just built and run your first task completion UI!

2.2 Building a simple coach

So now that you have everything in place, you can start working on designing your first coach. To do that, open the Submit Paper Draft service that we created in the previous chapter and go to the *Coaches* tab to open the Coach editor as shown in Figure 2-7 on page 18.

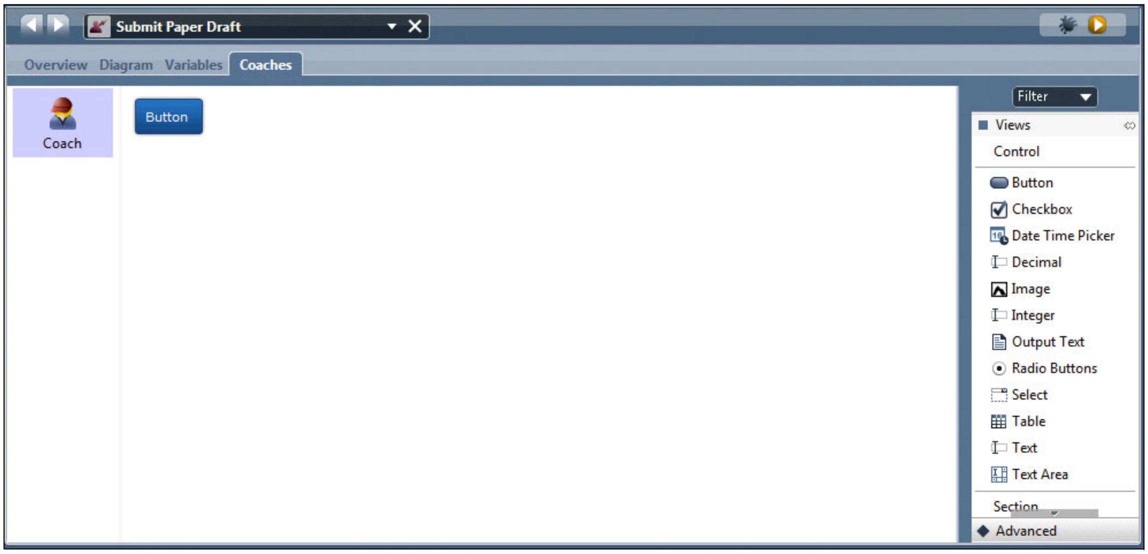


Figure 2-7 Default Coach in Submit Paper Draft Human Service

On the left side, you see the list of Coaches in this human service. At the moment there is only one named Coach—the one that has been created for you by the Activity wizard.

In the middle, you see the actual canvas of the Coach form. At the moment, there is a lot of white space because you have not started to lay out the UI.

On the right side, you see the palette of the Coach editor. By default it shows the stock controls provided as part of product. Stock controls are Coach Views that fall into the categories Controls and Sections. Controls are the actual controls like buttons and entry fields that later on allow users to interact with the form. Sections allow structuring the controls on the form. By default horizontal sections, vertical sections and tabs are provided. Dragging any of the coach views to the canvas in the middle adds it to the form.

Note: To learn more about stock controls refer to the following section in the information center:

`http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/buildcoach/topics/rstockcontrols.html`

Other elements on the palette are Variables and Advanced elements. Advanced allows to add custom HTML to a form, for example to add a company-specific banner or other HTML markup. Variables list the variables that have been defined for the human service. They represent the data of the human service. Dragging a variable to the canvas results in a matching coach view being selected and added to the coach, while a binding to the corresponding variable is created. We take a closer look at coach views and their bindings later in this book.

To make the previously introduced basic scenario a bit more interesting, we extended the Publish Paper process with a few more activities and lanes, describing a simple approval process where first Authors and Reviewers work on several drafts of a paper, before seeking approval, as shown in Figure 2-8.

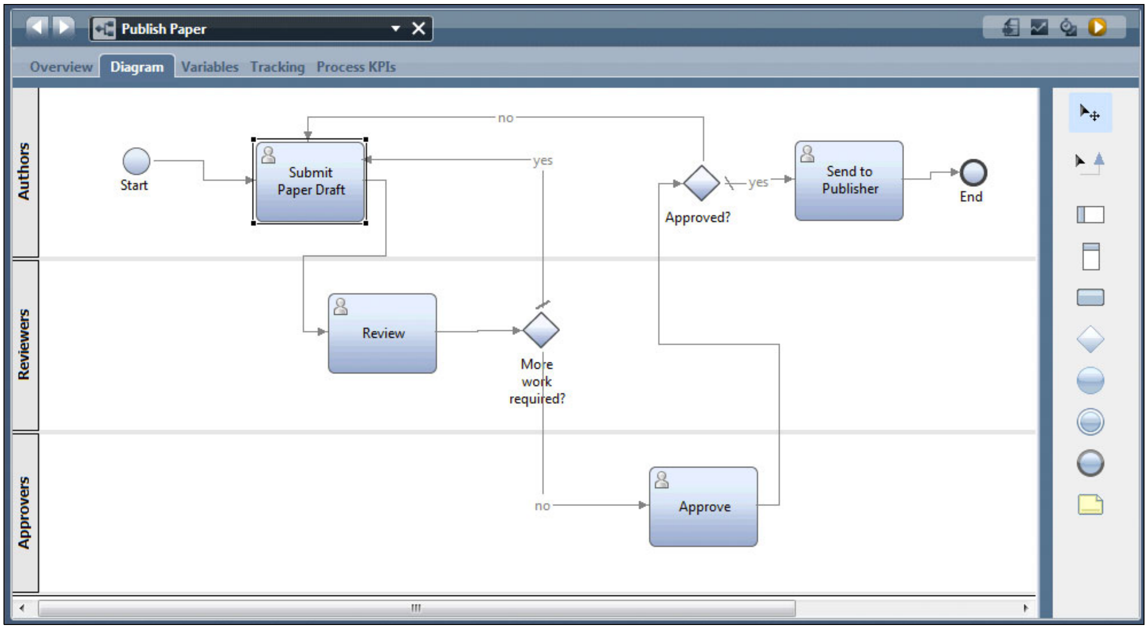


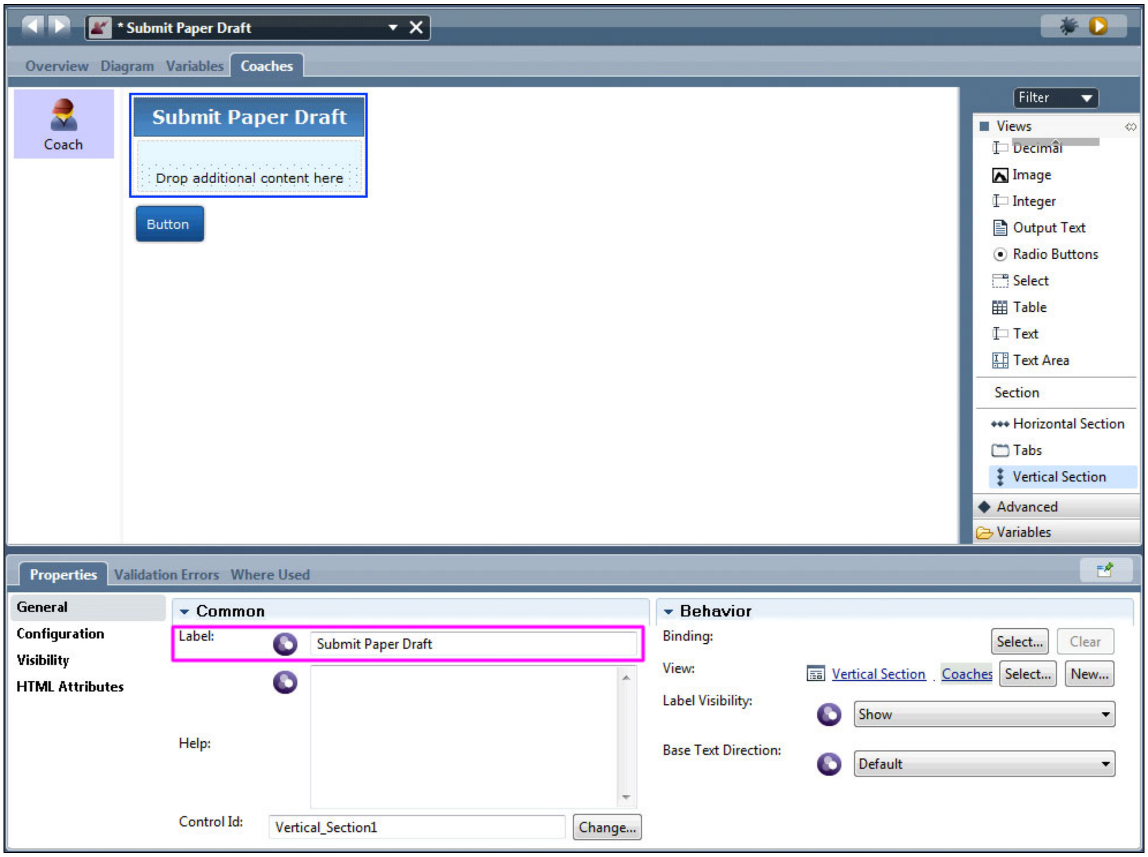
Figure 2-8 Evolved version of Publish Paper process

We also added variables to the picture. Initially the Submit Paper Draft activity deals with the following data:

- A string that describes the location where the paper can be found. We replace that with a real list of documents later
- A list of review comments
- A flag that indicates if the paper is ready for approval

Now we use Coach editor to create a Coach form for the Submit Paper Draft activity:

1. First, add a vertical section as an overall container for the coach and give it a name as shown in Figure 2-9.



Label: Submit Paper Draft

Help:

Control Id: Vertical_Section1

Binding:

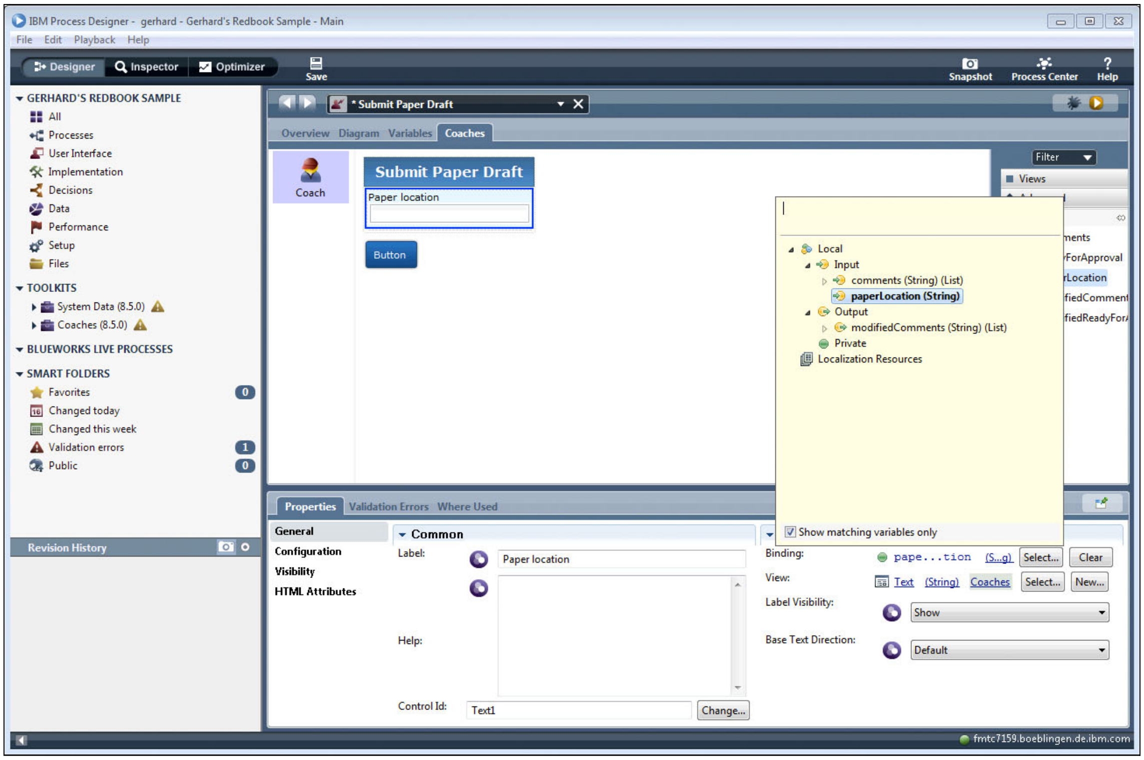
View: Vertical Section . Coaches

Label Visibility: Show

Base Text Direction: Default

Figure 2-9 Adding a vertical section

2. Next, add a text entry field for the “paperLocation” variable by dragging the corresponding variable into the previously added vertical section.



Label: Paper location

Control Id: Text1

Binding: pape...tion (S...g)

View: Text (String) Coaches

Label Visibility: Show

Base Text Direction: Default

Figure 2-10 Adding a text entry field for paperLocation variable

We see on the coach form in Figure 2-10 that a text entry field has been added, and in the properties section of the coach view we can observe that the Text Coach View has been automatically selected based on the variable type, and that the paperLocation input variable has been set as the Binding.

Binding is an important concept of Coach Views. Coach view and variable are coupled via the Binding. When the user changes the data of a coach view, for example by entering text, then the variable bound via the Binding is automatically updated. The change is also propagated to all other coach views that are bound to the same variable.

3. Next, drag the variable `readyForApproval` from the Variables section of the Coach editor menu to the canvas underneath the Paper Location field. Based on the type of the variable (Boolean) the Check Box coach view is selected as shown in Figure 2-11.

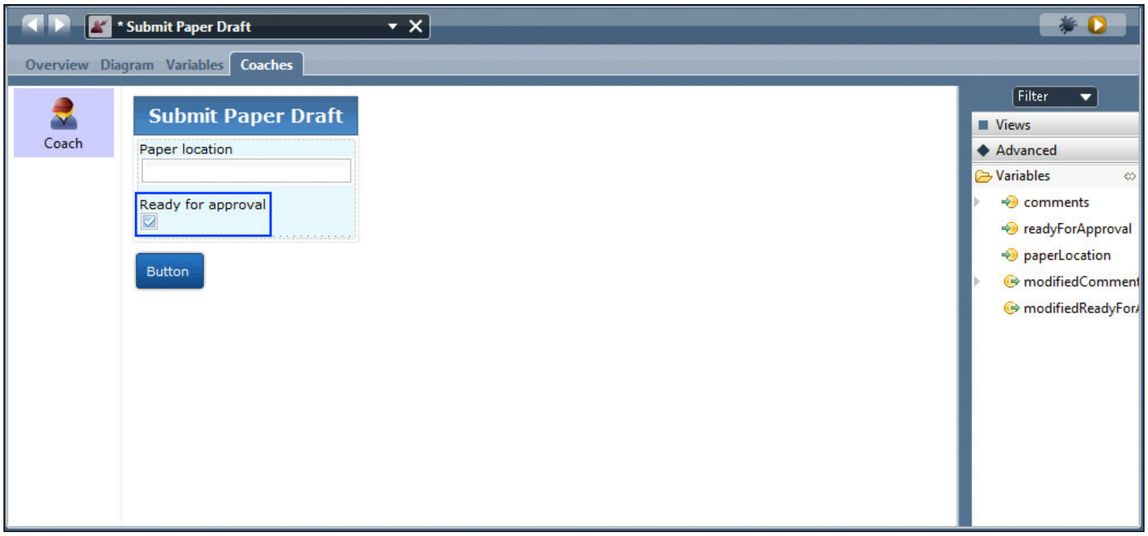
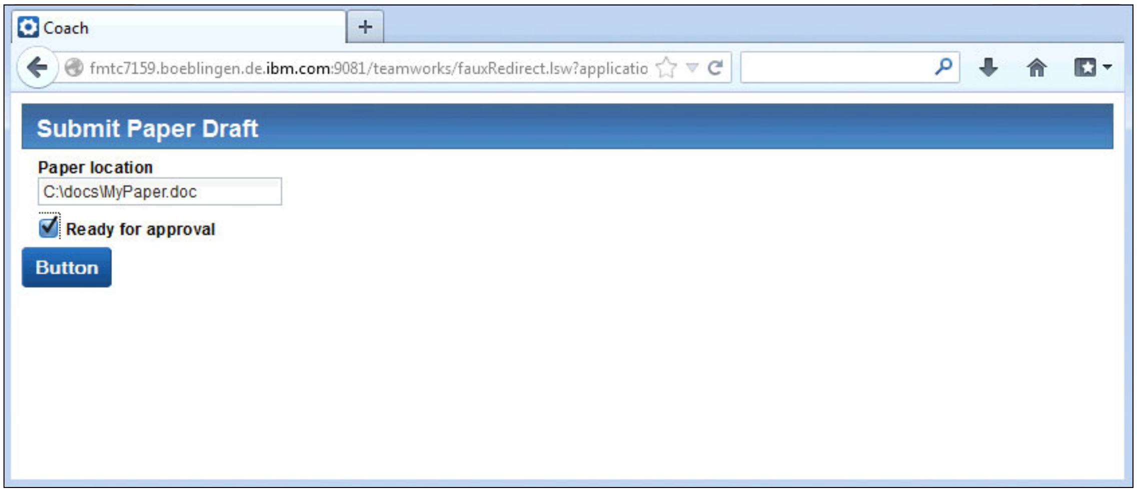


Figure 2-11 Adding check box for readyForApproval variable

4. Before adding support for lists of comments in the next chapter, we take a short break and run the Coach in playback mode to see how things look like in reality, as shown in Figure 2-12.


Coach
fmtc7159.boeblingen.de.ibm.com:9081/teamworks/fauxRedirect.lsw?applicatio
Submit Paper Draft
Paper location
Ready for approval
Button

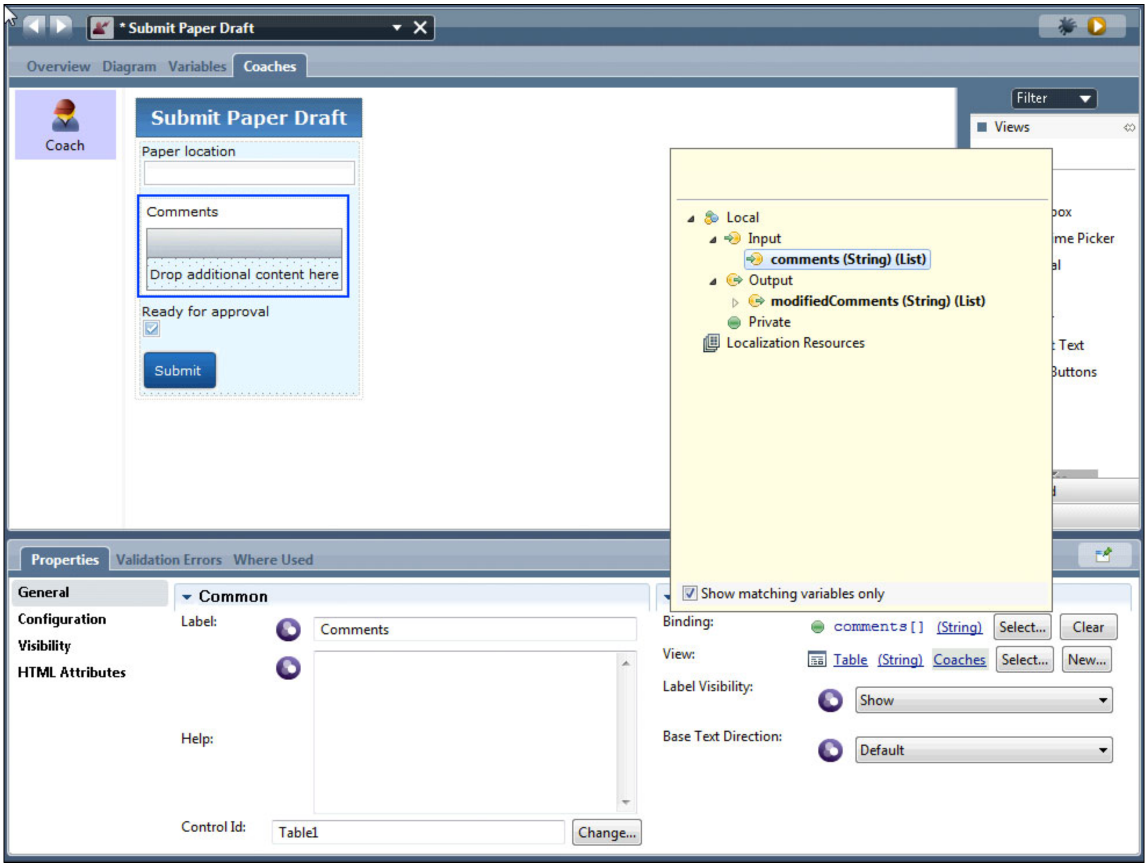
Paper location: C:\docs\MyPaper.doc

Figure 2-12 Running Submit Paper Draft Human Service

This looks promising already. Next, we move the “Button” that was created for us by default into the Submit Paper Draft vertical section that we added, and give it a more sensible name. And then we implement the UI for the list of comments.

2.3 Configuring coach views

Before we look at configuring coach views, we continue with the example. We want to be able to display, add, and remove comments from a list of comments. The UI control that we use to do that is the Table coach view. We add a Table coach view and assign it a label, Comments. Next, configure it to bind to the comments variable by clicking the **Select** button next to Binding and by selecting comments in the pop-up dialog that is displayed as shown in Figure 2-13.

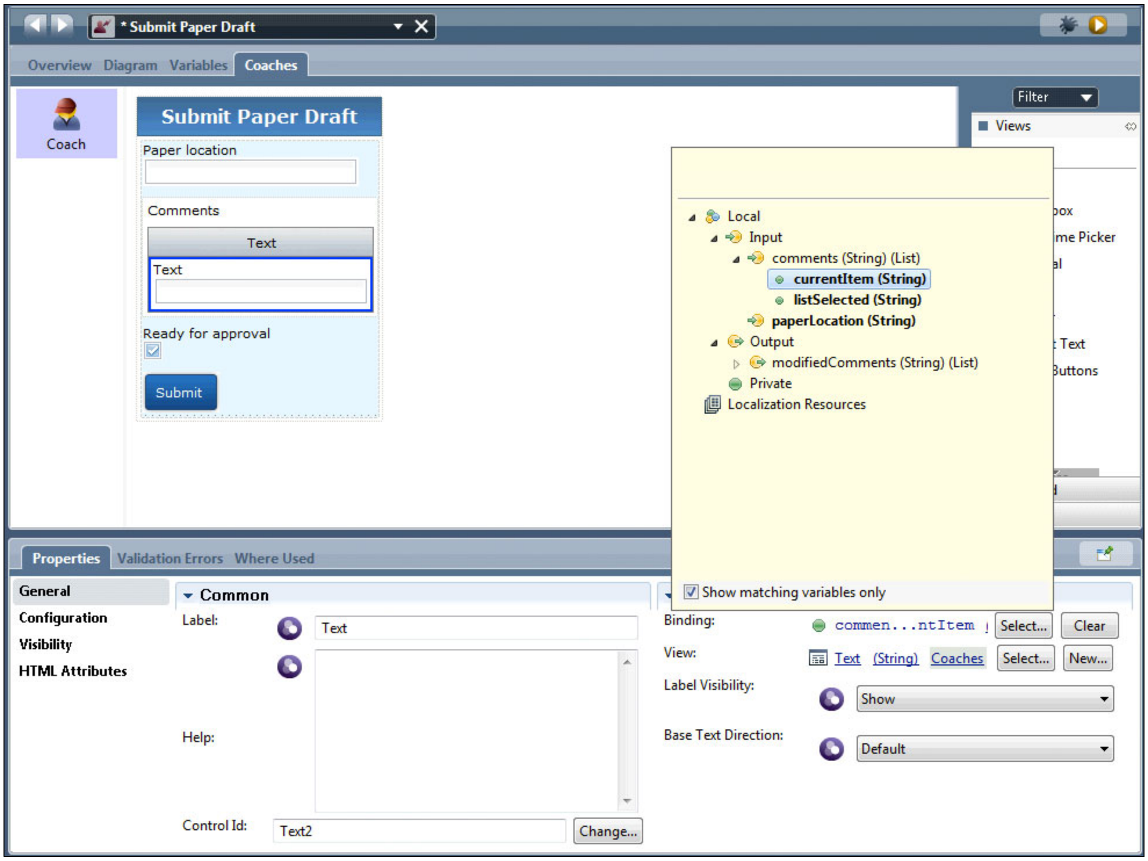


Label: Comments
Help:
Control Id: Table1 Change...
Binding: comments[] (String) Select... Clear
View: Table (String) Coaches Select... New...
Label Visibility: Show
Base Text Direction: Default

Figure 2-13 Defining the Binding for comments

What we have to do then is to add coach views to display and edit the columns of the table. In this case, the only column is the comment string, therefore all we have to do is to drop a Text coach view into the table and wire it up with the current item in the comments list to which the enclosing table is bound. We do that by selecting `currentItem` underneath comments. `currentItem` is a pseudo variable that represents the list iterator. The second pseudo variable is `listSelected`. We would use it if we wanted to refer to the currently selected item in the list.

After defining the binding for the table, we do the same for the Text control we have put inside the table. Figure 2-14 illustrates this. The Text coach view is bound to comments → currentItem.



Label: Text

Help:

Control Id: Text2 Change...

Binding: commen...ntItem Select... Clear

View: Text (String) Coaches Select... New...

Label Visibility: Show

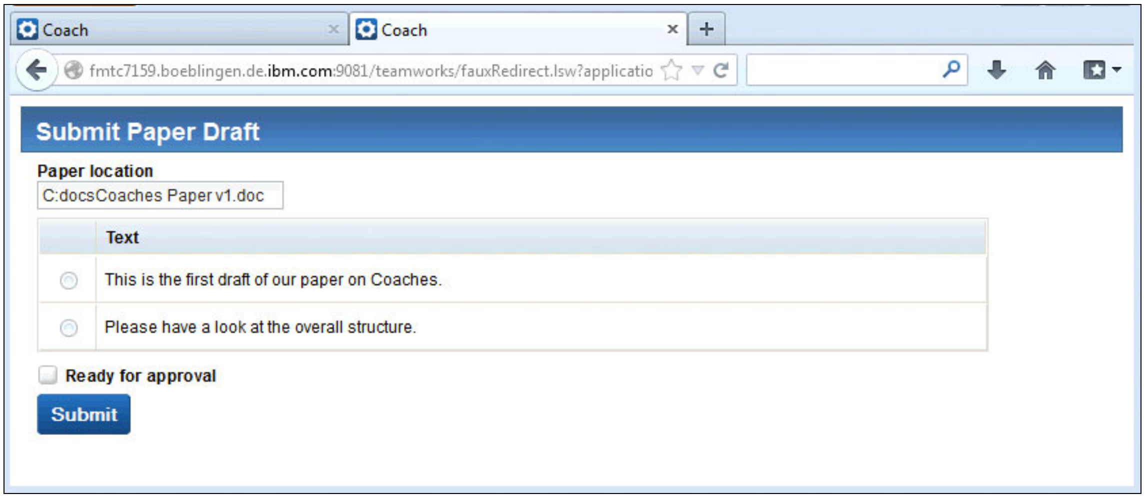
Base Text Direction: Default

Figure 2-14 Defining the binding for the Text Coach View in the table

Next, test-drive the coach by clicking the run button to run the service as shown in Figure 2-15.

Figure 2-15 Run button

Here is how the Coach is rendered using sample data as shown in Figure 2-16.


Coach
Coach
fmtc7159.boeblingen.de.ibm.com:9081/teamworks/fauxRedirect.lsw?applicatio

Submit Paper Draft

Paper location
C:docsCoaches Paper v1.doc

	Text
○	This is the first draft of our paper on Coaches.
○	Please have a look at the overall structure.

☐ Ready for approval

Submit

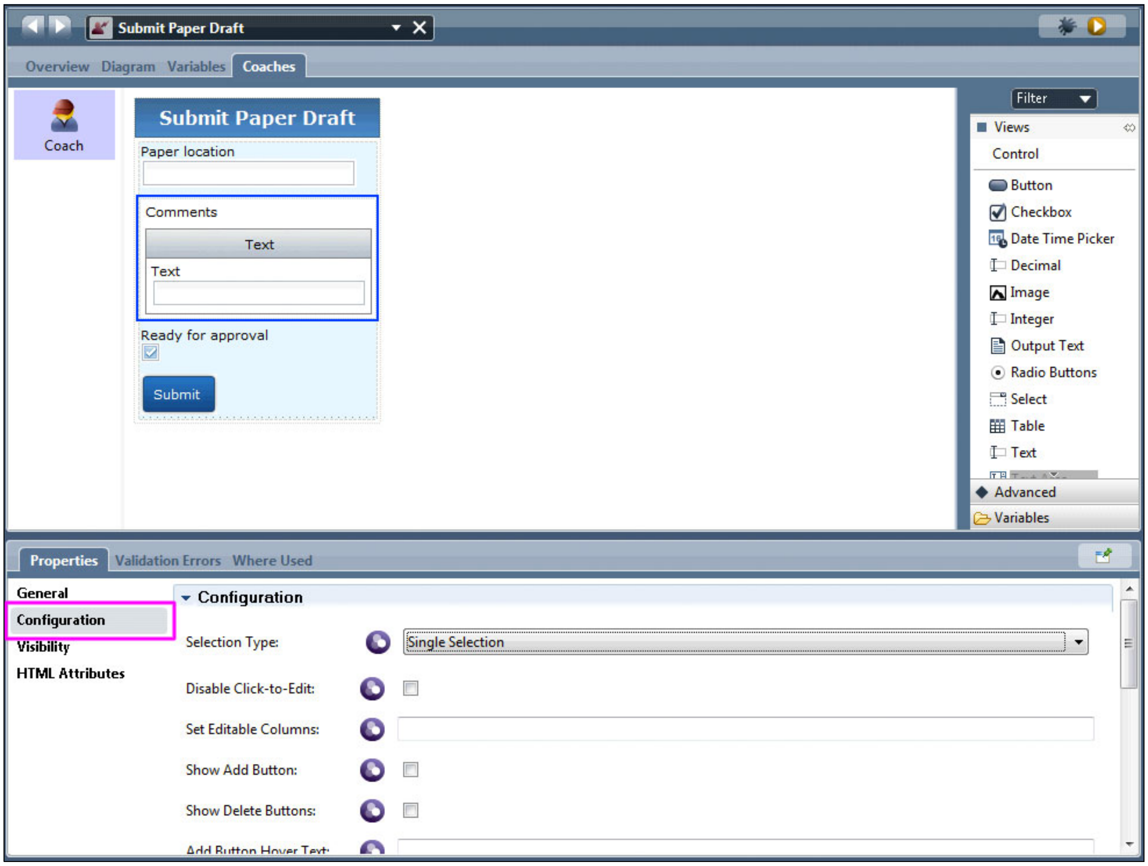
Figure 2-16 Rerunning Submit Paper Draft Human Service

Sample data: The sample data used in this chapter can be downloaded as Additional Material from the IBM Redbooks abstract page. Check Appendix A, "Additional material" on page 357.

Although this looks promising, the table has two issues:

- It allows us to select a line, which is not required in our use case
- It does not allow us to add or remove comments

Let us see if we can configure the Table coach view in a way that addresses these issues. To configure a coach view, select it and then click the Configuration tab in the Properties section as shown in Figure 2-17 on page 26.



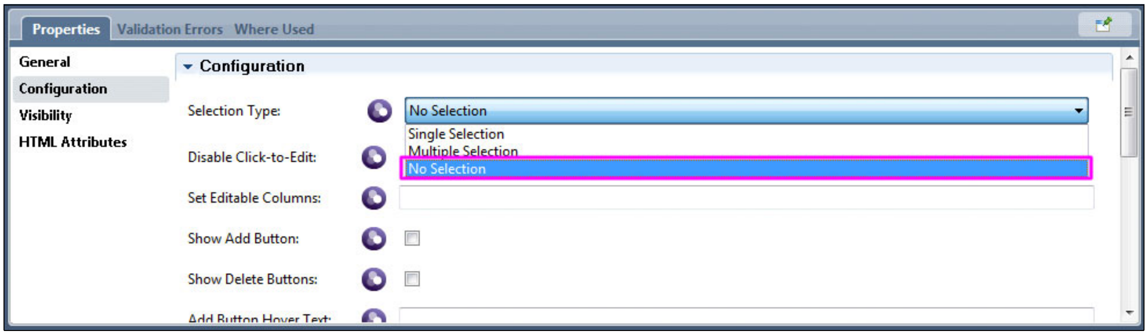
Selection Type: Single Selection
Disable Click-to-Edit:
Set Editable Columns:
Show Add Button:
Show Delete Buttons:
Add Button Hover Text:

Figure 2-17 Editing the configuration of the Table control

Displayed under Configuration are the configuration properties of the selected coach view, as shown in Figure 2-17 in the Table coach view. By manipulating configuration properties, the behavior of a coach view can be customized.

Configuration options: Configuration options are another important concept of coach views. Two kinds of configuration options exist: Objects and Services. Configuration options of type *Object* allow specifying configuration data for the coach view like showing a particular button as in Figure 2-17. Configuration options of type Service are used to bind the coach view to services it can use at run time to perform certain tasks like retrieving data or performing data validation. The provider of the coach view defines configuration options. When using the coach view, a Configuration property sheet is automatically generated based on the configuration option definitions.

To address the issues spotted before, we are now configuring the table view. To make sure the selection option goes away, we change the Selection Type to **No Selection** as shown in Figure 2-18.

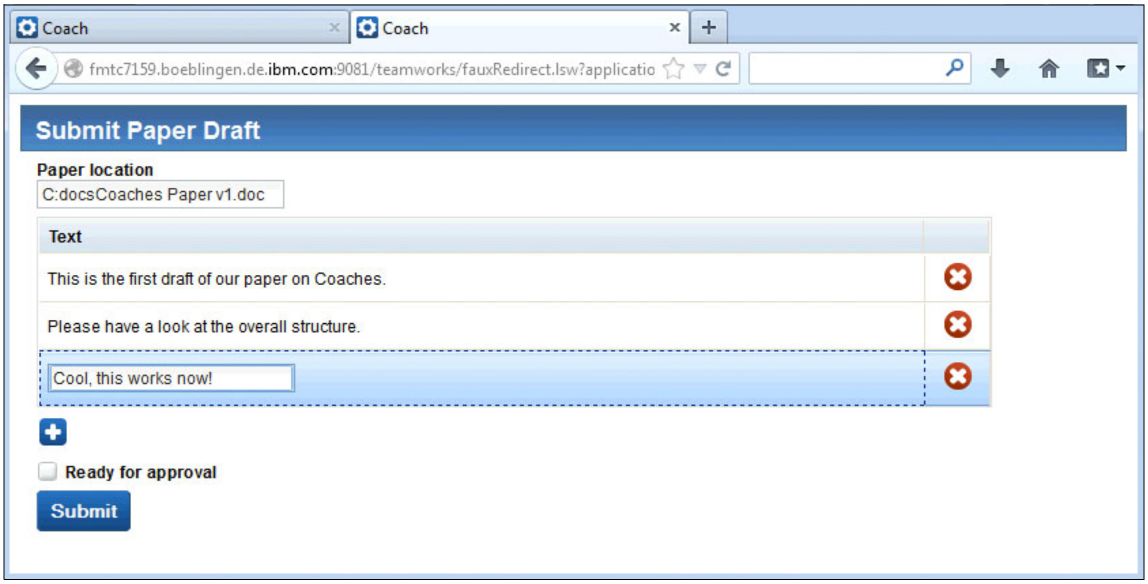


Selection Type: No Selection
Single Selection
Multiple Selection
No Selection
Disable Click-to-Edit:
Set Editable Columns:
Show Add Button:
Show Delete Buttons:
Add Button Hover Text:

Figure 2-18 Changing Selection Type to No Selection

To allow us to add and remove comments, we also select the check boxes for Show Add Button and Show Delete Buttons, respectively.

Rerunning the Coach now produces a result that is far closer to our expectations, as it can be seen in Figure 2-19.


Coach
Coach
fmtc7159.boeblingen.de.ibm.com:9081/teamworks/fauxRedirect.lsw?applicatio

Submit Paper Draft

Paper location
C:docsCoaches Paper v1.doc

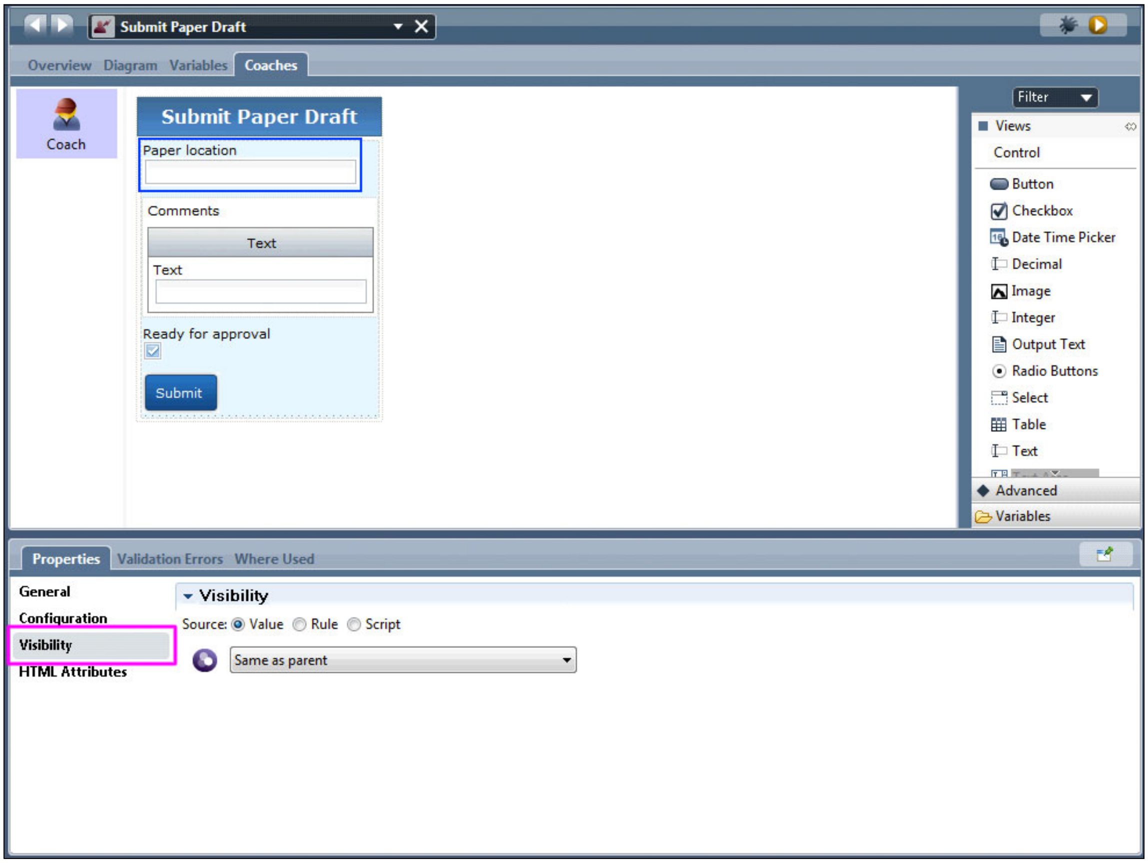
Text
This is the first draft of our paper on Coaches.
Please have a look at the overall structure.
Cool, this works now!

Ready for approval

Submit

Figure 2-19 Yet another run of the Submit Paper Draft Human Service

Another powerful capability of Coaches is the support for configuring the *visibility* of elements on the coach. A coach view's visibility can be defined by selecting the control, and then clicking the Visibility tab in the Properties section.



Visibility

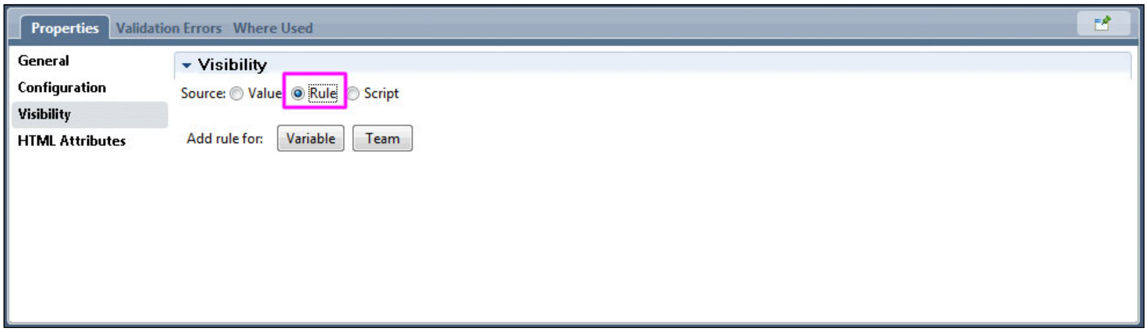
Source: [✓ Value] Rule Script

Same as parent

Figure 2-20   Default visibility of Paper location control

As you can see in Figure 2-20, by default the visibility of a coach view is the same as its parent. The parent in this case is the Coach, which means that the coach view is visible. That is in line with our observation from the previous test run.

Now what we want to do is to modify the visibility of the Paper location entry field based on the value of the Ready for approval check box. When that check box is checked, we consider the paper version final, and thus do not allow further changes of its location. We accomplish that by defining a rule for the visibility of the Paper location coach view by clicking the Rule radio button as shown in Figure 2-21 on page 29.



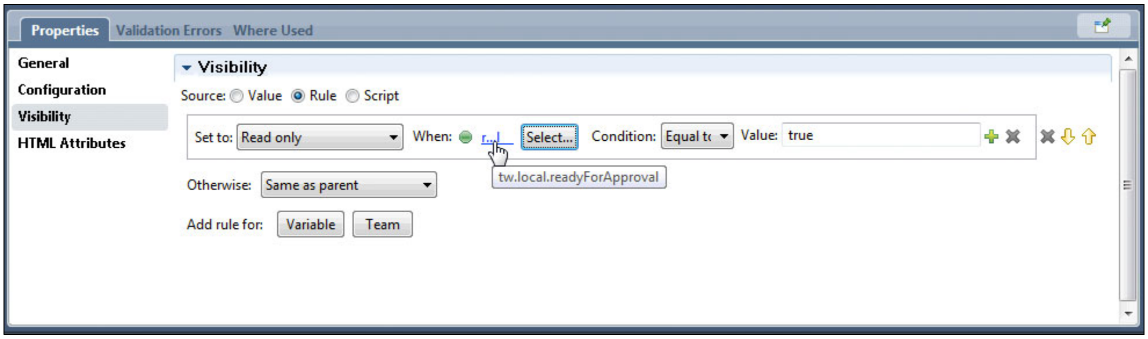
Source: Value [✓ Rule] Script

Add rule for: Variable Team

Figure 2-21 Switching to rule based visibility

We can now add visibility rules that depend on variable values or team membership. In this case, we define a variable rule that makes the Paper location control read only if “Ready for approval” is checked (in which case its value is `true`). We do that by clicking **Variable** to add a variable-based visibility rule. We select the target visibility state “Read only” for the rule and we select the variable “`readyForApproval`” as the variable that this rule depends on. We leave the condition as “Equal to” and set the value we compare with to “true” as shown in Figure 2-22.

If the rule does not apply, some default behavior occurs. The default behavior is defined by the Otherwise rule block, which by default assigns the control the same visibility of the coach view’s parent. Leave it that way.



Source: Value [✓ Rule] Script

Set to: Read only When: r...l Condition: Equal to Value: true

tw.local.readyForApproval

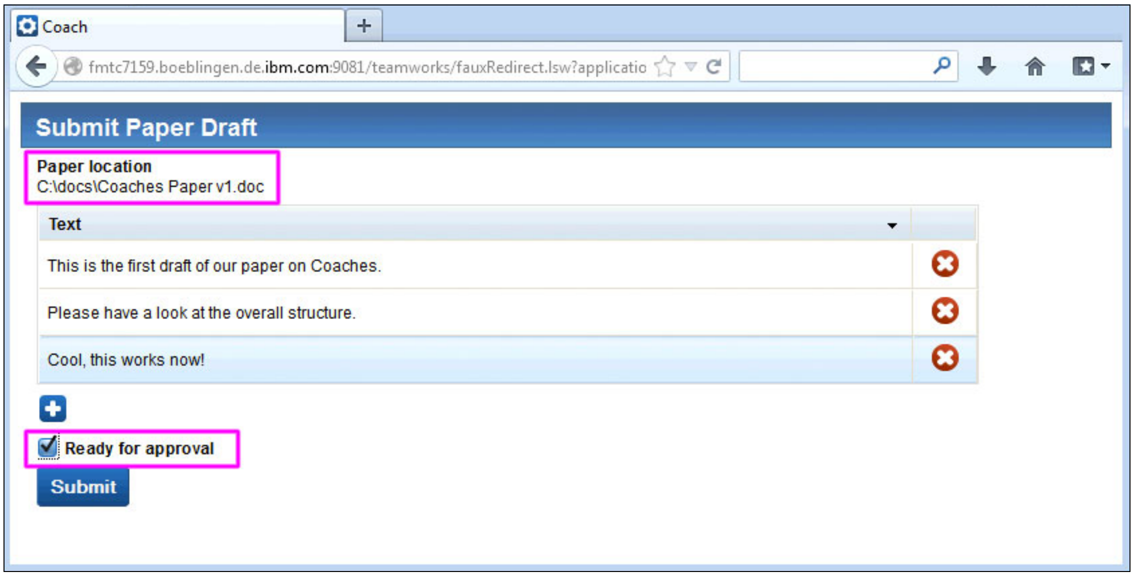
Otherwise: Same as parent

Add rule for: Variable Team

Figure 2-22 Defining the visibility rules for Paper location

We could now add additional visibility rules for teams and variables to manage visibility even more fine grained, but we leave things as they are for now and instead do another test run.

As you can see from Figure 2-23, what we were trying to accomplish works out nicely. Checking and unchecking Ready for approval disables and enables Paper location respectively.


Coach
fmtc7159.boeblingen.de.ibm.com:9081/teamworks/fauxRedirect.lsw?applicatio
Submit Paper Draft
Text
Ready for approval
Submit

Paper location
C:\docs\Coaches Paper v1.doc

This is the first draft of our paper on Coaches.

Please have a look at the overall structure.

Cool, this works now!

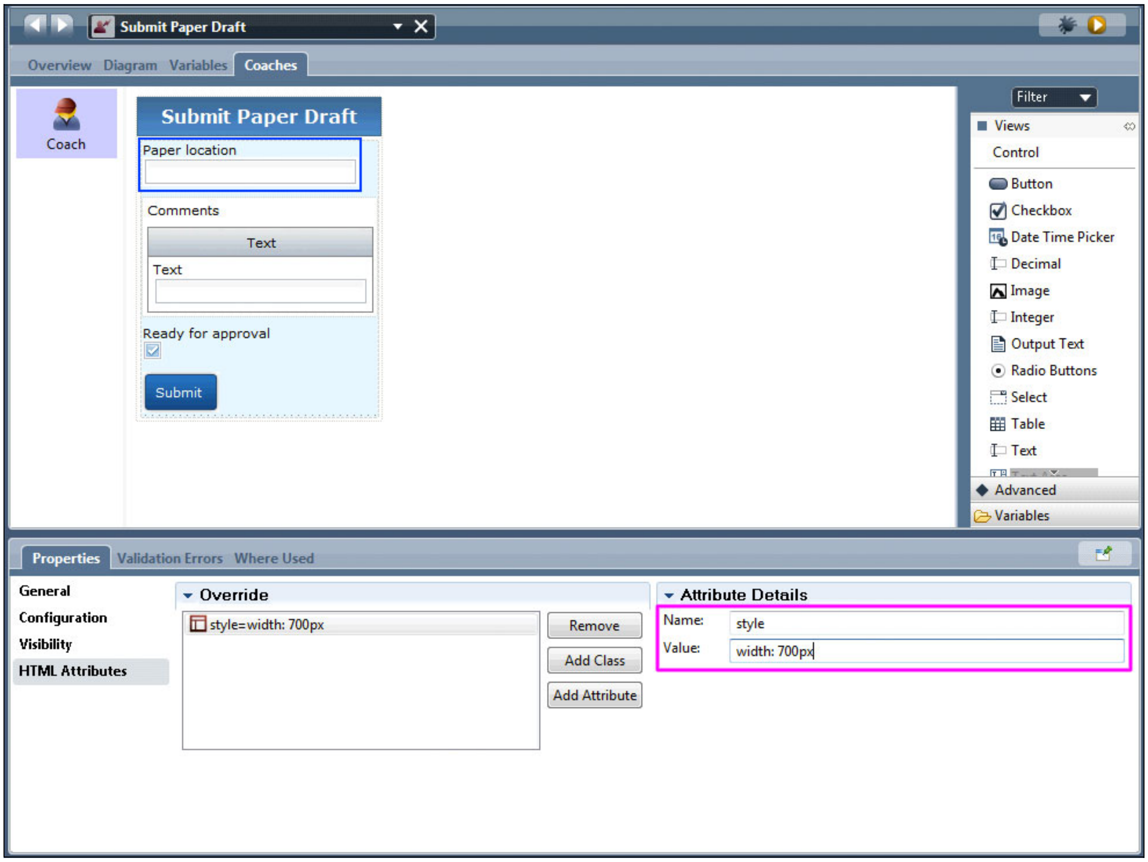
Figure 2-23 Rerunning Submit Paper Draft to test visibility rules

By the way, looking at the test result again, have you wondered why we were using such a short file name for the paper so far? This is mostly because the default width of the Text coach view allows for entering longer text, but it will not display it all. As we have enough space on our form, and because we are envisioning longer file names to be entered, we aim to reconfigure the Text coach view to use more horizontal pixels. Unfortunately the Text control does not allow us to define its width as a configuration option. Therefore, we use some advanced configuration techniques that are introduced in the next chapter.

2.4 Advanced configuration

The methods introduced in this chapter require a bit of knowledge about HTML and CSS. Actually, less knowledge than what you might think now. One of the coach views Properties sections we have not yet looked at is HTML Attributes. In HTML Attributes, we can define CSS attributes and CSS classes for the coach view. To be precise, what we define there is added to the HTML <div> element for the coach view. Therefore, to set the width of the Text coach view we are adding an attribute *style* to set the width to 700 pixels.

To do that we select the Paper location control and then click HTML Attributes on the Properties tab. There we click the Add Attribute button and complete the entry fields as demonstrated in Figure 2-24.



style=width: 700px

Name: style

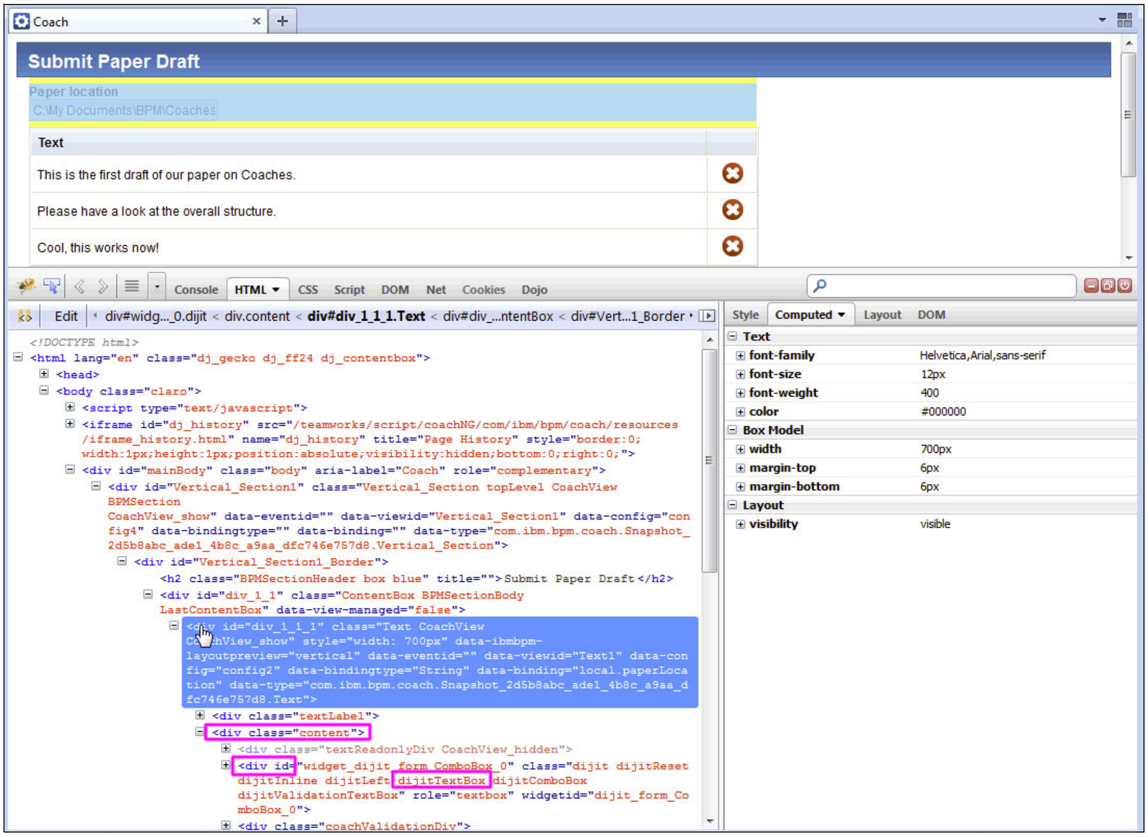
Value: width: 700px

Figure 2-24 Adding a style attribute to Paper location

Unfortunately when we test the change, the result is a bit disappointing. Setting the style attribute on the Text coach view alone has no visible effect.

Now it is time to slightly extend our tools set and to debug the Coach. We need to see what the HTML markup looks like to get an indication of what else needs to be done. We use the Firebug plug-in that is available for Firefox to continue our exploration. We launch the Coach again and turn on Firebug. In Firebug, we click the HTML tab and look at the HTML code.

We go down starting from the topmost <div> element and keep expanding the child elements. We locate the <div> that corresponds to our Text coach view by clicking each <div> until we see the corresponding element in the browser view above getting highlighted. In Figure 2-25 we have found and selected that <div>. We see that the attribute we have defined (width: 700px) got properly applied. Now all we must do is to make sure that the contained elements also benefit from that change. The easiest way to do this is to tell these elements to use 100% of the available horizontal space. This can be done by assigning an attribute *width* with a value of *100%* to these elements.



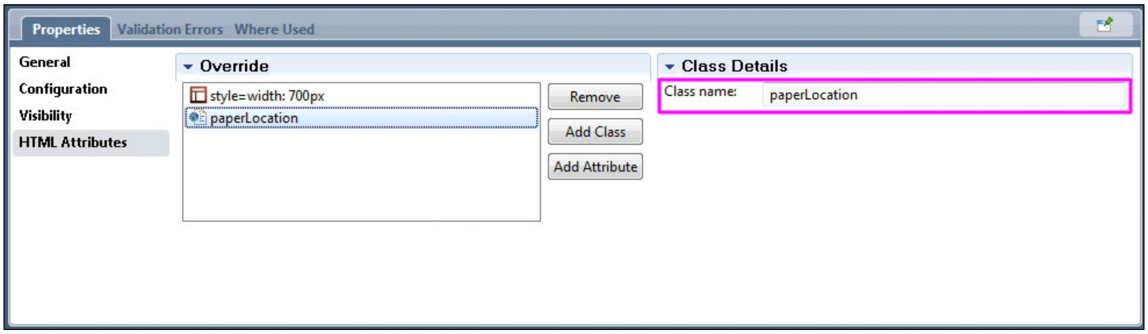
Coach
Submit Paper Draft
Paper location
C:\My Documents\BPM\Coaches
Text
This is the first draft of our paper on Coaches.
Please have a look at the overall structure.
Cool, this works now!
Console
HTML
CSS
Script
DOM
Net
Cookies
Dojo
Edit
div#widg..._0.dijit < div.content < div#div_1_1_1.Text < div#div_...ntentBox < div#Vert...1_Border
Style
Computed
Layout
DOM

```
<!DOCTYPE html>
<html lang="en" class="dj_gecko dj_ff24 dj_contentbox">
  <head>
  <body class="claro">
    <script type="text/javascript">
    <iframe id="dj_history" src="/teamworks/script/coachNG/com/ibm/bpm/coach/resources
    /iframe_history.html" name="dj_history" title="Page History" style="border:0;
    width:1px;height:1px;position:absolute;visibility:hidden;bottom:0;right:0;">
    <div id="mainBody" class="body" aria-label="Coach" role="complementary">
      <div id="Vertical_Section1" class="Vertical_Section topLevel CoachView
      BPMSection
      CoachView_show" data-eventid="" data-viewid="Vertical_Section1" data-config="con
      fig4" data-bindingtype="" data-binding="" data-type="com.ibm.bpm.coach.Snapshot_
      2d5b8abc_ade1_4b8c_a9aa_dfc746e757d8.Vertical_Section">
        <div id="Vertical_Section1_Border">
            <h2 class="BPMSectionHeader box blue" title="">Submit Paper Draft</h2>
          <div id="div_1_1" class="ContentBox BPMSectionBody
          LastContentBox" data-view-managed="false">
            <div id="div_1_1_1" class="Text CoachView
            CoachView_show" style="width: 700px" data-ibmbpm-
            layoutpreview="vertical" data-eventid="" data-viewid="Text1" data-con
            fig="config2" data-bindingtype="String" data-binding="local.paperLoca
            tion" data-type="com.ibm.bpm.coach.Snapshot_2d5b8abc_ade1_4b8c_a9aa_d
            fc746e757d8.Text">
              <div class="textLabel">
              <div class="content">
                <div class="textReadonlyDiv CoachView_hidden">
                <div id="widget_dijit_form_ComboBox_0" class="dijit dijitReset
                dijitInline dijitLeft dijitTextBox dijitComboBox
                dijitValidationTextBox" role="textbox" widgetid="dijit_form_Co
                mboBox_0">
                <div class="coachValidationDiv">
```

Text	
font-family	Helvetica,Arial,sans-serif
font-size	12px
font-weight	400
color	#000000
Box Model	
width	700px
margin-top	6px
margin-bottom	6px
Layout	
visibility	visible

Figure 2-25 Using HTML debugger to examine Coach elements

Now this is easier said than done. There is no direct way in Process Designer to assign attributes to child elements of a coach view. But there is a way, and once you have seen it, it will look natural and easy to you. All we must do is inject the style attributes into the coach and ensure that they get applied to the right elements. We use the Custom HTML construct to do this. Before we can add the custom HTML, we must do one step of preparation though. We need to assign a *class* to the Text coach view to make sure we can find it later, and to ensure that we manipulate the width of the right child elements. Therefore we add a class named “paperLocation” to our Paper location coach view. We do that by clicking the Add Class button in the HTML Attributes section in Properties of the Paper location control, and entering “paperLocation” as Class name as shown in Figure 2-26.

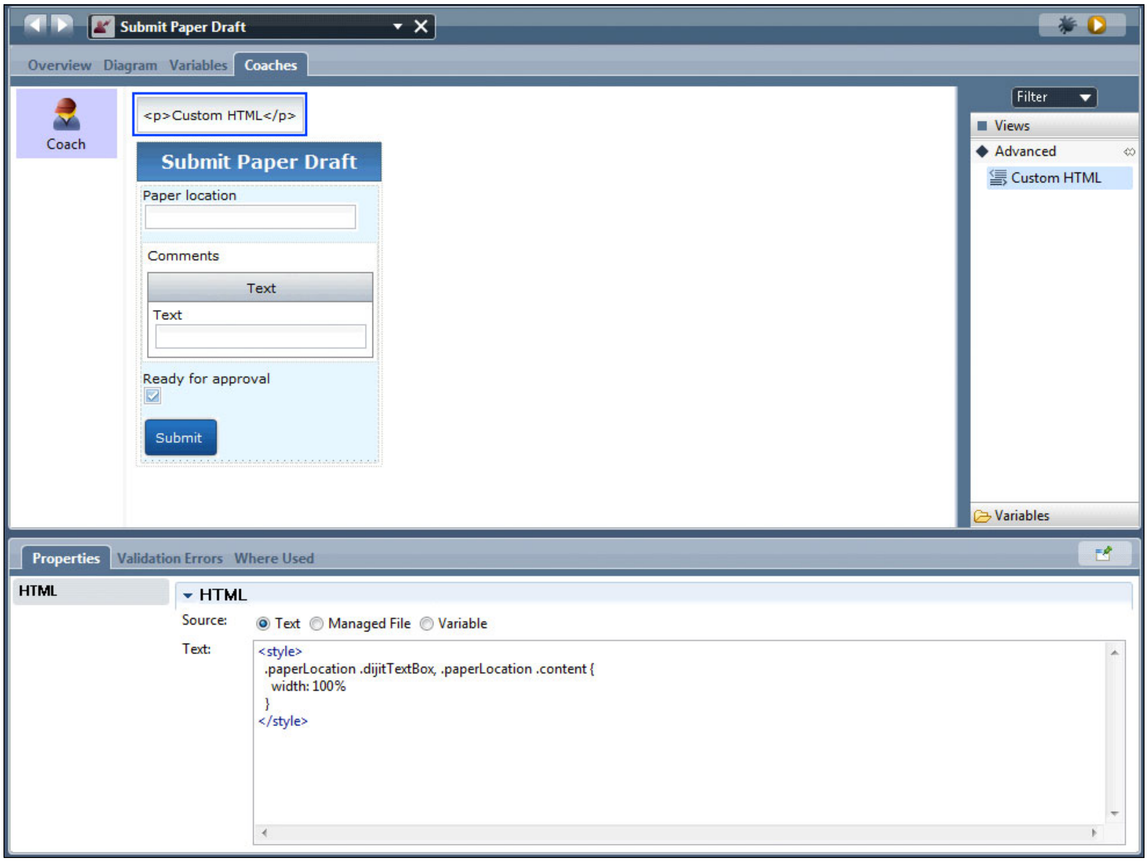


style=width: 700px
paperLocation

Class name: paperLocation

Figure 2-26 Adding paperLocation class to Paper location coach view

With that in place we can now add a Custom HTML element that introduces the style changes we require. Custom HTML can be found in the Advanced section of the palette. Drag it to the top of the Coach and then add the required HTML snippet, as demonstrated in Figure 2-27 on page 34.



Source: [✓ Text] Managed File Variable

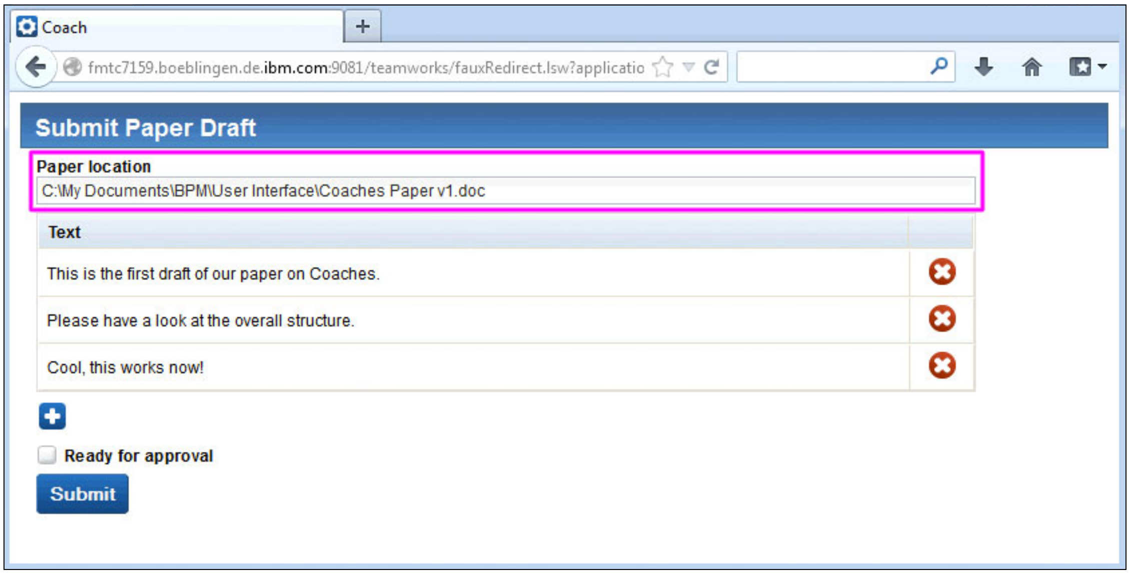
Text:

```
<style>
 .paperLocation .dijitTextBox, .paperLocation .content {
   width: 100%
 }
</style>
```

Figure 2-27 Adding Custom HTML section to Coach

In the HTML snippet, we add a <style> element that searches for classes *dijitTextBox* and *content* and assigns these classes an attribute *width:* with a value of *100%*.

With these changes applied, it is time to test our coach again. Running the coach now produces the following output as demonstrated in Figure 2-28 on page 35.


Coach
fmtc7159.boeblingen.de.ibm.com:9081/teamworks/fauxRedirect.lsw?applicatio

Submit Paper Draft

Paper location

C:\My Documents\BPM\User Interface\Coaches Paper v1.doc

Text
This is the first draft of our paper on Coaches.
Please have a look at the overall structure.
Cool, this works now!

Ready for approval

Submit

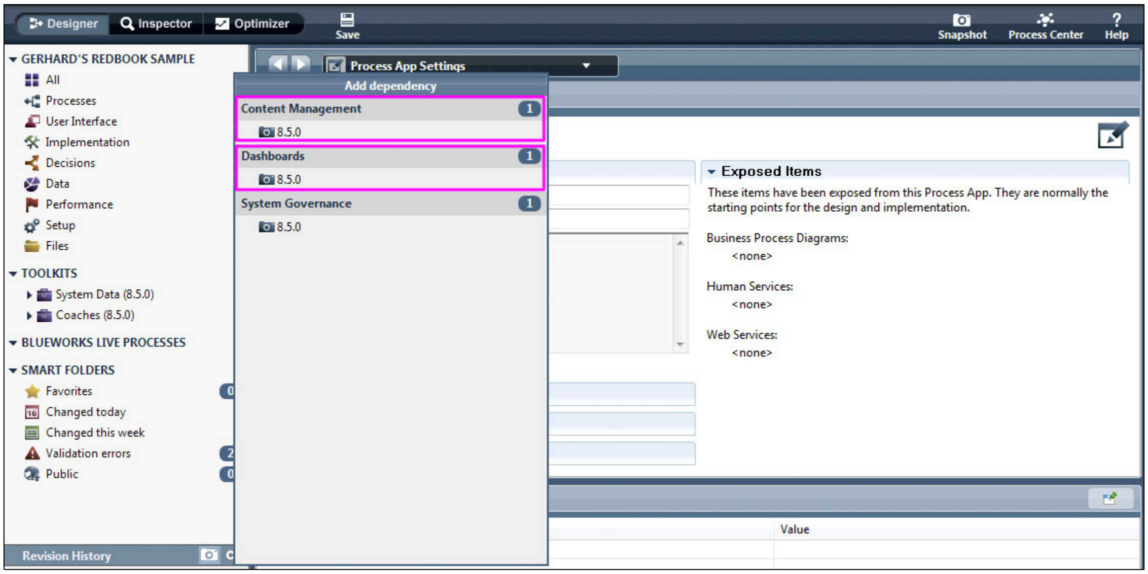
Figure 2-28 Rerunning Submit Paper Draft to test the effectiveness of style changes

As you can see, everything worked out as planned and our Coach looks more beautiful than ever. Note that by looking at the width of the Text coach view we picked a simple use case. The methodology introduced in this chapter can be applied for all sorts of style-based customizations. In many cases, applying these simple manipulations will save you from creating your own custom coach views. These CSS class names are not part of the formal Business Process Manager (BPM) API and are subject to change from release to release.

2.5 Coach view toolkits

In the previous sections, we learned about the full set of out-of-the-box capabilities of Coaches and Coach Views. What we have not looked at yet is other Coach View libraries that are provided as part of the product. There are two more libraries with Coach Views you can make use of: *Content Management* and *Dashboards*. All you need to do to leverage coach views from these libraries is to add a dependency to the corresponding toolkits using the Process Designer library.

First, we add the dependencies to Content Management and Dashboards by clicking the plus icon next to TOOLKITS in the Process Designer main menu. We select Content Management to add a dependency to that toolkit and when we are done with that, we do the same for Dashboards as shown in Figure 2-29 on page 36.



These items have been exposed from this Process App. They are normally the starting points for the design and implementation.

Business Process Diagrams:
<none>

Human Services:
<none>

Web Services:
<none>

Figure 2-29 Adding toolkit dependencies

Next, we open our coach again and change the Filter settings of the palette to show controls from the new categories Content and Dashboard by selecting the corresponding check boxes as highlighted in Figure 2-30.

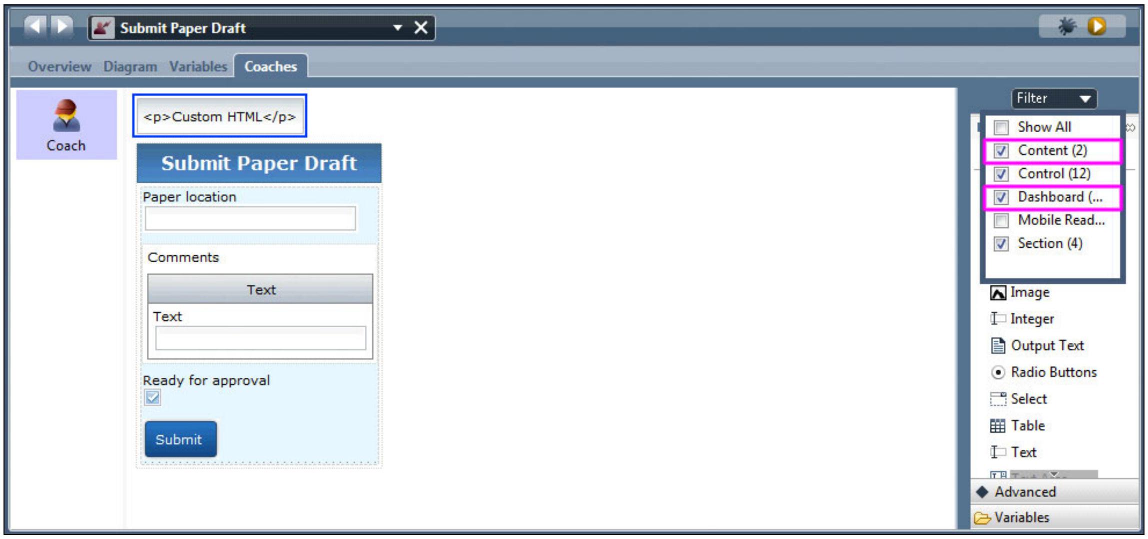


Figure 2-30 Enabling Content controls and Dashboard controls in Coach editor

Now if we scroll through the Coach Views on the palette, we see all the new Coach Views that have been added. Under *Content*, we find the Coach Views for interacting with documents in an Enterprise Content Management system, as well as for business process attachments.

Note: To learn more about content controls, refer to the following section in the information center:

`http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/buildcoach/topics/rstockcontentcontrols.html`

Under *Dashboard*, we find all the new Coach Views introduced in IBM Business Process Manager version 8.5 for creating dashboards. If you are looking at Process Performance Dashboard and Team Performance Dashboard in Process Portal, you get a good overview of the capabilities provided by the Coach Views in the Dashboard category because these out-of-the-box dashboards have been built using the Dashboard controls as shown in Figure 2-31 on page 38. The set of controls includes a chart control that allows visualizing data, controls for task lists and process lists, process diagrams and team summaries, and many more.

Note: To learn more about dashboard controls, see the following section in the information center:

`http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/rdashboardtoolkitcontrolscontainer.html`

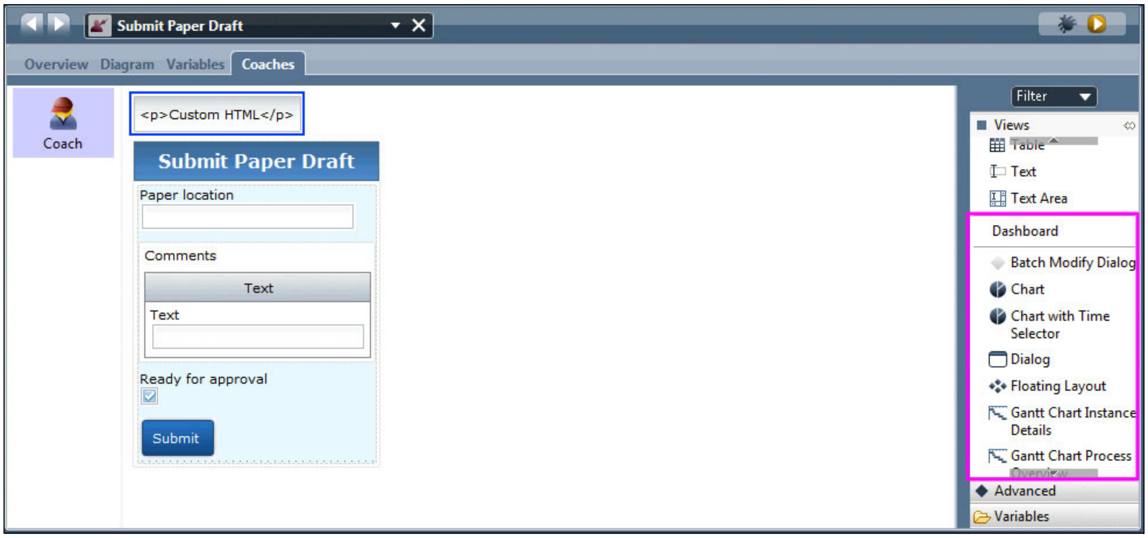
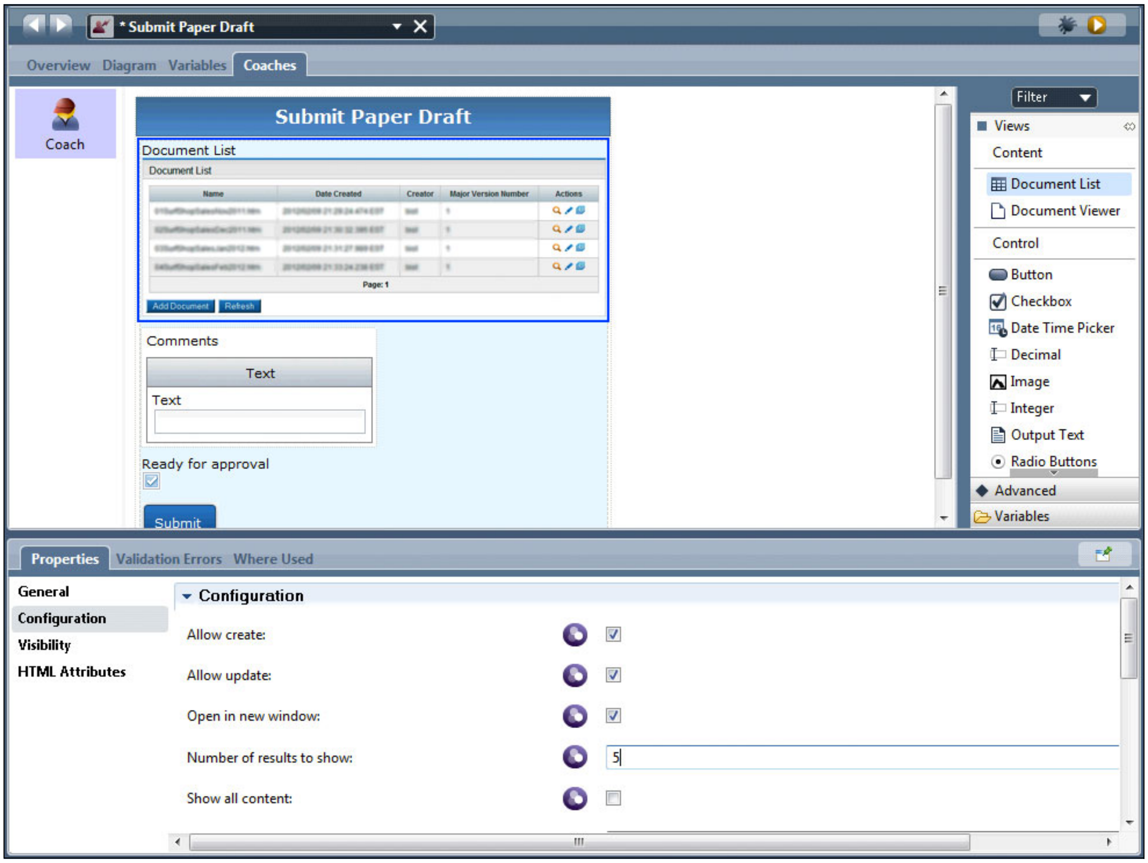


Figure 2-31 Dashboard controls in Coach editor

To continue with our sample from above, we scroll up to the Content section. We want to evolve our sample to not point to a document file name anymore. Instead, we want to be able to attach documents and to store them as part of our Publish Paper process. We do not need the Paper location control anymore and therefore delete it. We also delete the “Custom HTML” coach view from the coach. Instead, we now want to use the Document List control to manage the document versions. To accomplish that, we drag the Document List control from the Coach editor palette and drop it as the topmost control on the Submit Paper Draft vertical section of our coach. The result can be seen in Figure 2-32 on page 39.



Configuration

Allow create: ✓

Allow update: ✓

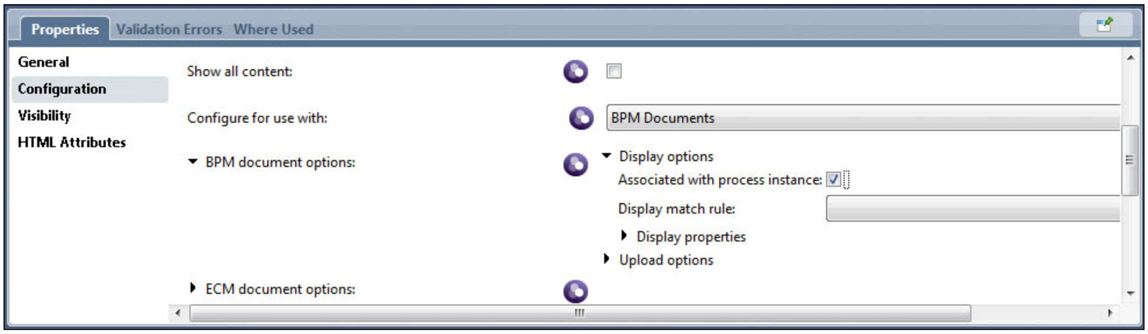
Open in new window: ✓

Number of results to show: 5

Show all content: ☐

Figure 2-32 Submit Paper Draft coach after adding Document List control

As you can see above, we also started to change the configuration properties of the Document List view to fit our needs. Other configuration changes that we made are displayed in Figure 2-33 on page 40.



Show all content: ☐

Configure for use with: BPM Documents

BPM document options:
- Display options
  - Associated with process instance: ☑
  - Display match rule:
  - Display properties
- Upload options

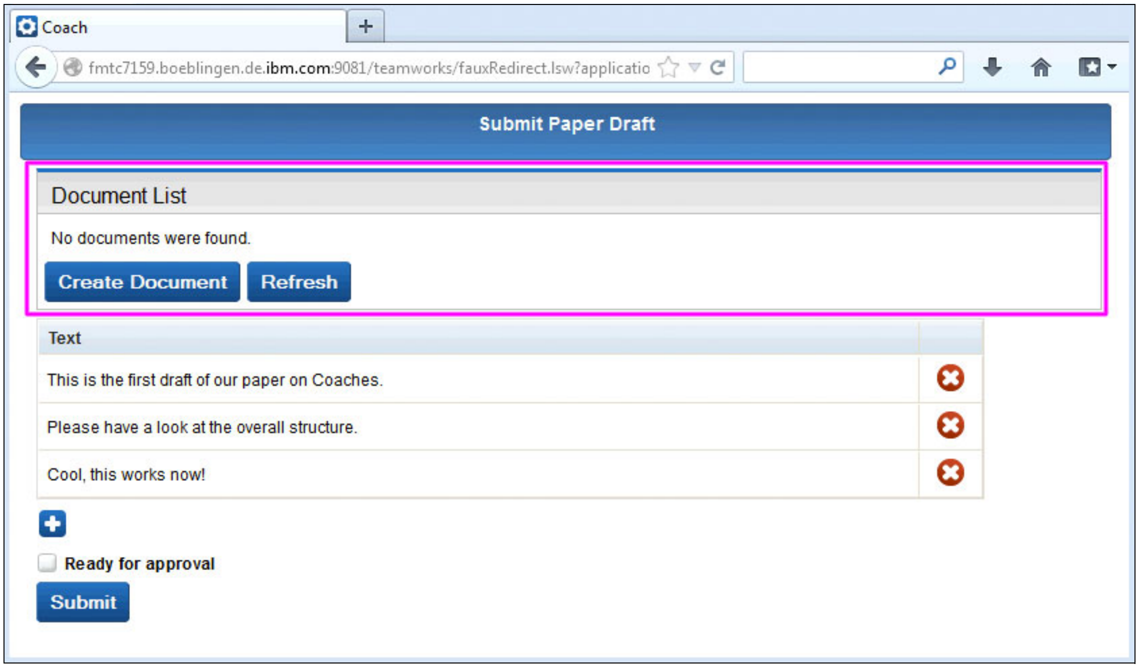
ECM document options:

Figure 2-33 Changing the configuration of Document List control

The primary change is to select “BPM Documents” as document store, instead of using an external Enterprise Content Management (ECM) system by selecting the corresponding value for the “Configure for use with” option.

Another important configuration change is to ensure that you only display documents that are associated with the corresponding process instance. Doing that allows multiple processes that are run in parallel, each operating on a distinct set of document drafts. To make that change we drill into Display options of BPM document options. There, we select the “Associated with process instance” check box to ensure that only documents are shown that are associated with the process instance.

To verify the effectiveness of the configuration changes, we start the process (not the Human Service) in the playback environment. To do that, we open the Publish Paper process in Process Designer and press the play button. A pop-up window appears that indicates that the process should wait for human input. We accept to switch over to the Inspector view and there select the Submit Paper Draft step of the just started Publish Paper process instance and press the play button to run it. Figure 2-34 on page 41 shows what comes up in the browser window.

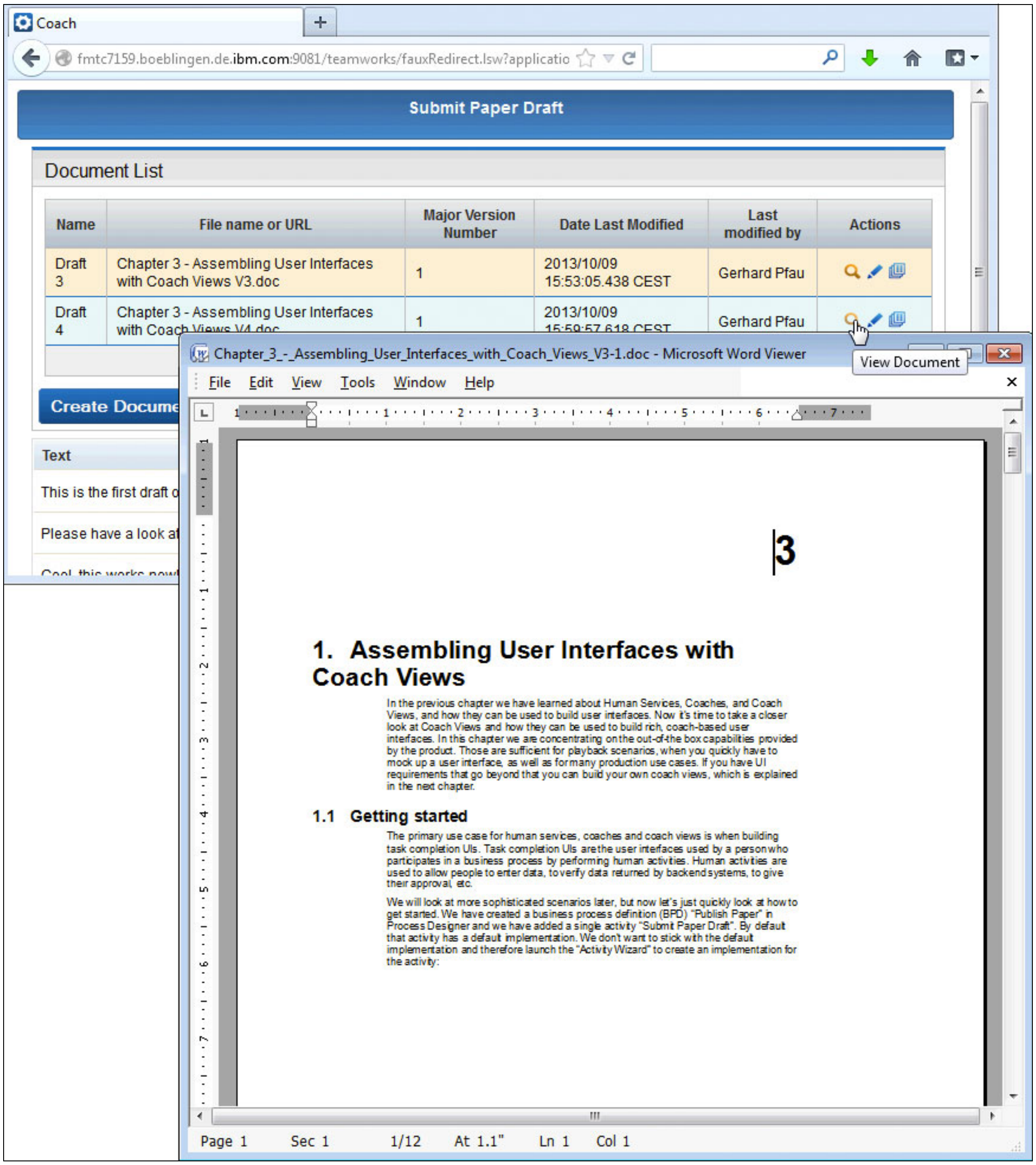


No documents were found.

Text
This is the first draft of our paper on Coaches.
Please have a look at the overall structure.
Cool, this works now!

Figure 2-34 Running Submit Paper Draft activity in Publish Paper process

The new Document List control is displayed but no documents have been added yet and therefore the list is empty. We continue our test by pressing the Create Document button to create a document as an attachment to the business process instance. In the dialog that comes up, choose to upload a Microsoft Word document and give it the name “Draft 3”. We repeat that and upload another document, Draft 4, and then look at the Document List again. Everything works smoothly now. Pressing the View Document icon next to a document launches it in an external viewer. Figure 2-35 on page 42 demonstrates that.


Coach

fmtc7159.boeblingen.de.ibm.com:9081/teamworks/fauxRedirect.lsw?applicatio

Submit Paper Draft

Document List

Name	File name or URL	Major Version Number	Date Last Modified	Last modified by	Actions
Draft 3	Chapter 3 - Assembling User Interfaces with Coach Views V3.doc	1	2013/10/09 15:53:05.438 CEST	Gerhard Pfau	
Draft 4	Chapter 3 - Assembling User Interfaces with Coach Views V4.doc	1	2013/10/09 15:59:57.618 CEST	Gerhard Pfau	

Create Docume

Text

This is the first draft o

Please have a look at

Cool, this works now

Chapter_3_-_Assembling_User_Interfaces_with_Coach_Views_V3-1.doc - Microsoft Word Viewer

View Document

3

1. Assembling User Interfaces with Coach Views

In the previous chapter we have learned about Human Services, Coaches, and Coach Views, and how they can be used to build user interfaces. Now it's time to take a closer look at Coach Views and how they can be used to build rich, coach-based user interfaces. In this chapter we are concentrating on the out-of-the box capabilities provided by the product. Those are sufficient for playback scenarios, when you quickly have to mock up a user interface, as well as for many production use cases. If you have UI requirements that go beyond that you can build your own coach views, which is explained in the next chapter.

1.1 Getting started

The primary use case for human services, coaches and coach views is when building task completion UIs. Task completion UIs are the user interfaces used by a person who participates in a business process by performing human activities. Human activities are used to allow people to enter data, to verify data returned by backend systems, to give their approval, etc.

We will look at more sophisticated scenarios later, but now let's just quickly look at how to get started. We have created a business process definition (BPD) "Publish Paper" in Process Designer and we have added a single activity "Submit Paper Draft". By default that activity has a default implementation. We don't want to stick with the default implementation and therefore launch the "Activity Wizard" to create an implementation for the activity:

Page 1 Sec 1 1/12 At 1.1" Ln 1 Col 1

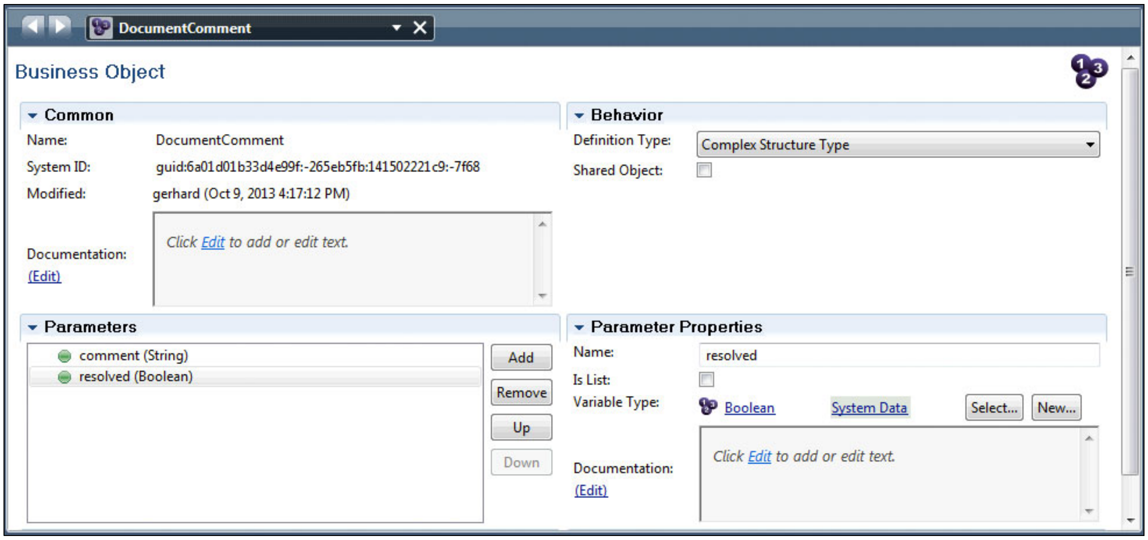
Figure 2-35 Working with Document List and viewing documents

Now that you have seen how to leverage some of the Coach Views provided by IBM Business Process Manager, it is important to point out that the mechanisms introduced here not only apply to IBM provided toolkits with Coach Views, but also to Coach View libraries that are provided by third parties, as introduced in later chapters of this book.

2.6 Creating Coach Views for business objects

When you are using complex data types in your applications, a convenient way to consistently visualize these data types is by generating a Coach View for them. The procedure is quite straightforward and can be accomplished with the knowledge that you already acquired.

Continuing our sample from above, we now want to allow users for each comment on a paper to indicate if that comment has been resolved or not. For that purpose, we define a new Business Object type DocumentComment using the Business Object editor in Process Designer, as shown in Figure 2-36.


DocumentComment

Business Object

Common

Name: DocumentComment
System ID: guid:6a01d01b33d4e99f:-265eb5fb:141502221c9:-7f68
Modified: gerhard (Oct 9, 2013 4:17:12 PM)
Documentation: (Edit) Click Edit to add or edit text.

Behavior

Definition Type: Complex Structure Type
Shared Object: ☐

Parameters

- comment (String)
- resolved (Boolean)

Parameter Properties

Name: resolved
Is List: ☐
Variable Type: Boolean System Data

Documentation: (Edit) Click Edit to add or edit text.

Figure 2-36 Defining Business Object DocumentComment

After we are done defining DocumentComment, we now create a Coach View for it by selecting the corresponding menu item from the menu of the Business Object as demonstrated in Figure 2-37 on page 44.

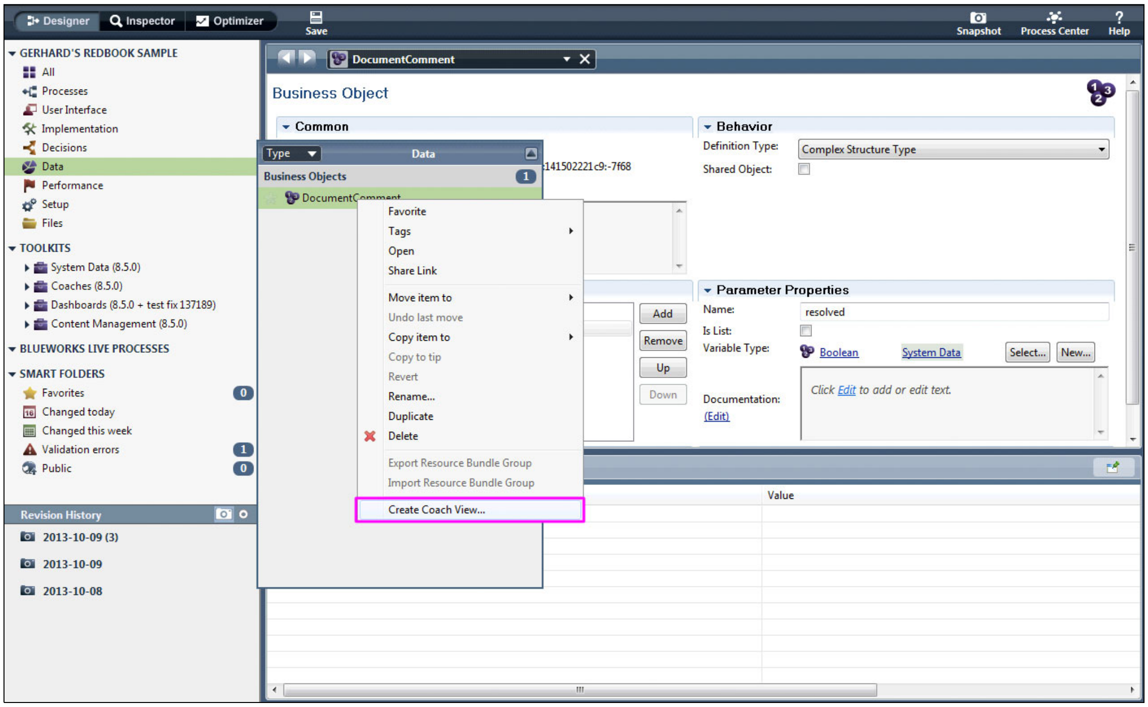


Figure 2-37 Creating a Coach View for DocumentComment

We give the new Coach View the name DocumentComment Coach View and we define it by just dragging and dropping the variable elements of the corresponding Business Object type to the canvas, and wrapping them with a vertical section. We also change the configuration of the Resolved check box to be rendered as a switch, as shown in Figure 2-38 on page 45. That is done by selecting the Switch option for the Show As configuration option for the coach view.

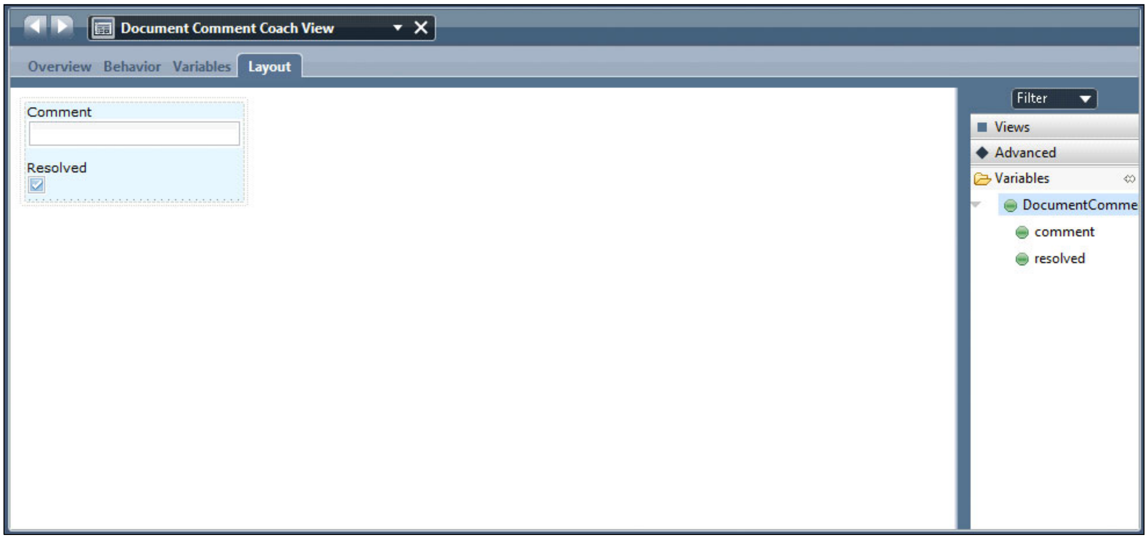
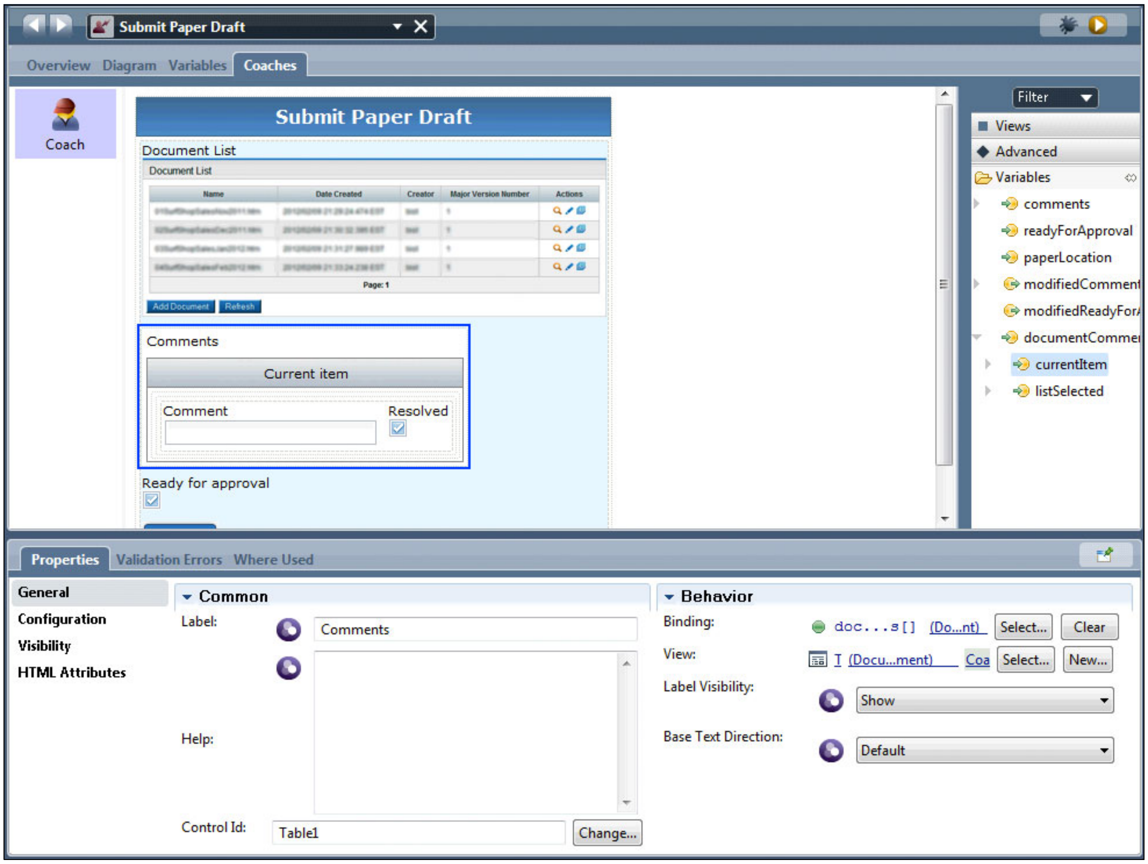


Figure 2-38   Layout of Document Comment Coach View

When that is done, we go back to our Coach and change the binding of the Comments table control to now bind to the `documentComments` variable. That variable holds a list of DocumentComment business objects. We remove the Text control in the table. Then, drag `currentItem` from the `documentComments` variable and drop it into the table control. The matching Coach View Document Control Coach View is automatically selected and added as shown in Figure 2-39 on page 46.



Label: Comments

Help:

Control Id: Table1

Binding: doc...s[] (Do...nt)

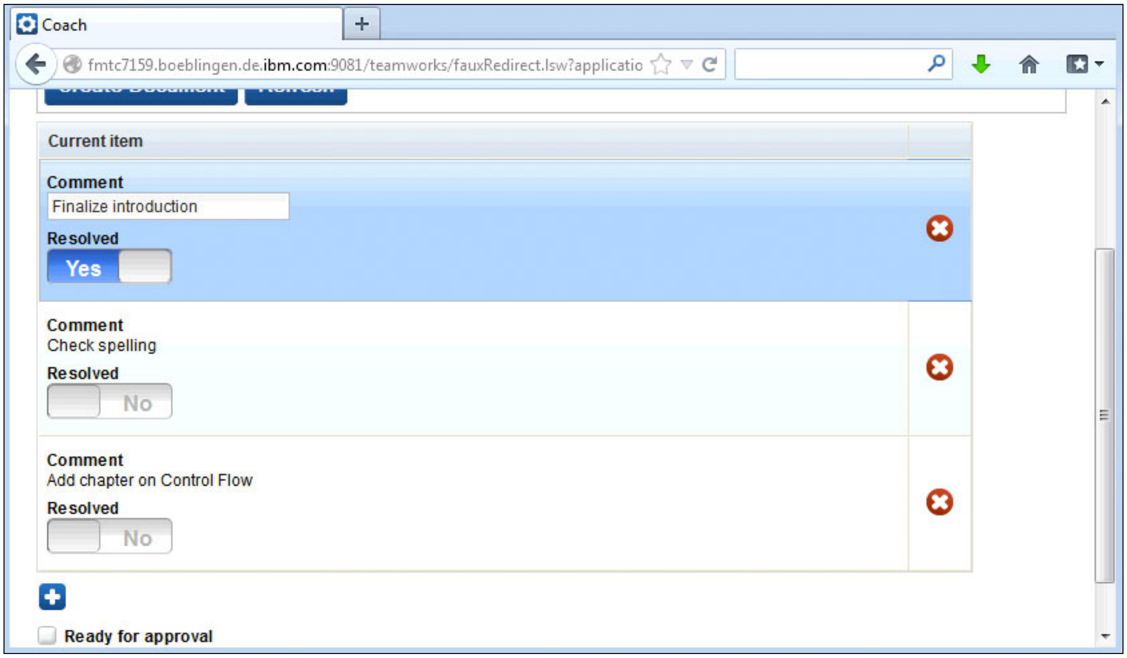
View: T (Docu...ment) Coa

Label Visibility: Show

Base Text Direction: Default

Figure 2-39 Using Document Comment Coach View to render rows of the Comments table

Now we test the Coach again. We see that the Comments table now not only has text comments anymore but also for each comment a switch that indicates if the comment has been already resolved as shown in Figure 2-40 on page 47.



Coach
fmtc7159.boeblingen.de.ibm.com:9081/teamworks/fauxRedirect.lsw?applicatio
Current item

Comment
Finalize introduction
Resolved
Yes

Comment
Check spelling
Resolved
No

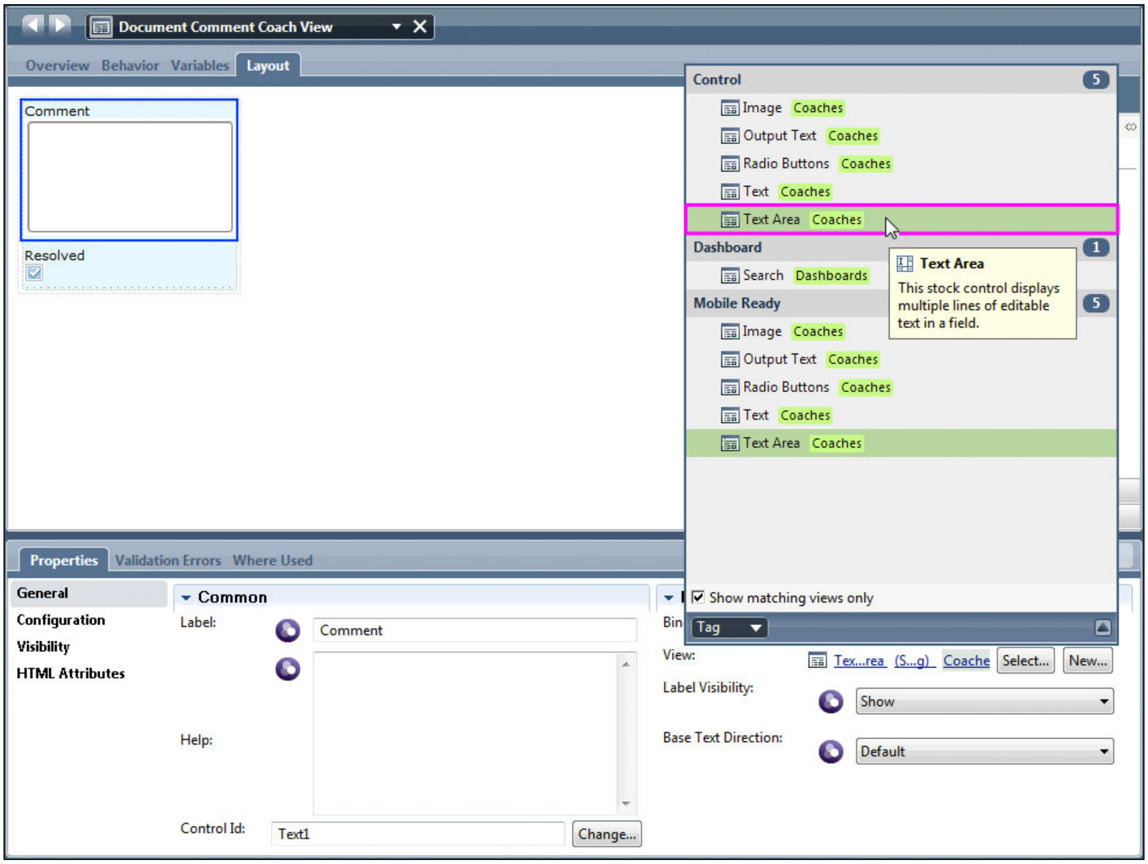
Comment
Add chapter on Control Flow
Resolved
No

Ready for approval

Figure 2-40 Rerunning Submit Paper Draft activity in Publish Paper process

Having seen all the power at our fingertips, we now are encouraged to further improve our UI to make it look *really* nice. To further improve our Coach look and feel, we apply a couple of simple changes to the Document Comment Coach View.

First, we change the Coach View that is used for the comment from Text to Text Area in the General section under Properties. That should give us a nice multi-line entry field for each comment as shown in Figure 2-41 on page 48.



Text Area

This stock control displays multiple lines of editable text in a field.

Label: Comment

Help:

Control Id: Text1

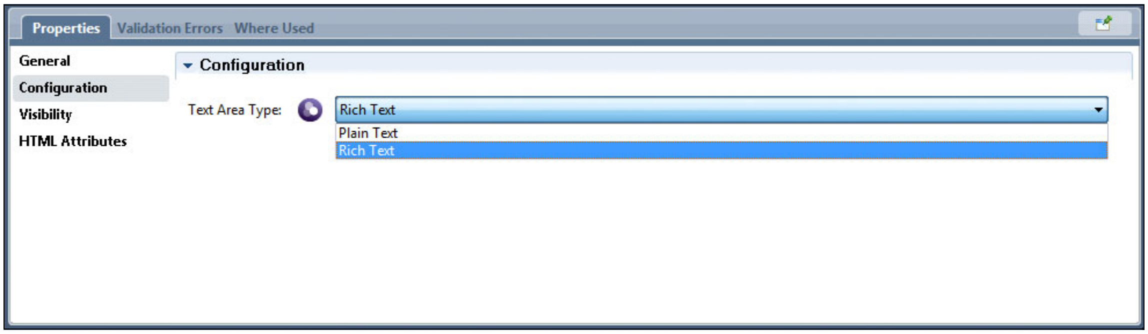
View: Tex...rea (S...g) Coache

Label Visibility: Show

Base Text Direction: Default

Figure 2-41 Changing View for Comment to Text Area

Because we also want to use rich text for our comments, we also change the Text Area Type configuration option of Text Area to Rich Text, as demonstrated in Figure 2-42 on page 49.

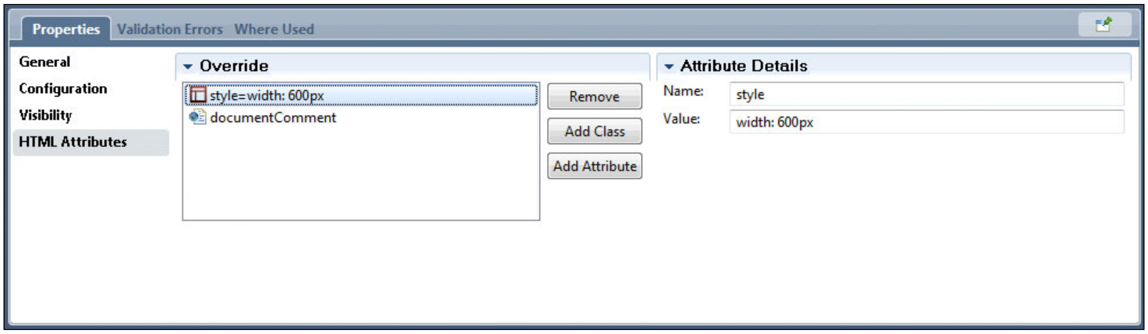


Text Area Type: Rich Text

Figure 2-42   Switching Text Area into Rich Text mode

The only thing that we are still missing is to make sure that we use all available horizontal screen space for Text Area. To do that, we use almost the same approach as described in 2.4, “Advanced configuration” on page 30.

We add under HTML Attributes an attribute *style* with a value of *width: 600px* and we also add a class named documentComment as shown in Figure 2-43.

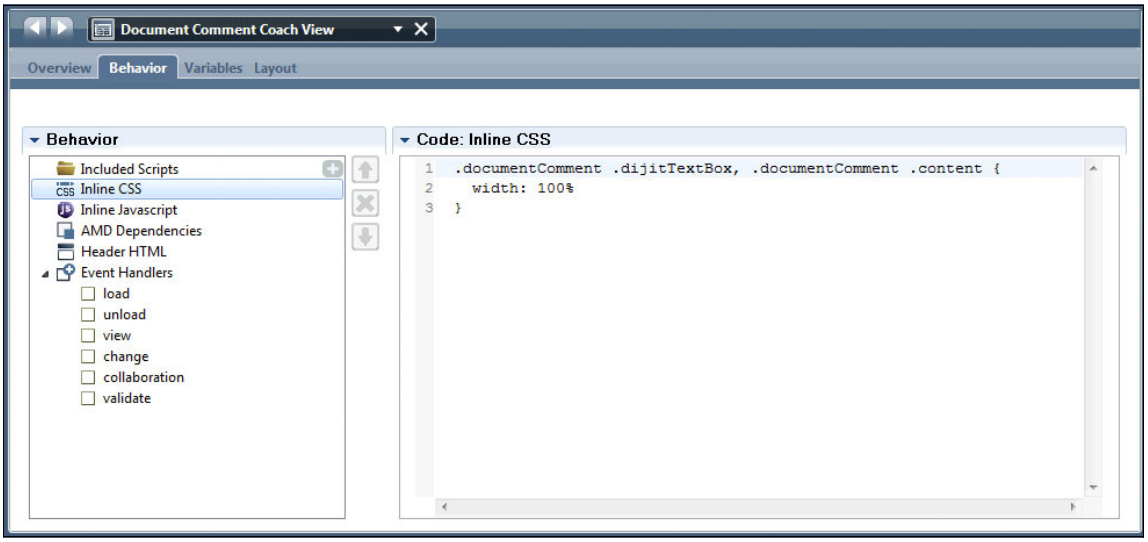


style=width: 600px
documentComment

Name: style
Value: width: 600px

Figure 2-43   Adding HTML Attributes to Text Area

Now all that remains to be done is to add the style extensions to the Coach View. When we did that for the Coach earlier, we had to add a Custom HTML block and write a <style> element. For Coach Views doing the same is even easier. All that you must do is to define your style extensions as Inline CSS on the Behavior tab of the Coach View. Figure 2-44 on page 50 shows how to do this.



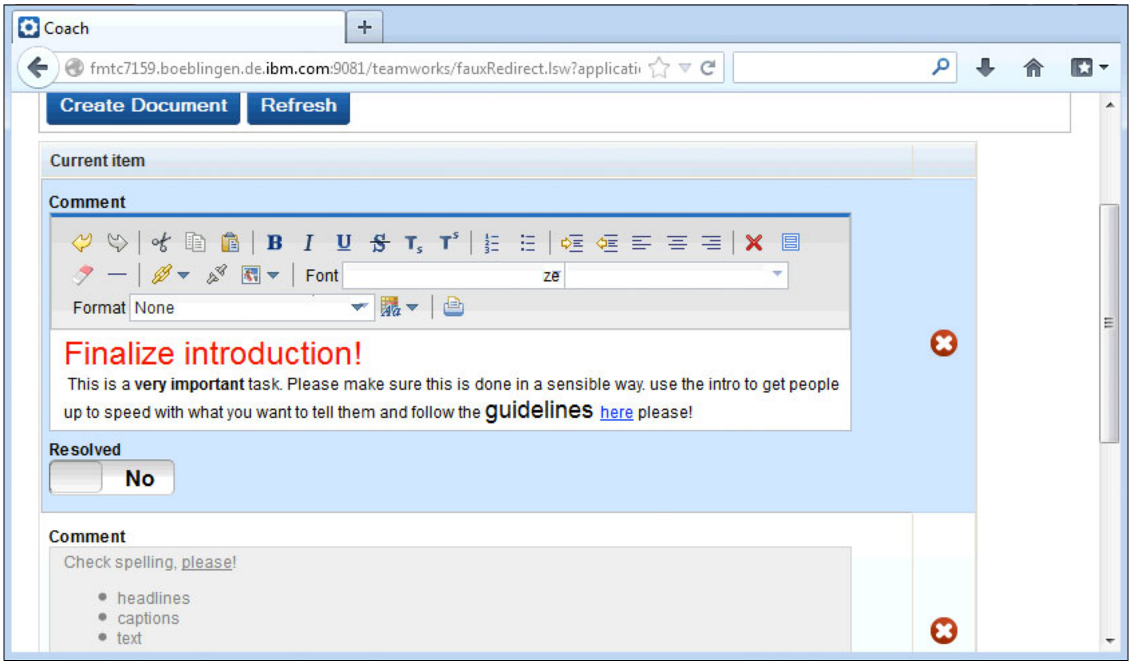
```
.documentComment .dijitTextBox, .documentComment .content {
  width: 100%
}
```

Figure 2-44 Adding Inline CSS to Document Comment Coach View

Styling: Styling coaches and Coach Views can be done in different ways. Earlier in 2.4, "Advanced configuration" on page 30, we showed coach-level styling by adding an HTML class to a Coach View on the canvas and by adding style tags in a custom HTML element at the top of the Coach. Here we have seen that the same is possible for Coach Views by also adding an HTML class to a Coach View on the canvas and by applying styles using the inline CSS capabilities of Coach Views. Other options are to include a CSS file in a Coach View or to set styles dynamically using JavaScript in a Coach View.

As you can see, there are many more things that you can do on the Behavior tab of the Coach View. Most of that gets relevant and important not for the casual Coach View builder who just clicks together a Coach View for a Business Object, but for modelers who design Coach Views from scratch to take over full control. This topic is discussed in more detail in Chapter 3, "Building Coach Views" on page 57.

Having done all of these changes to the Document Comment Coach View, you might now wonder if we have to touch the Coach that uses that Coach View again. Rerun the process and see if our changes got picked up automatically. See Figure 2-45 on page 51.



Finalize introduction!
This is a **very important** task. Please make sure this is done in a sensible way. use the intro to get people up to speed with what you want to tell them and follow the guidelines here please!

Check spelling, please!

- headlines
- captions
- text

Figure 2-45 Rerunning Submit Paper Draft activity with improved Document Comment Coach View

As you can see from Figure 2-45, it was sufficient to just revise the Coach View. The changes got picked up by the Coach using that Coach View and we got a UI that now not only has basic editing capabilities for comments but provides rich text editing capabilities and nice switches to indicate if comments have been resolved or require further work.

2.7 Further information

To not overload the sample, we stop adding additional details now. There are a few aspects though that might be interesting for you and thus we want to briefly mention them.

Coach Views can fire *Boundary Events*. Boundary Events are used to model the flow between coaches. In our simple example above we used only a single coach. In more sophisticated use cases, we would have multiple coaches in our human service, each representing a separate page of our UI. To navigate between these coach pages Boundary Events are used: A Boundary Event fired by one coach is used at modeling time to draw a connector to another coach. Alternatively, Boundary Events can be used to trigger some server-side logic, and then go back to the same Coach. That allows modeling Web 2.0 style user interface logic where only areas of the Coach are updated, not the entire page.

Several Coach Views fire Boundary Events. The one we have implicitly used in our example above is Button. When the button is pressed, a Boundary Event is fired, triggering navigation to the Coach that the button is connected to.

If you want to model a set of coaches where the end user can jump back and forth between the pages, you would add a Previous button that is wired to the previous coach and a Next button that is wired to the next coach, for example.

The following example in the IBM Business Process Manager version 8.5 information center has further details about wiring Coaches:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/index.jsp?topic=%2Fcom.ibm.wbpm.wle.editor.doc%2Fbuildcoach%2Ftopics%2Fexample_wiringcoaches.html

Boundary Events are also used to do server-side *validation* of a Coach. Validation is used to make sure that all required data has been entered in a complete and consistent way before leaving a Coach. The following sample introduces validation:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/buildcoach/topics/ezample_validatingacoach.html

Validation is usually modeled using validation scripts or validation services together with *Stay On Page* events. In bullet 10 of the introductory chapter, *Building Coaches*, in the information center, you can find further details:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/index.jsp?topic=%2Fcom.ibm.wbpm.wle.editor.doc%2Fbuildcoach%2Ftopics%2Ftbuildviewcoaches.html

2.8 Performance considerations

Now that we have seen all the capabilities that are available to us from a modeling perspective, let us take a quick look at some performance considerations. How we model a Coach can have an impact on the performance observed by the user. Therefore, it is worthwhile considering the following performance recommendations. For further information about this topic, including runtime deployment performance considerations, see Chapter 4, "Advanced performance considerations" on page 137.

2.8.1 Consider the expected browser version when designing Coaches

Rendering speed and efficiency vary significantly by browser vendor and release. When designing a Coach, consider the browser that will be used in production. This is particularly important for older browsers. Generally speaking, the more complicated the Coach user interface, the longer it will take to render on older browsers. Specific issues include:

- Large Document Object Model (DOM) trees
- Large JavaScript scripts
- Deeply nested Coach Views

2.8.2 Judiciously use the Table control

Table controls are very powerful, and are the appropriate control for many business scenarios. However, tables can be very expensive to render on some browsers, particularly older browsers. When using the Table control, ensure the total number of cells (number of rows x number of columns) is no more than are necessary. This minimizes the browser rendering time.

In addition, there are recent code optimizations in Business Process Manager to improve Table rendering time, sometimes dramatically. In particular, for Business Process Manager 8.0.1.1, APAR JR47845 delivers this improvement.

2.8.3 Judiciously use the Tab control

Tab controls are an excellent way to structure information within a Coach. However, in some Business Process Manager releases, a Coach with several tabs can take a significant amount of time to render. This is particularly true for older browsers. Consider this when designing the Coach interface, especially if the browser that will be used in production is an older browser.

If several tabs are required, implement lazy creation and loading of the tabs. In other words, do not create and load the tab until the user selects the tab. This will delay the rendering of the tab until the user is ready to interact with it.

In addition, there are recent code optimizations in Business Process Manager to improve Tab creation and rendering time, sometimes dramatically. In particular, for Business Process Manager 8.0.1.1, APAR JR47845 delivers this improvement.

2.8.4 Minimize the number of Boundary Events

For each Boundary Event, a network request is made to the server and the current state of the Coach (for example, all data values) is persisted. Both of these operations can be expensive, so use the fewest number of Boundary Events while still satisfying business requirements.

2.8.5 Minimize the size of Business Objects bound to Coach Views

Business Objects that are bound to Coach Views are persisted when Boundary Events occur. This can be a costly operation, especially for large Business Objects such as Industry Standard Schema (ACORD, HIPAA, and so on).

To reduce this cost, if the Business Object used in the process flow is large and complex, create a separate Business Object that only contains fields that are relevant to the Coach user interface, and bind that Business Object to the Coach View. After the Coach actions complete, merge the contents of the Coach-bound Business Object into the (larger) process flow Business Object. This is more efficient than persisting unnecessarily large Business Objects in Coaches when Boundary Events occur.

2.8.6 Pick appropriate delay time for auto-complete fields

For some coach controls, auto completion can be configured. Auto completion uses an auto completion service that is called after a certain delay to search for auto completion options. Caution should be used as these searches are performed against the server (for example: searching for a user name). The delay option specifies the time to wait after the user has finished typing in the field to when the request is sent to the server. If a short delay is chosen, several requests could be sent before the user finishes typing, causing unnecessary load against the server, and also delaying end-user response time in the Coach. Set a delay value that ensures the user has finished typing. Several hundred milliseconds is a good compromise between responsiveness and reducing server load.

2.9 Conclusion

With what you learned in this chapter, you should be able to build nice Coaches based on stock controls. You have learned how to assemble Coaches based on the Coach Views provided as part of the product and you have seen how to customize the stock Coach Views to fit your needs. You now know how Coaches can be wired together, how validation can be added, and what you need to keep in mind to allow for good response times when users are interacting with the Coaches that you built. You have also seen how easy it is to generate simple Coach Views for Business Objects and how to use them. In the next chapter, you learn about how to go beyond that by building custom Coach Views from scratch to get full control over the components that constitute the user interface.

3

Building Coach Views

This chapter provides an introduction to creating new Coach Views by using the Process Designer and its associated APIs. The Coach API,[1] also known as the Coach View framework or Coach View API, is introduced by samples in the first part of this chapter. Lessons that are learned from this practice are given in the second part of this chapter.

Notes: Before you start developing custom Coach Views, enable JavaScript debugging in your IBM BPM environment, which allows the more readable versions of the Coach Framework JavaScript and associated libraries to be loaded. For more information about this topic, go to the following website:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/t_enablingdebuggingforcoaches.html

This chapter includes example JavaScript code that uses the Dojo Toolkit. The Coach Framework does not require Coach View developers to use Dojo. For example, the product information center includes a sample that uses jQuery to create a custom coach view. You can find that sample at the following website:

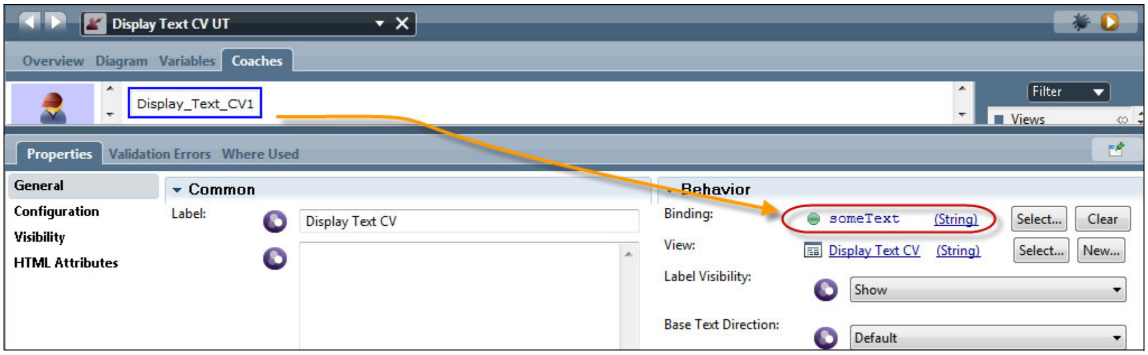
http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/tjquerybutton.html

[1] You can find more information about the Coach API in the IBM BPM 8.5 Information Center, found at:
http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/rcoachapi.html

3.1 Business data

Business data is essentially the data of a company, and is information with business value. You use configuration options to customize Coach Views, to communicate with other Coach View instances, and to interact with back-end systems through services. Configuration options are described in 3.2, “Read/Write configuration options” on page 73.

Coach Views can support one business data variable and one or more configuration options, which is different from services in IBM Business Process Manager (BPM), where there are input and output variables. The Coach View framework uses a data binding mechanism to communicate changes to Coach Views through Coach View event lifecycle handlers. You can bind variables to Coach View through a configuration option or binding, as shown in Figure 3-1.



Label: Display Text CV

Binding: someText (String)

View: Display Text CV (String)

Label Visibility: Show

Base Text Direction: Default

Figure 3-1 Binding business data

Note: More information about data binding can be found in the information center that is found at the following website:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/cdatabind.html

3.1.1 Accessing business data

To access business data in the Coach View event handlers, you must ensure that some preliminary setup is done:

- The Coach View must be added to a Coach.
- The Human Service that contains the Coach should have a variable instance of the type of the Coach View business data.

- The variable instance from the Human Service should be bound to the Coach View on the coach.

> **Note:** It is a preferred practice to develop a Coach View in a unit that is test a Human Service so that when you are getting started with a new Coach View, the impact on other developers that work on the same Coach is minimized. Also, side effects by other Coach Views can be minimized by focusing only on one Coach View in the unit test. When the custom Coach View becomes stable, you can then integrate it into the Coach. More details about this approach are 3.1.2, "Testing Coach Views with Human Services" on page 68.

The Coach View framework offers a set of behaviors and event handlers that can be used to build Coach Views. The following sections introduce the different event handlers and behaviors with samples. You can find basic introductions for each event handler in the information center that is found at the following website:

`http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/reventhandler.html`

A custom Coach View sample

Look at a sample of a simple Coach View implementation. This implementation shows how to create a basic Coach View that displays text. Then, the sample shows how the function of the Coach View is extended to give you an idea of how iterative Coach View development works and to introduce API functions and event handlers of the Coach View framework.

To get started with the sample, create a Coach View named Display Text CV.

In Process Designer, click the plus icon on the User Interfaces line, and then select **Coach View** to create a Coach View, as shown in Figure 3-2.

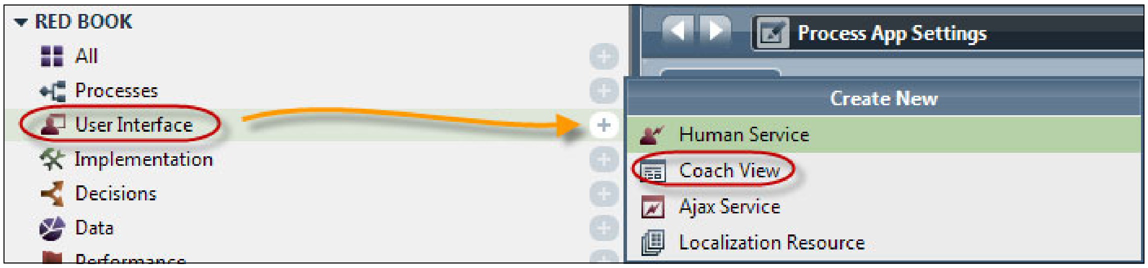
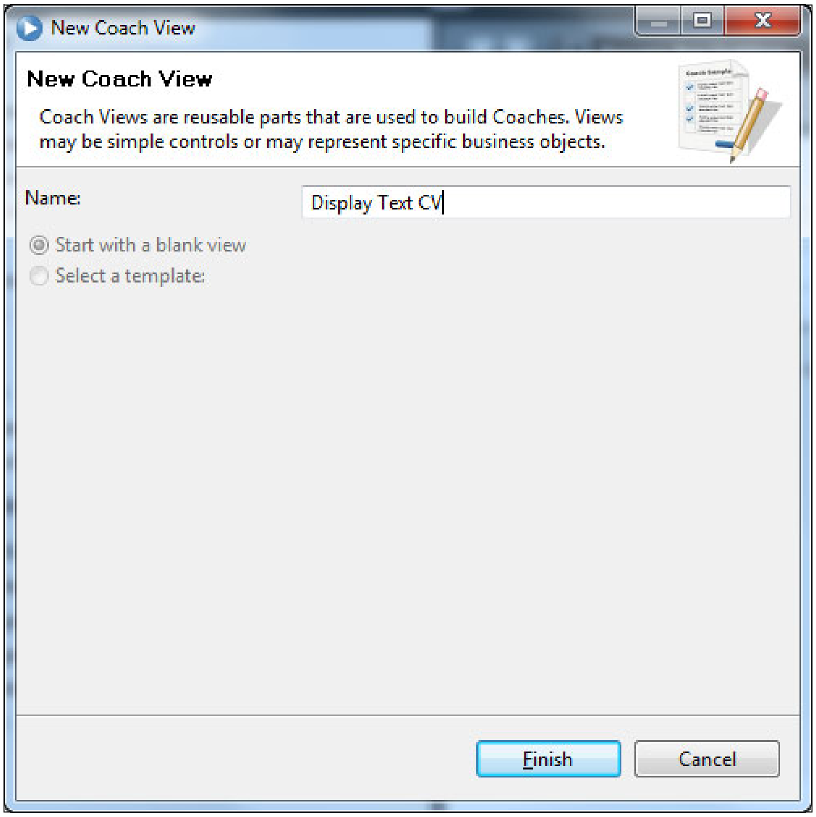


Figure 3-2 Create new Coach View

A window opens and prompts you for the name of the new Coach View, as shown in Figure 3-3.



New Coach View

Coach Views are reusable parts that are used to build Coaches. Views may be simple controls or may represent specific business objects.

Name: Display Text CV

Figure 3-3 Create Coach View through a wizard

Variables

In the Variable section of the editor, define a Business Data variable in the new Coach View. For this sample, use the simple data type String, as shown in Figure 3-4 on page 61.

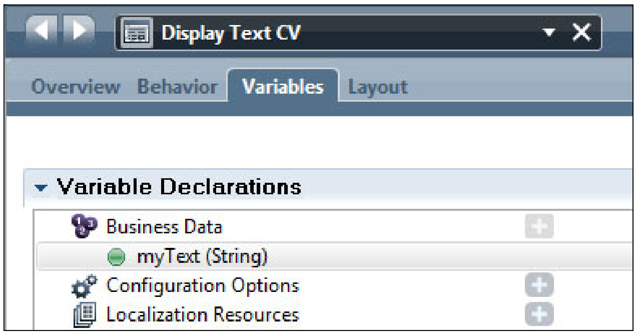


Figure 3-4 Define a Business Data variable

Note: You can also derive a Coach View from an object definition. When complex objects are used as business data, the preferred way to create a Coach View is through the Create Coach View window that is shown in Figure 3-5.

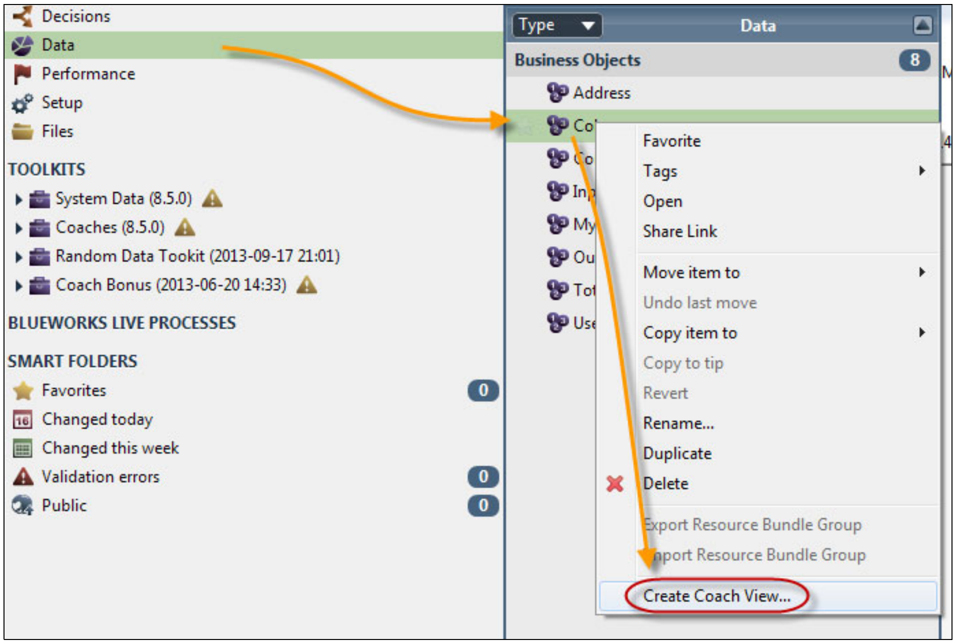
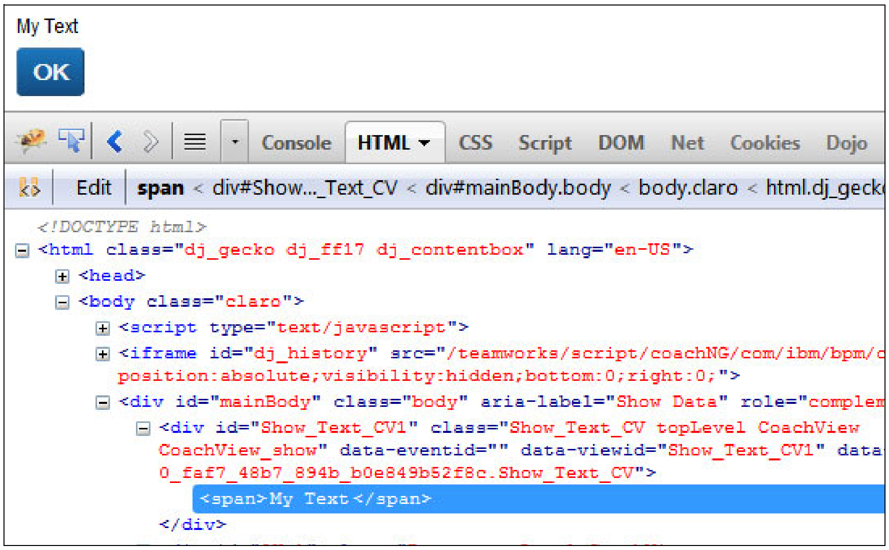


Figure 3-5 Create Coach View wizard

Layout

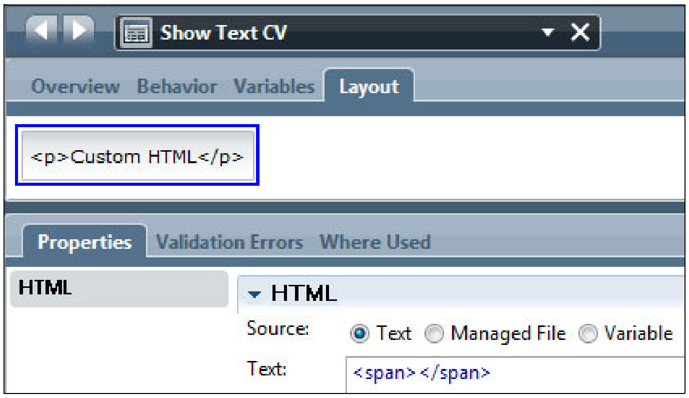
If you want to display a text in a simple Coach View, it is convenient to have an *anchor point*. This anchor point can be a static HTML element. For this sample, use a span HTML element. Figure 3-6 shows the span highlighted in Firebug.


My Text
OK
Console
HTML
CSS
Script
DOM
Net
Cookies
Dojo
Edit
span < div#Show..._Text_CV < div#mainBody.body < body.claro < html.dj_geck

```
<!DOCTYPE html>
<html class="dj_gecko dj_ff17 dj_contentbox" lang="en-US">
  <head>
  <body class="claro">
    <script type="text/javascript">
    <iframe id="dj_history" src="/teamworks/script/coachNG/com/ibm/bpm/c
    position:absolute;visibility:hidden;bottom:0;right:0;">
    <div id="mainBody" class="body" aria-label="Show Data" role="complem
      <div id="Show_Text_CV1" class="Show_Text_CV topLevel CoachView
      CoachView_show" data-eventid="" data-viewid="Show_Text_CV1" data
      0_faf7_48b7_894b_b0e849b52f8c.Show_Text_CV">
        <span>My Text</span>
      </div>
```

Figure 3-6 Span HTML element

To specify the span in the sample Coach View, switch to the Layout section to add a Custom HTML control. Here you can add the span tag, as shown in Figure 3-7.



Source: [✓ Text] Managed File Variable

Text: `<span></span>`

Figure 3-7 Coach View Layout

Now you have an anchor point that you can programmatically use through the Coach View API or standard HTML selectors.

Inline JavaScript

In this sample, switch to the Behavior section and then to Inline JavaScript. Inline JavaScript can hold common functions and local variables for your custom Coach Views. These functions and private variables can be used in all events of a Coach View instance. Inline JavaScript is not an event handler, but it is part of the Coach View definition itself.

For this sample, in the Coach View, specify local variables to hold the content of bound variable instances and Document Object Model (DOM) elements in the Coach View instance context by completing the following steps:

1. Switch to the Behavior section.
2. Select **Inline JavaScript**.
3. Add a variable by using the following lines:

```
var _this
   , span
   , text;
```

You see that we defined the JavaScript variables `_this` and `text`, as shown in Figure 3-8 on page 64. The variable `_this` holds a general reference to the context object.

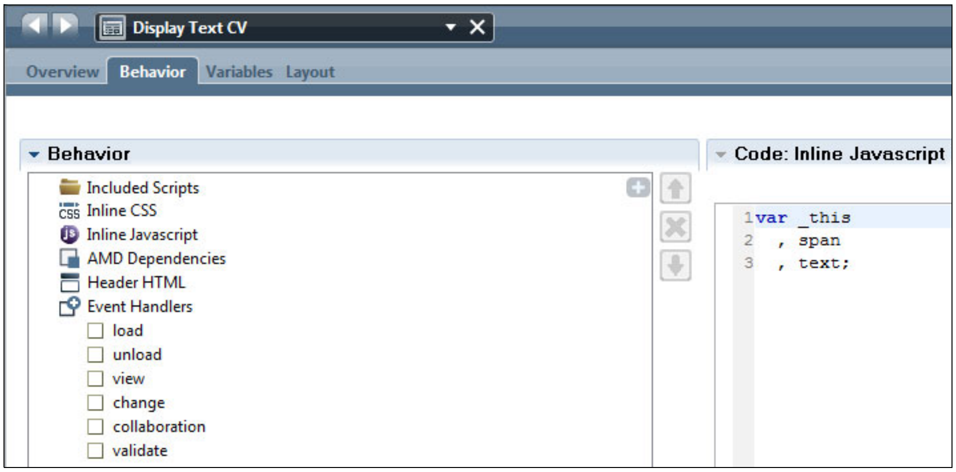
Note: When using a local _this cache that is defined in Inline JavaScript, a good practice is to null out _this in the unload event handler to free up resources.

Sample:

_this = null;

The context object provides access to helper functions, such as a callback to fire a named Boundary Event. More information about this topic can be found at the following website:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/rcontext.html



```
var _this
  , span
  , text;
```

Figure 3-8 Creating local variables

The variable `text` is used to hold the business data binding of the Coach View instance. We show the value of `text` within the `span` element. The local `span` variable is used to hold a reference to the span HTML DOM element that was defined in “Layout” on page 62.

Loading

The load event handler is used to run initialization tasks, as the one shown below, that are executed before the view event handler is run:

```
myBusinessData = this.context.binding ?
this.context.binding.get(“value”) : null;
```

Note: When you cache objects, you must be aware of potential stale objects. Stale objects are basically instances that are out of sync. Instances are out of sync if your local object instance contains outdated information that does not reflect the most recent state of the object instance. To ensure that you have the latest information, update your local copy regularly with the current state.

Consider avoiding caching altogether. For more information about this topic, see “Caching binding objects” on page 134.

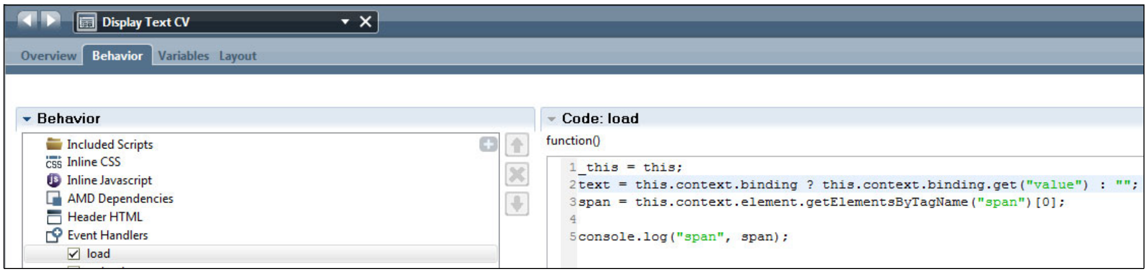
When you bind to variables in the load event handler, it is a preferred practice to use conditional operators. These operators allow you to define defaults for variables that can make your Coach View more robust. Defaults allow you to handle unbound configuration options or bindings in a more controlled way.

Conditional operators: Let us show you a typical conditional operator:

```
yourvariable = (condition) ? value1 : value2;
```

Value1 is used if the condition evaluates to true. value2 is the default value.

In this sample, gather information from bindings and reach out to the HTML anchor point (which was defined in “Layout” on page 62), as shown in Figure 3-9.



```
_this = this;
text = this.context.binding ? this.context.binding.get("value") : "";
span = this.context.element.getElementsByTagName("span")[0];

console.log("span", span);
```

Figure 3-9 Simple load event handler

The first line places this information into the _this variable (which was defined in “Inline JavaScript” on page 63), as shown in Example 3-1. Variables that are defined in the Inline JavaScript section are applied globally for a Coach View instance, which means that you can access the variables from any event handler in a Coach View instance. Thus, when you assign the value during the loading process, the assigned value also is accessible in the view event handler, as shown in Example 3-1.

Example 3-1 Assigning values during load

```
_this = this;
text = this.context.binding ? this.context.binding.get("value") : "";
span = this.context.element.getElementsByTagName("span")[0];
console.log("span", span);
```

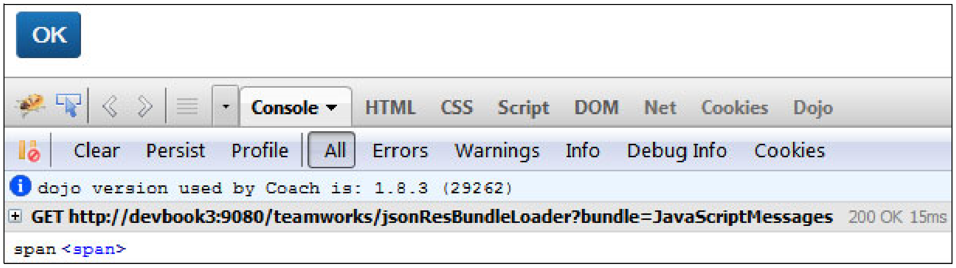
In the load event handle, you get the initial value of the bound variable and store it in the text variable. Use a conditional operator to ensure that there is always a value that is assigned to text.

Note: A key part of this process is the usage of the API function **`this.context.element`**, which scopes the search for DOM elements to this Coach View instance. When this API is used, the whole Coach is *not* searched for the specified tag.

Now, reach out to the anchor point, the span HTML element, which was defined in “Layout” on page 62. For this sample, use the getElementsByTagName API call that searches for HTML elements in the scope of this Coach View instance. Use the tag that you are looking for as the parameter. In this sample, search for span tags. There is only one span tag that is defined in this Coach View, so our anchor to the span is at position 0 of the list of nodes that is returned by the call.

Note: There are multiple ways that you can find DOM elements. You can also use, for example, querySelectorAll or querySelector, which are HTML 5 functions.

In this sample, we added a `console.log` statement. When you develop Coach Views, it is handy to use a console for debugging purposes. In Firebug, a plug-in for Firefox, the output that is shown in Figure 3-10 is shown for our sample.



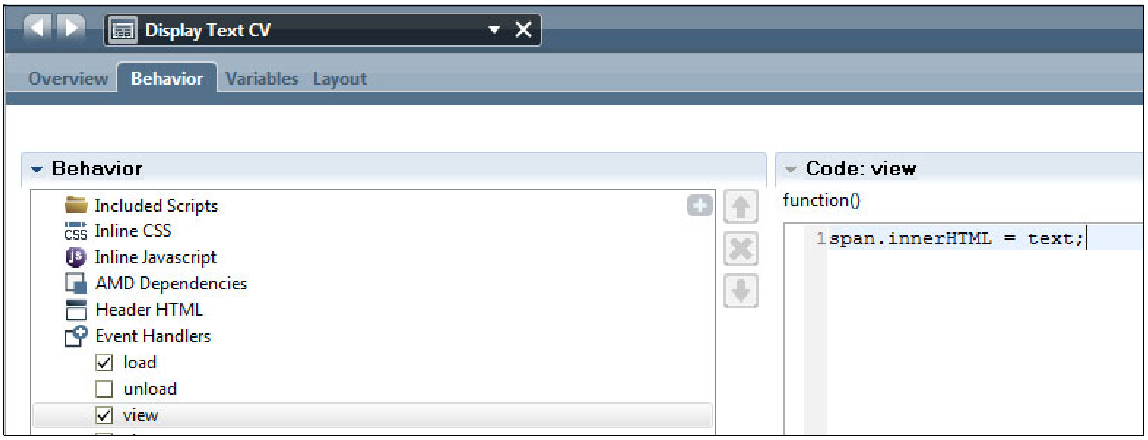
```
dojo version used by Coach is: 1.8.3 (29262)
GET http://devbook3:9080/teamworks/jsonResBundleLoader?bundle=JavaScriptMessages    200 OK 15ms
span <span>
```

Figure 3-10 Firebug console output

Now, you can see that the span tag was successfully found by the API call and you can now assign text that is bound to the Coach View.

View

The `span.innerHTML` assignment sets the text when the view event handler is called, as shown in Figure 3-11 on page 67.

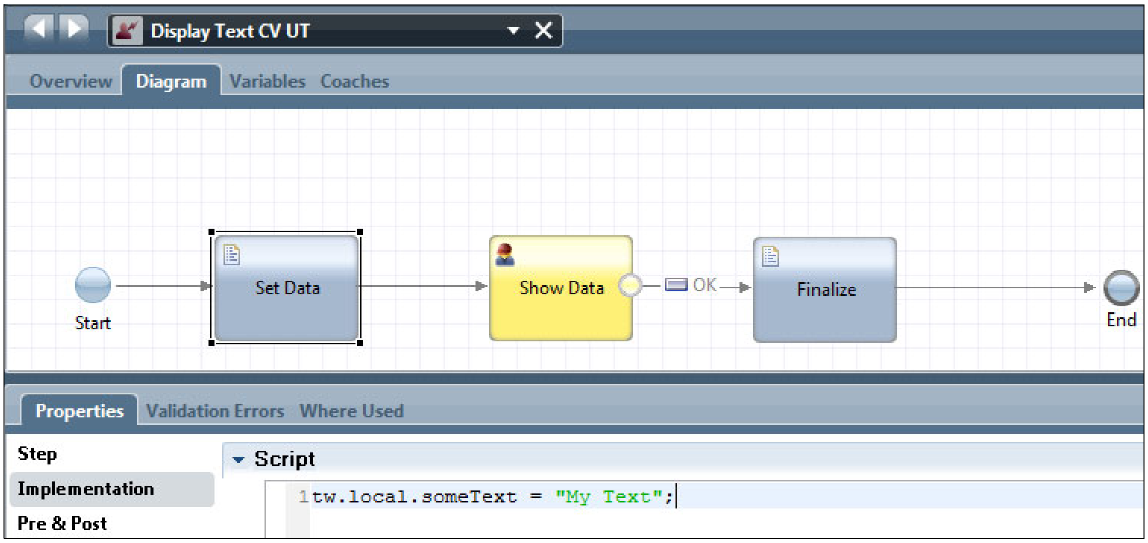


```
function()
span.innerHTML = text;
```

Figure 3-11   Simple view event handler

Human Service

To test our custom Coach View, create a unit test Human Service, as shown in Figure 3-12, to validate that everything works as expected.



```
tw.local.someText = "My Text";
```

Figure 3-12   Simple Human Service unit test

3.1.2 Testing Coach Views with Human Services

As mentioned in 3.1.1, “Accessing business data” on page 58, it is a preferred practice to create Unit Test services when you develop custom functions for the following reasons:

- ► This service can be used to develop a Coach View in isolation, so you are not working in the coach directly where it should be used. This isolation allows you to reduce the side effects by other controls on a coach.
- ► When you complete development, you can use the Human Service as a sort of unit test, allowing you to debug and to reproduce errors. This can save time, especially for complex Coach Views with many parameters because you do not have to create a unit test first before you can start debugging.
- ► When another developer takes over, he has a better starting point because there is already a sample that uses the Coach View. In this case, the Human Service serves as documentation.

> **Note:** When you develop Coach Views, it is convenient to have data to test your custom Coach View functions. You can use the Random Data toolkit for data generation. The toolkit can be found in the IBM BPM wiki at the following website:
>
> `http://bpmwiki.blueworkslive.com/display/samples/Data+Generator`

A good pattern is for your test Human Service to have some of the following items:

1. A service to perform basic initialization. If you use complex objects, you can create a service for each complex object. If you must initialize multiple complex objects, you can nest the initialization services. Using services enables you to reuse initialization logic, which improves maintainability.
2. A server script to set defaults.
3. A first coach where you can manually adjust parameters of the Coach View.
4. The main coach.
5. A server script to do final actions, such as nulling out complex object instances to reduce general memory usage, as shown in Figure 3-13 on page 69.

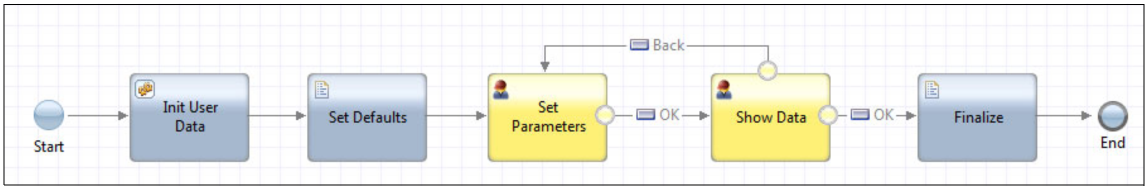
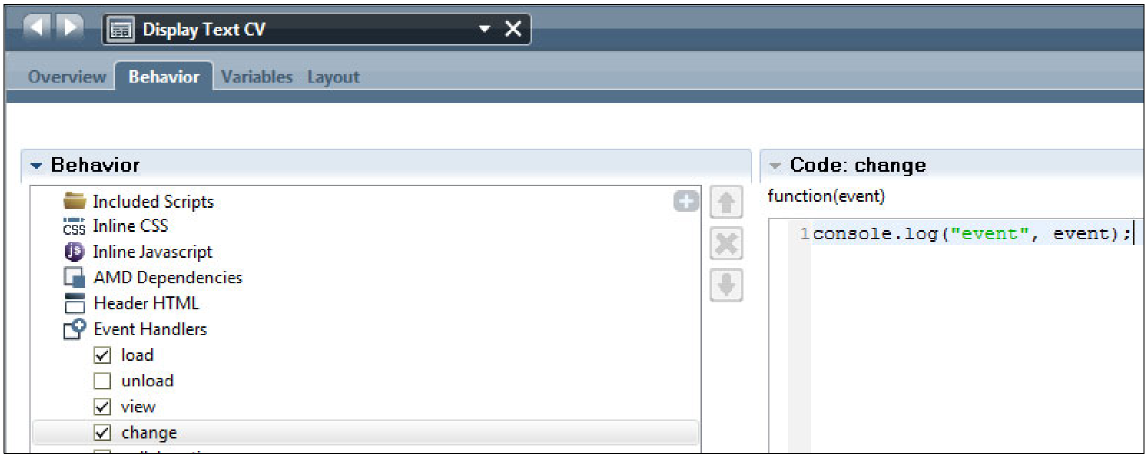


Figure 3-13 Unit test Human Service inspiration pattern

3.1.3 Change event handler introduction

If you want to make your Coach View react to changes in the business data variable, you must implement the change event handler. Each time the business data variable instance that is bound to the coach is changed, the Coach View framework triggers the change event handler.

Let us look at a change event handler; you can put a console log in to the change event to inspect the event object in Firebug, as shown in Figure 3-14.



```
function(event)
console.log("event", event);
```

Figure 3-14 Simple change event handler

In Process Designer, you can place your custom Coach View on a Coach and bind it to a variable, as shown in Figure 3-15 on page 70.

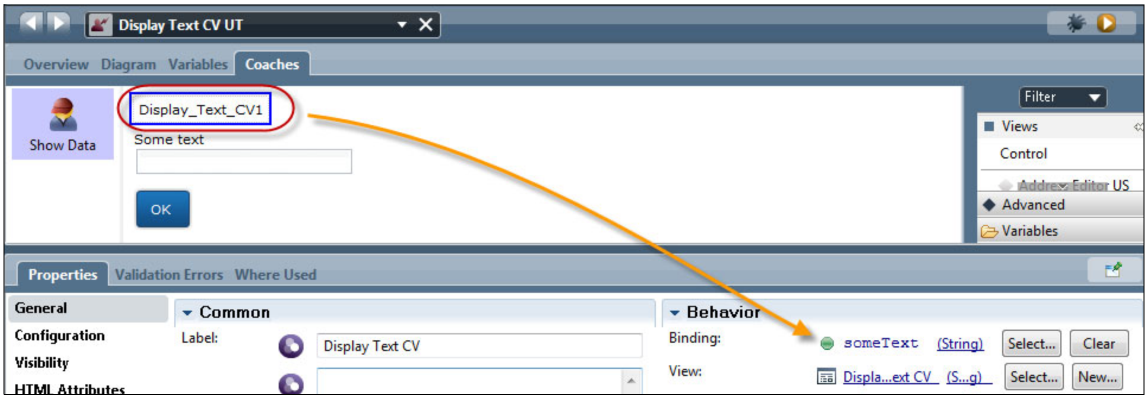
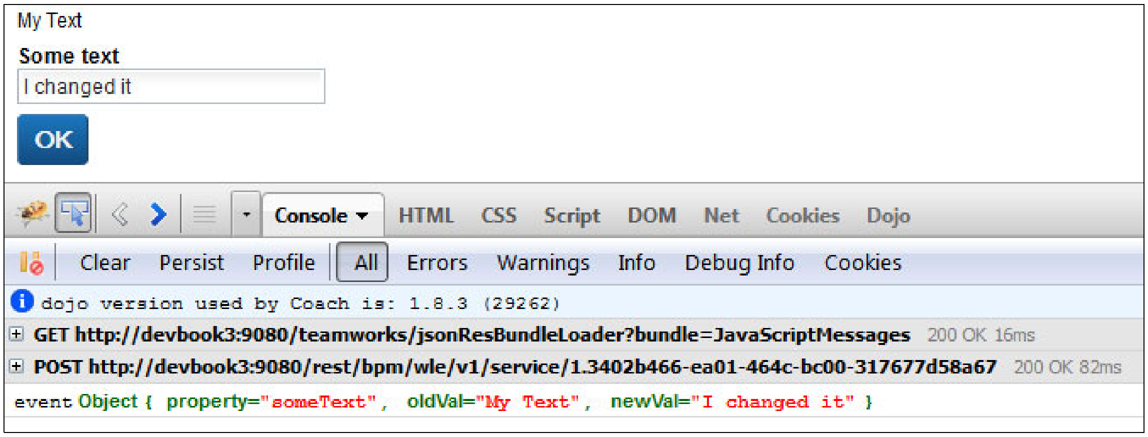


Figure 3-15 Custom Coach View in Process Designer

Then, you can see what happens in the Firebug console when you change the value, as shown in Figure 3-16.



```
dojo version used by Coach is: 1.8.3 (29262)
GET http://devbook3:9080/teamworks/jsonResBundleLoader?bundle=JavaScriptMessages   200 OK 16ms
POST http://devbook3:9080/rest/bpm/wle/v1/service/1.3402b466-ea01-464c-bc00-317677d58a67   200 OK 82ms
event Object { property="someText", oldVal="My Text", newVal="I changed it" }
```

Figure 3-16 Change event in Firebug

You can see that the old value and the new value are contained in the event. Also, you can see the name of the variable that is bound to the Coach View, as shown in Figure 3-17 on page 71.

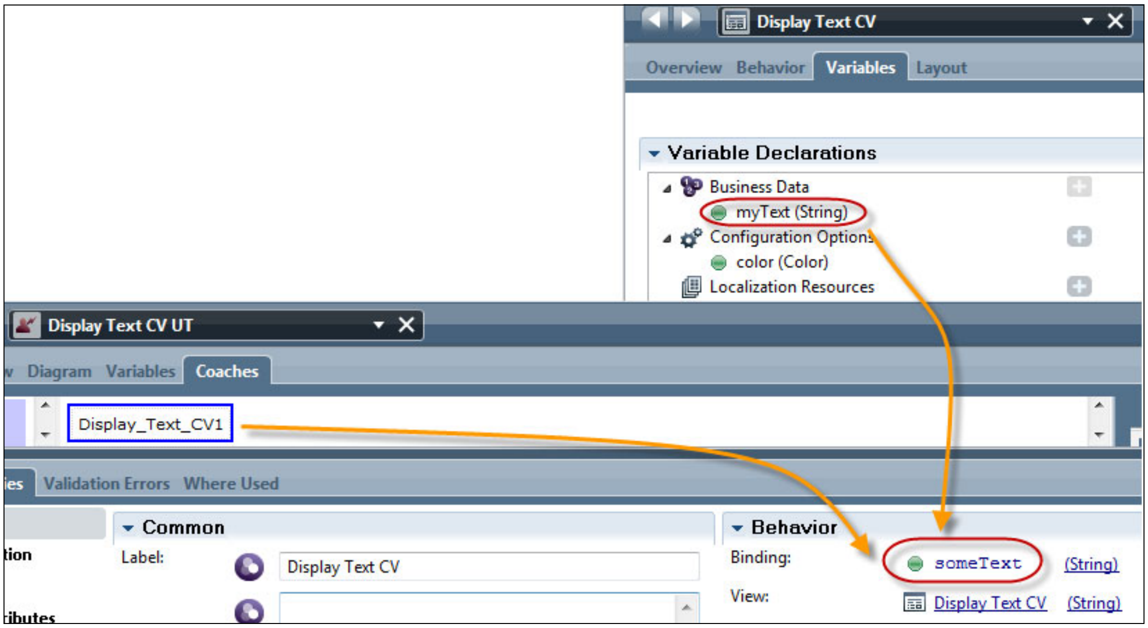


Figure 3-17 Relationship between private variable name and event property

The name of our business data variable in our sample was myText, but the property attribute has the value someText, as shown in Figure 3-17. This is the name of the variable that was bound to the Coach View instance in the Coach.

Filter change event

To change the text that is displayed by the Coach View, you need to capture the new value of the bound variable when it changes. The change event handler is triggered each time the value of the bound variable changes. The handler receives an event object when a change is made.

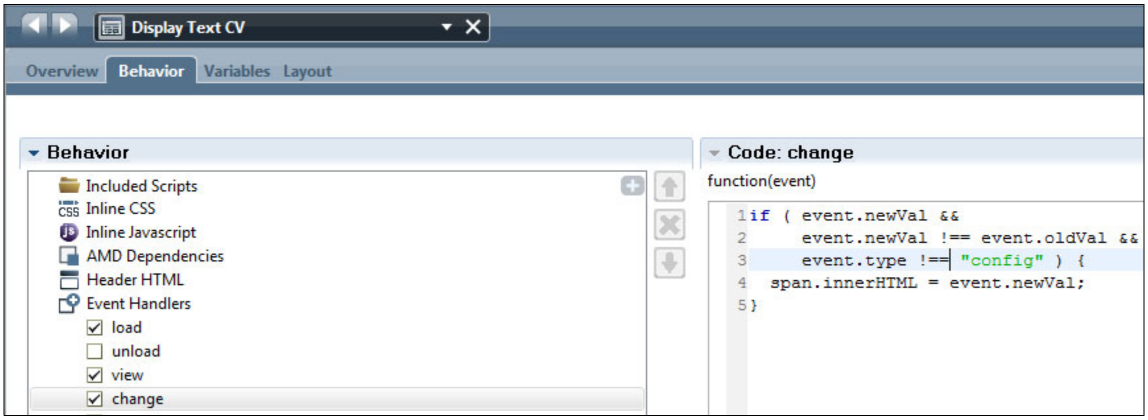
Note: You can find more information about the change event handler in the information center that is found at the following website:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/rchangeevent.html

In this sample, we want to change only the value that is shown by the Coach View if the event object contains a new value and if the new value is not the same as the old value. You can write a filter condition to reduce the number of updates to the span element to a minimum.

Note: Filtering events is important, especially when expensive operations are called in your change logic. For example, if a back-end integration that takes some seconds to run is called as a result of a change, you want to ensure that it is not called unnecessarily.

In this sample, we added a filter that checks that the event type is not config because configuration options also use events, but we do not want to update the span if a configuration option of our Coach View is updated. In the change event handler, the event type is either a "config" value or undefined / null. If the event type is not a "config" value, there is an indicator that business data was changed, as shown in Figure 3-18.

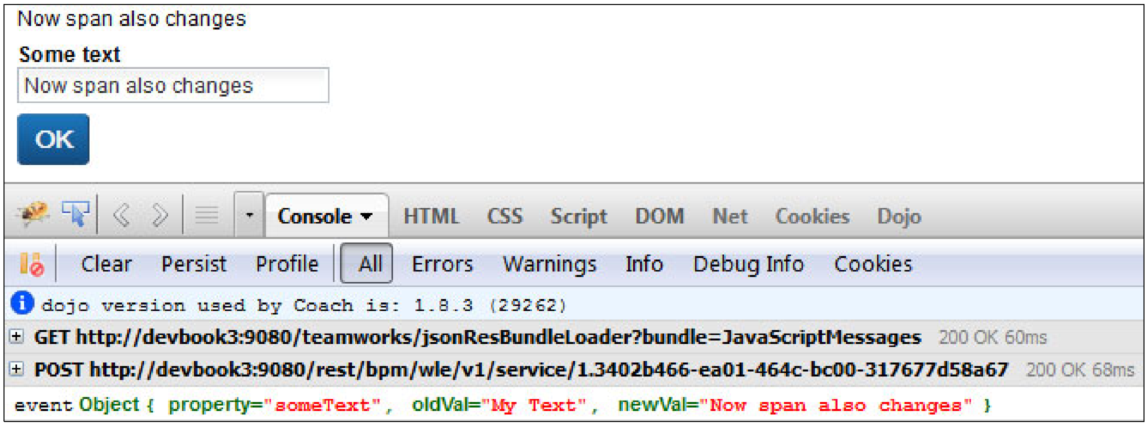


function(event)

```
if ( event.newVal &&
     event.newVal !== event.oldVal &&
     event.type !== "config" ) {
  span.innerHTML = event.newVal;
}
```

Figure 3-18   Change event handler with filter

In this condition, update the innerHTML of the span element with the new value. With Firebug, you can now validate if this works as expected, as shown in Figure 3-19 on page 73.



```
dojo version used by Coach is: 1.8.3 (29262)
GET http://devbook3:9080/teamworks/jsonResBundleLoader?bundle=JavaScriptMessages   200 OK 60ms
POST http://devbook3:9080/rest/bpm/wle/v1/service/1.3402b466-ea01-464c-bc00-317677d58a67   200 OK 68ms
event Object { property="someText", oldVal="My Text", newVal="Now span also changes" }
```

Figure 3-19 Change event in Firebug

3.2 Read/Write configuration options

Configuration options can be used so that a Coach View can interact with other Coach Views. Typical configuration options contain data only used within your Process App, for example, to help you to select the correct path and to keep state information.

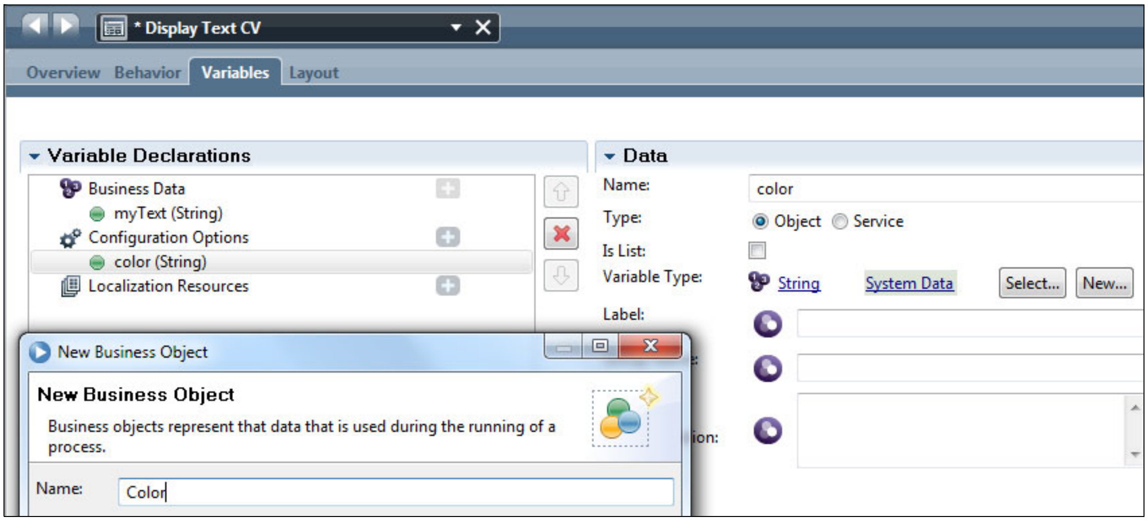
The value of a configuration option that is bound to a Coach View can be accessed through the API as follows:

```
var myOption = !!this.context.options.myOption ?
this.context.myOption.get(“value”) : null;
```

Sample

Let us look at a simple sample that gives insight in to how you can use configurations options. In this sample, extend the previous Display Text CV with a configuration option that allows you to change the text color.

First, create a configuration option in the Variables section. You can create a variable instance that is called color and a type that is named Color for it, as shown in Figure 3-20 on page 74.



Name: color
Type: [✓ Object] Service
Is List:
Variable Type: String System Data
Label:

New Business Object

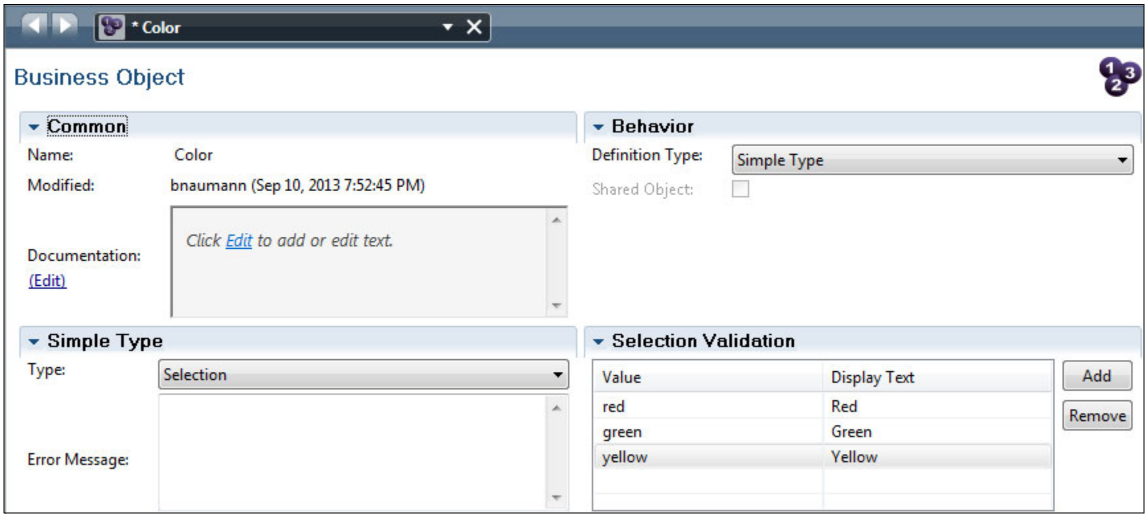
New Business Object

Business objects represent that data that is used during the running of a process.

Name: Color

Figure 3-20 Add a custom typed configuration option

The new type should be a Simple Type of the Selection type. This type makes the configuration option appear as a select box in the Coach View instance configuration. You can specify all potential values in the Selection Validation part, as shown in Figure 3-21.



Name: Color
Modified: bnaumann (Sep 10, 2013 7:52:45 PM)
Documentation: (Edit) Click Edit to add or edit text.
Definition Type: Simple Type
Shared Object:
Type: Selection
Error Message:

Value	Display Text
red	Red
green	Green
yellow	Yellow

Figure 3-21 Create selection type

Now, when you create an instance of the Coach View in the designer by using it on a Coach, you see a drop-down menu with the options that were defined in the configuration section of the Coach View, as shown in Figure 3-22.

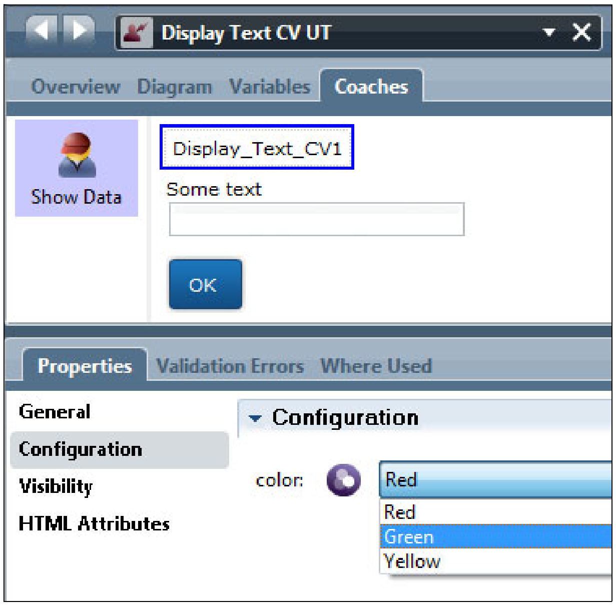
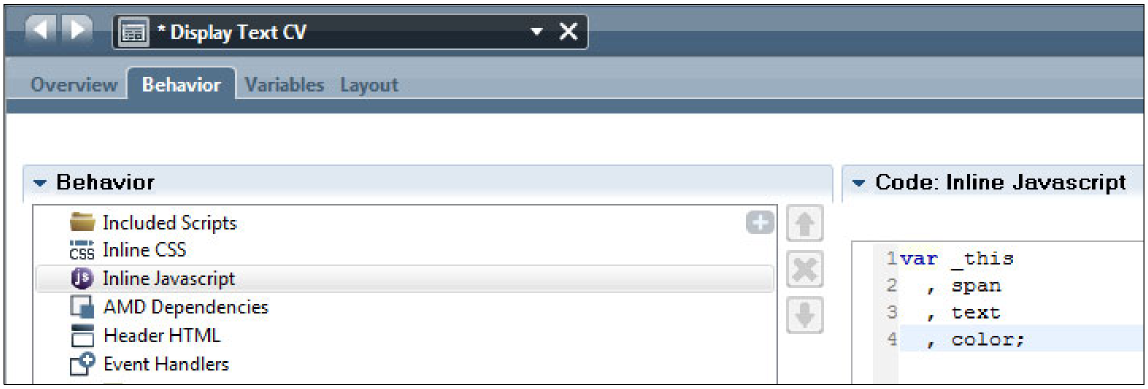


Figure 3-22 Selection type in a Coach View instance configuration

Inline JavaScript

You can now extend the Coach View implementation. In this sample, add a color variable to the Inline JavaScript, as shown in Figure 3-23.



```
var _this
  , span
  , text
  , color;
```

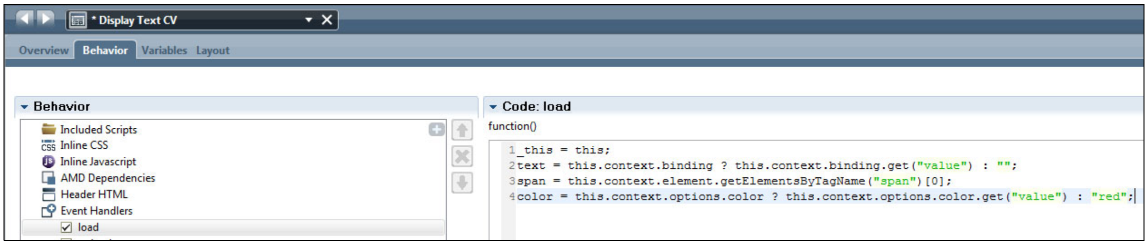
Figure 3-23 Sample Inline JavaScript

Use the following lines for the variable:

```
var _this
    , span
    , text
    , color;
```

Loading

Now, you can extract the initial value that is assigned to the configuration option through data binding. For this sample, add the following lines to the load event handler with a default value of red, as shown in Figure 3-24 and Example 3-2. This value is used if no option is selected by the user.



function()

```
_this = this;
text = this.context.binding ? this.context.binding.get("value") : "";
span = this.context.element.getElementsByTagName("span")[0];
color = this.context.options.color ? this.context.options.color.get("value") : "red";
```

Figure 3-24   Sample Load event handler

Example 3-2   Assigning values during load

```
_this = this;
text = this.context.binding ? this.context.binding.get("value") : "";
span = this.context.element.getElementsByTagName("span")[0];
color = this.context.options.color ?
this.context.options.color.get("value") : "red";
```

Asynchronous Module Definition dependency

Asynchronous Module Definition (AMD) dependencies are used to load libraries into Coach Views.

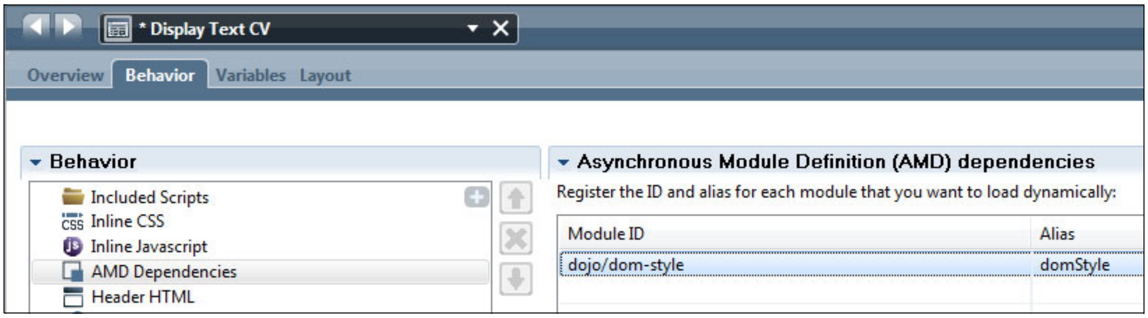
Note: More information about AMD in Dojo can be found at the following website:

https://dojotoolkit.org/documentation/tutorials/1.8/modules/

You can now extend the view event handler to change the color of the text. In this sample, add a Dojo library to the AMD dependencies. Use the dom-style library. For more information about the dom-style library, go to the following website:

https://dojotoolkit.org/reference-guide/1.8/dojo/dom-style.html

To use the library, you create an alias, as shown in Figure 3-25 on page 77.



Register the ID and alias for each module that you want to load dynamically:

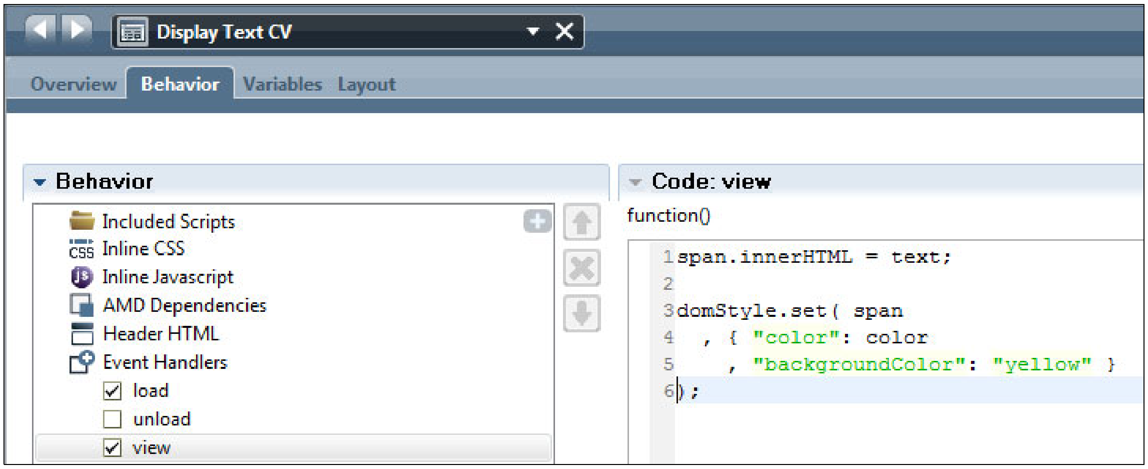
Module ID	Alias
dojo/dom-style	domStyle

Figure 3-25 Defining AMD dependencies

View

Now, you can use the library, as shown in the Dojo reference in the view function.

See Figure 3-26 for a sample view event handler.



function()

```
span.innerHTML = text;

domStyle.set( span
  , { "color": color
    , "backgroundColor": "yellow" }
);
```

Figure 3-26 Sample view event handler

Here is the content of the Dojo reference:

```
span.innerHTML = text;
domStyle.set( span
  , { "color": color
    , "backgroundColor": "yellow" }
);
```

For the span element in this sample, set a style for color and backgroundColor. For the color tag, use the value of the color variable instance. For backgroundColor, hardcode yellow as the value, as shown in Figure 3-26.

If you now run the Human Service that you created before to test the custom Coach View, it shows the dialog box that is shown in Figure 3-27.

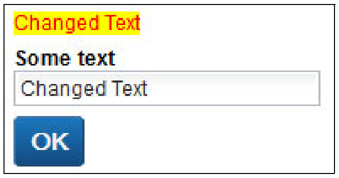


Figure 3-27 Sample dialog box (red text)

You see that the text is colored in red and the background is set to yellow. If you change the configuration option to green and run it again, the text changes to what is shown in Figure 3-28.

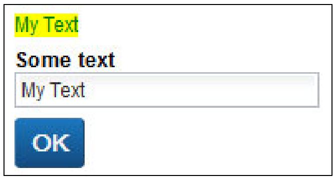


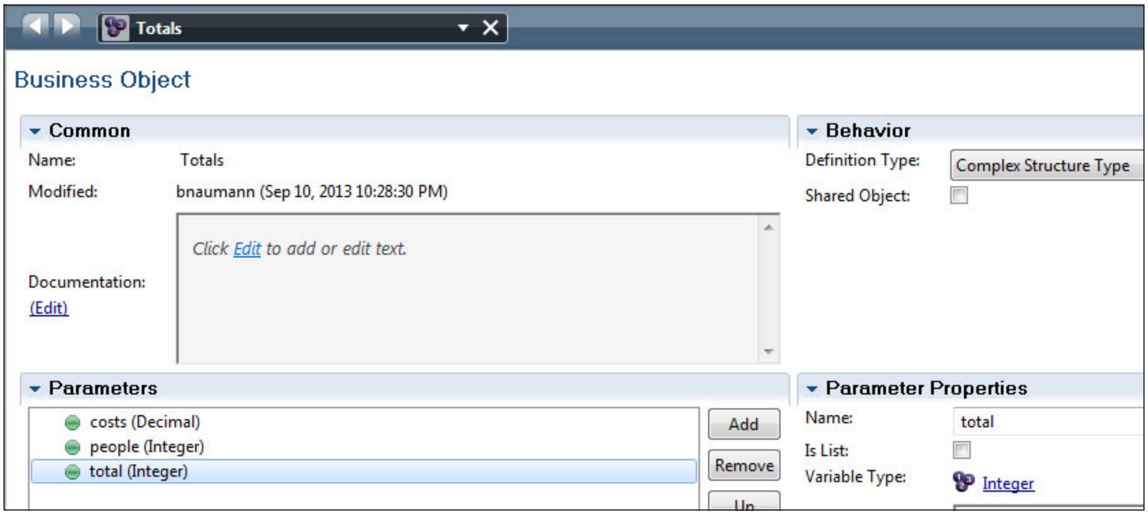
Figure 3-28 Sample dialog box(green text)

3.3 Complex objects

Earlier, we showed how your Coach Views can react to changes in the values of simple variables, such as string typed variables. In contrast, complex object changes are not automatically propagated. If an attribute value changes, other controls that also bind to that attribute are not automatically notified about a value change. Instead, you can use the Coach Framework to have complete control of binding to and propagating changes to better optimize performance.

Sample

In this sample, create a business object named Totals with three attributes, as shown in Figure 3-29 on page 79.



Name: Totals

Modified: bnaumann (Sep 10, 2013 10:28:30 PM)

Documentation: (Edit)

Click Edit to add or edit text.

Definition Type: Complex Structure Type

Shared Object: ☐

- costs (Decimal)
- people (Integer)
- total (Integer)

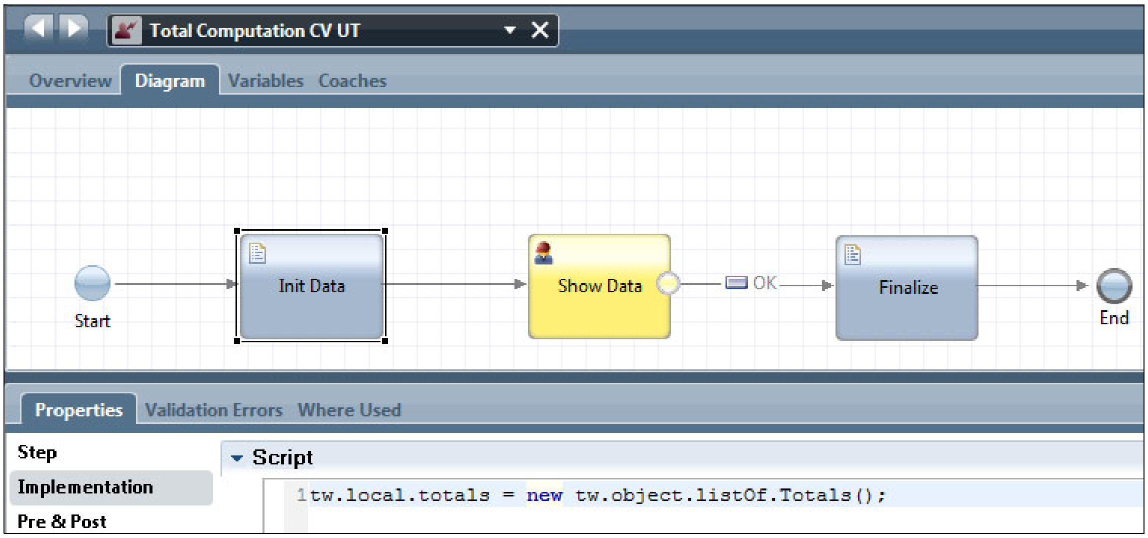
Name: total

Is List: ☐

Variable Type: Integer

Figure 3-29   Complex object definition

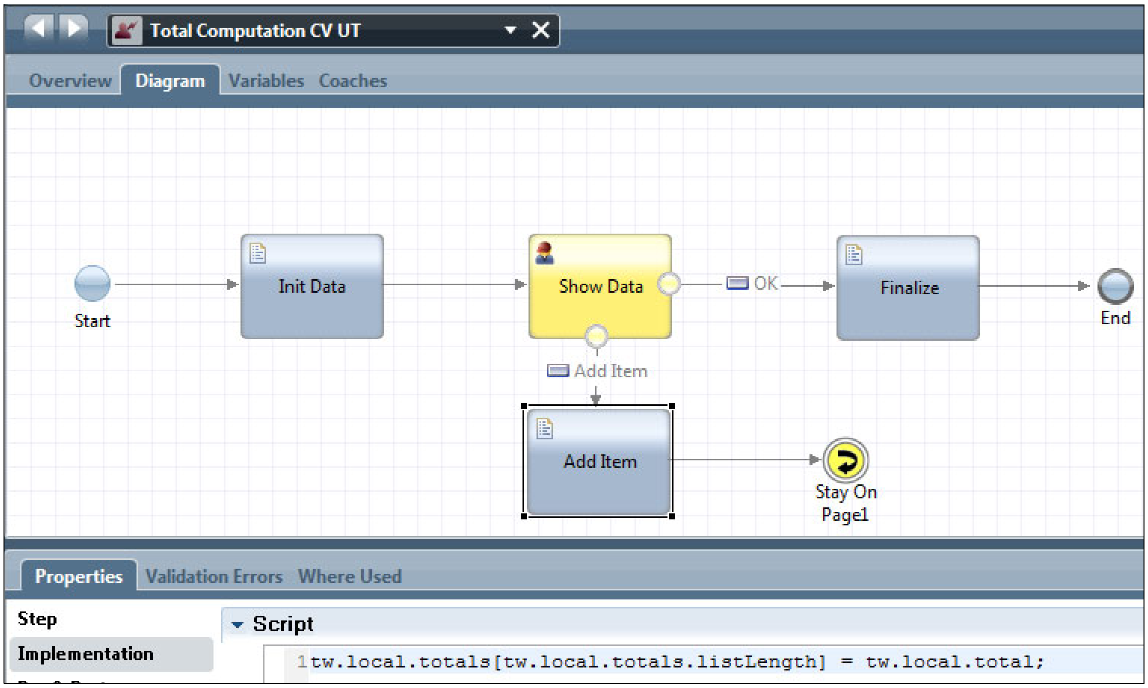
For this sample, create a Human Service. It is a preferred practice to initialize complex objects before using them in a Coach. In this sample, do this in the first server script of the Human Service, as shown in Figure 3-30.



```
tw.local.totals = new tw.object.listOf.Totals();
```

Figure 3-30 Sample Human Service unit test

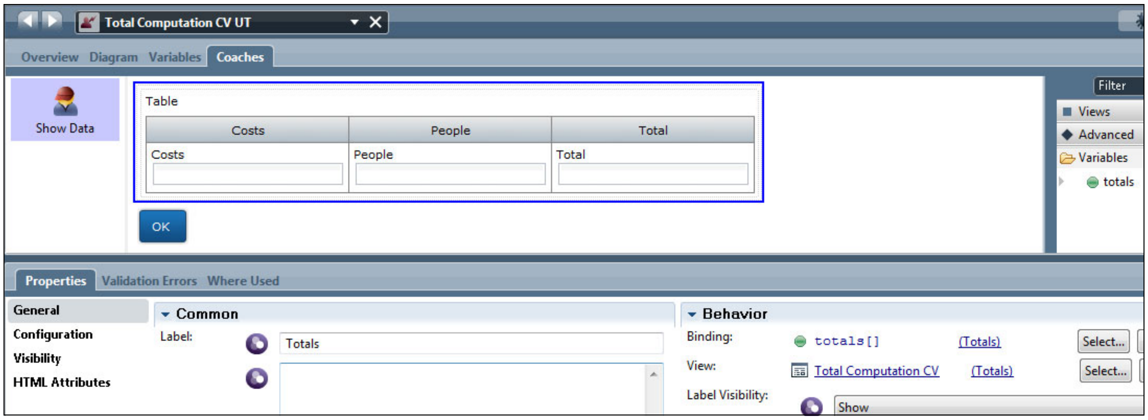
To simulate how the Coach View responds to external data binding changes in this sample, add a server script to the Human Service that adds an item to the totals list. This server script is run when you click the Add Item button that is specified on the Coach. This action adds an item to the list of totals, as shown in Figure 3-31 on page 81.



```
tw.local.totals[tw.local.totals.listLength] = tw.local.total;
```

Figure 3-31 Boundary Events in a Human Service

In this sample, you can now create the Coach View Totals Computation CV for a list of total objects. You can create the Coach View by using the process that is described in 3.1.1, “Accessing business data” on page 58. In this sample, bind it on the Coach to the list of totals that you initialized in the Human Service, as shown in Figure 3-32.

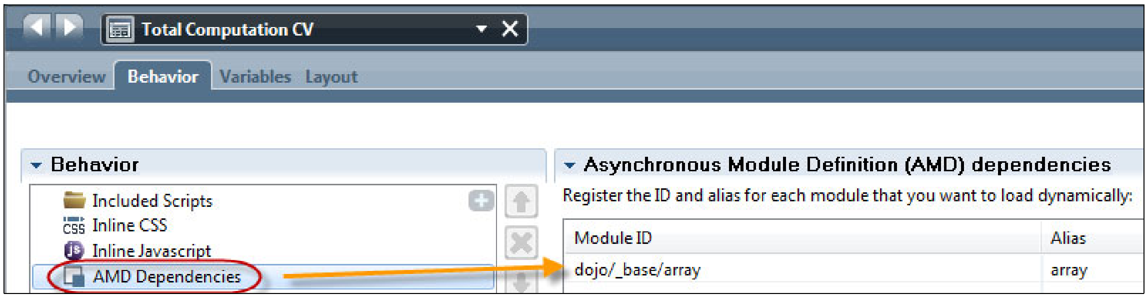


Label: Totals
Binding: totals[] (Totals)
View: Total Computation CV (Totals)
Label Visibility: Show

Figure 3-32 Table binding

AMD dependencies

For compatibility with older browsers such as Internet Explorer 8, use the dojo/_base/array implementation for arrays in this sample. Add an alias for arrays through AMD dependencies, as shown in Figure 3-33. Instead of the Dojo implementation, you might have chosen another Ajax frameworks implementation.



Register the ID and alias for each module that you want to load dynamically:

Module ID	Alias
dojo/_base/array	array

Figure 3-33 AMD dependency for a Dojo array

Inline JavaScript

In the Inline JavaScript section in this sample, which is shown in Example 3-3 on page 83, specify a set of functions that you use to manage handlers that are used to update totals. Use an approach that is suggested in the information center that is found at the following website:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r0m1/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/runload.html

First, create an empty list of handlers. This array contains handlers for all items in the totals list. Here is the empty list:

```
this.connectHandles = [];
```

Create a handler for one item in the totals list and add the handler to the array of item handlers. Use this handler to react to changes in the cost and people attributes. When one of the two values is changed, a total is calculated from these values and the value is updated in the table.

New event handler option: Version 8.5 has a new feature that you can use to reduce the memory footprint of Coach Views that are used multiple times on a Coach. More information about the new Prototype-level event handler option can be found in 3.6, “Preferred practices for developing Coach Views” on page 101.

Example 3-3 Manage the handlers

```
this.setBindingHandler = function(config) {
  var item = config.item;
  this.connectHandles.push(
    item.bindAll( function ( data ) {
      var costs = this.get("costs") ? this.get("costs") : 0;
      var people = this.get("people") ? this.get("people") : 0;
      item.set("total", people * costs);
    }, item)
  );
}
#You can remove all active handles through the remove handles function:
this.removeHandles = function () {
  console.log("removeHandles this", this);
  if (this.bindAllHandle) { this.bindAllHandle.remove(); }
  array.forEach(this.connectHandles, function(handle) {
    if ( handle ) { handle.remove(); }
  });
  this.connectHandles = [];
}
#Add handles for all relevant actions through this function.:
this.addHandles = function () {
  this.bindAllHandle =
this.context.binding.get("value").bindAll(this.bindAllChange, this);
  var totals = this.context.binding.get("value");
  for (var x = 0; x < totals.length(); x++) {
    this.setBindingHandler( { item: totals.get(x) } );
  }
}
#Refresh all handles by removing them and then adding them again:
this.refreshItemBindings = function () {
  this.removeHandles();
  this.addHandles();
}
#If a new item is added to the totals list, refresh the bindings:
this.bindAllChange = function(event) {
  if ( event.newVal !== null &&
       event.newVal !== undefined &&
       event.newVal !== event.oldVal ) {
    this.refreshItemBindings();
  }
}
```

Loading

In this sample, add all relevant handles by calling the addHandles that you defined in Inline JavaScript on load:

```
_this = this;
this.addHandles();
```

Unloading

During the unload in this sample, call the **removeHandles** function that you defined in Inline JavaScript that releases all relevant handles:

```
this.removeHandles();
```

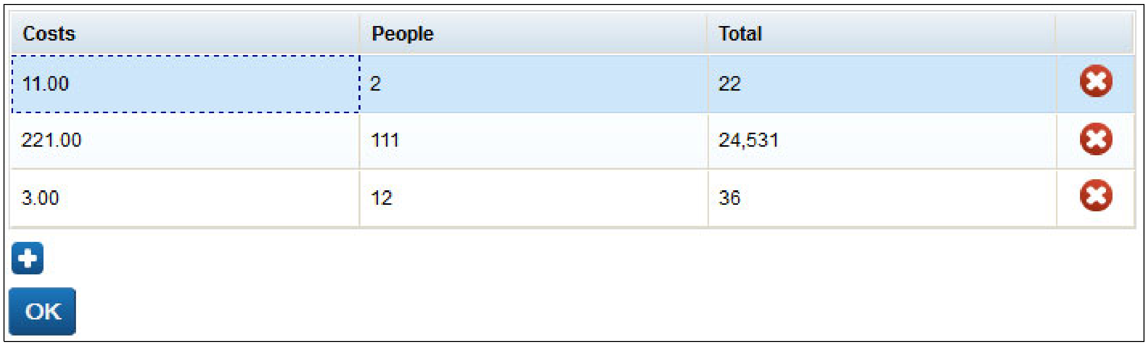
Changes

Refresh all bindings if a change of the business data is detected:

```
if ( event.newVal !== null &&
     event.newVal !== event.oldVal &&
     event.type !== "config" ) {
  this.refreshItemBindings();
}
```

Output

When you run the Human Service, the window looks like Figure 3-34. When you change one of the cost or people values, the handlers that you created in the sample trigger a recalculation of the total.



Costs	People	Total	
11.00	2	22	
221.00	111	24,531	
3.00	12	36	

Figure 3-34 Table sample output

3.4 Back-end integration

If you want to retrieve more data while a Coach is running on the browser, you have a set of options. You can retrieve data through the Ajax service construct or you can trigger a Boundary Event. More information about this topic can be found at the following website:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/modeling/topic/building_ajax_service.html

Boundary Events can be triggered through buttons or the Coach View framework. More information about this topic can be found at the following websites:

- http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/buildcoach/topics/rstockcontrols_button.html
- http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/rcontext.html

3.4.1 IBM BPM Ajax call

IBM BPM provides Ajax Service type services. In Coach Views, you can define an Ajax Service as configuration option. A set of the standard controls, such as the Select control, support the usage of Ajax Services to load content asynchronously.

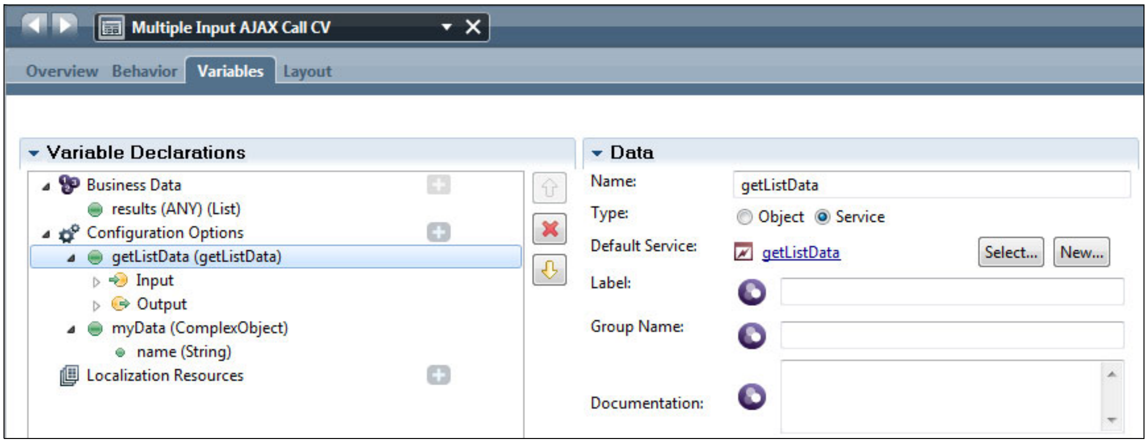
The following sample shows how you can create a custom Coach View that performs an Ajax call by using the Ajax Service. You can use this method to gather business data asynchronously instead of loading it up front on the server side, which can make Coaches more responsive.

When data is loaded before the Coach in a Human Service, this action delays the loading of the Coach. When data is loaded asynchronously, you can load data while the page is being rendered on the client side. Because complex UIs might require processor-intensive work on the client side because of heavy JavaScript usage, it makes sense to start the rendering as soon as possible and to load data in the meanwhile.

Notes: Most browsers have a limit for concurrent Ajax connections. You must keep in mind these limits and avoid making excessive Ajax calls.

Variables

For this sample, create a configuration option of the Service type. You can then specify a default Ajax Service implementation. The selected service defines the interface for potential Ajax Services that can be used with this Coach View, as shown in Figure 3-35.


Multiple Input AJAX Call CV

Overview Behavior Variables Layout

Variable Declarations
- Business Data
  - results (ANY) (List)
- Configuration Options
  - getListData (getListData)
    - Input
    - Output
  - myData (ComplexObject)
    - name (String)
- Localization Resources

Data

Name: getListData

Type: Object [✓ Service]

Default Service: getListData

Label:

Group Name:

Documentation:

Figure 3-35 Coach View with Ajax Services

Inline JavaScript

In the Inline JavaScript section in this sample, specify a set of local variables and the **getParams** function that is used to define the configuration for the Ajax Service. Set the parameters that you send to the Ajax Service as input. Define the callback function that is called when the Ajax Service returns a result. Define an error handler, as shown in Example 3-4.

Note: For versions of IBM BPM before Version 8.5, you must remove the @metadata attribute. In Version 8.5, this might not be necessary anymore.

Example 3-4 Inline JavaScript for the getParams function

```
var _this
  , results
  , myData
  , service;

this.getParams = function(config) {
  var params = {
      params: JSON.stringify(config.inputs)
    , load: function(data) {
      var result = data.results;
      delete result['@metadata'];
      for (var x = 0; x < result.items.length; x++) {
        delete result.items[x]['@metadata'];
      }
      _this.context.binding.set("value", result);
    }
    , error: function(error) {
      console.error("error", error);
    }
  };
  return params;
}
```

Loading

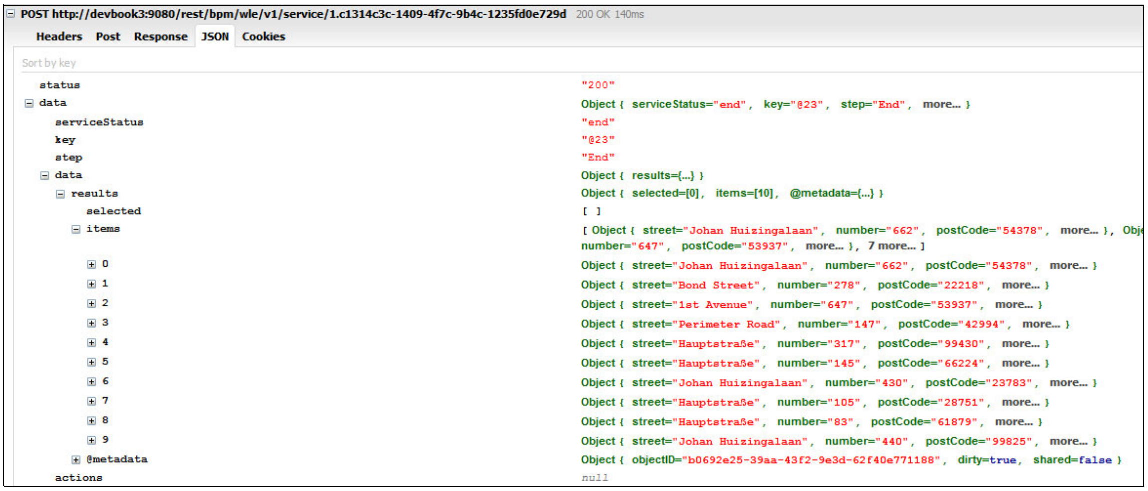
In the load event handler in this sample, populate the local variables that were defined in the Inline JavaScript with the initial data from the variables to which the Coach View instance was bound. Do the same for the service, and in this sample, call the Ajax Service directly on load. Call the **getParams** function that was defined in the Inline JavaScript, as shown in Example 3-5.

Example 3-5 Populating local variables

```
_this = this;
results = this.context.binding ? this.context.binding.get("value") :
null;
myData = this.context.options.myData ?
this.context.options.myData.get("value") : null;
service = this.context.options.getListData;
service(this.getParams( { inputs: { myData: { name: myData.get("name")
} } } ));
```

Calling

In Firebug, you can inspect the Ajax call and look at the response from the server, as shown in Figure 3-36.



```
⊟ POST http://devbook3:9080/rest/bpm/wle/v1/service/1.c1314c3c-1409-4f7c-9b4c-1235fd0e729d   200 OK 140ms
    Headers  Post  Response  JSON  Cookies
  Sort by key
    status                      "200"
 ⊟  data                        Object { serviceStatus="end", key="@23", step="End", more... }
      serviceStatus             "end"
      key                       "@23"
      step                      "End"
   ⊟  data                      Object { results={...} }
    ⊟  results                  Object { selected=[0], items=[10], @metadata={...} }
         selected               [ ]
     ⊟  items                   [ Object { street="Johan Huizingalaan", number="662", postCode="54378", more... }, Obj
                                number="647", postCode="53937", more... }, 7 more... ]
      ⊞ 0                       Object { street="Johan Huizingalaan", number="662", postCode="54378", more... }
      ⊞ 1                       Object { street="Bond Street", number="278", postCode="22218", more... }
      ⊞ 2                       Object { street="1st Avenue", number="647", postCode="53937", more... }
      ⊞ 3                       Object { street="Perimeter Road", number="147", postCode="42994", more... }
      ⊞ 4                       Object { street="Hauptstraße", number="317", postCode="99430", more... }
      ⊞ 5                       Object { street="Hauptstraße", number="145", postCode="66224", more... }
      ⊞ 6                       Object { street="Johan Huizingalaan", number="430", postCode="23783", more... }
      ⊞ 7                       Object { street="Hauptstraße", number="105", postCode="28751", more... }
      ⊞ 8                       Object { street="Hauptstraße", number="83", postCode="61879", more... }
      ⊞ 9                       Object { street="Johan Huizingalaan", number="440", postCode="99825", more... }
     ⊞ @metadata                Object { objectID="b0692e25-39aa-43f2-9e3d-62f40e771188", dirty=true, shared=false }
      actions                   null
```

Figure 3-36 Example Ajax call result

3.4.2 Boundary Event

Simple Boundary Events happen if you click a button. If the outgoing path on the Coach that corresponds to the button is going to another Coach, you are taken to a new page where the next Coach is rendered, as shown in Figure 3-37.

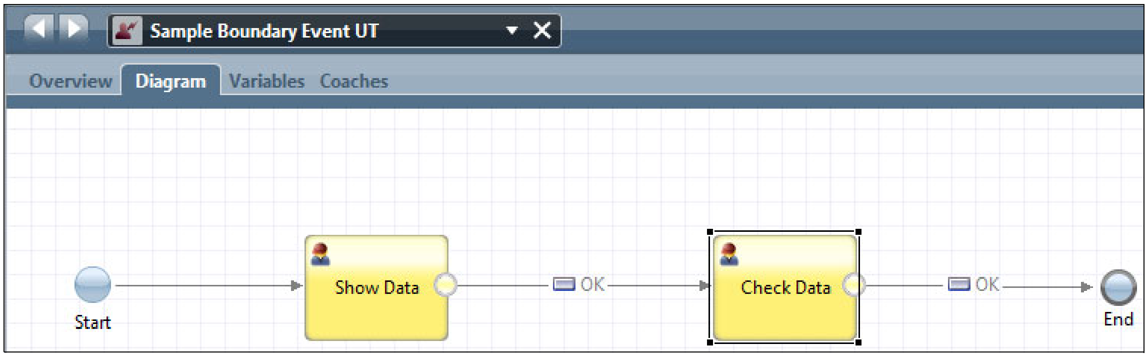


Figure 3-37 Simple Coach flow

If you go to a service instead of to another Coach, the behavior is different. No page transfer happens, just data that was changed or added by the service is sent back to the Coach through an Ajax call that is done by the Coach View framework.

In Figure 3-38, the Get Data server script is called when you click **Fetch Data** in the Coach, as shown in Figure 3-39.

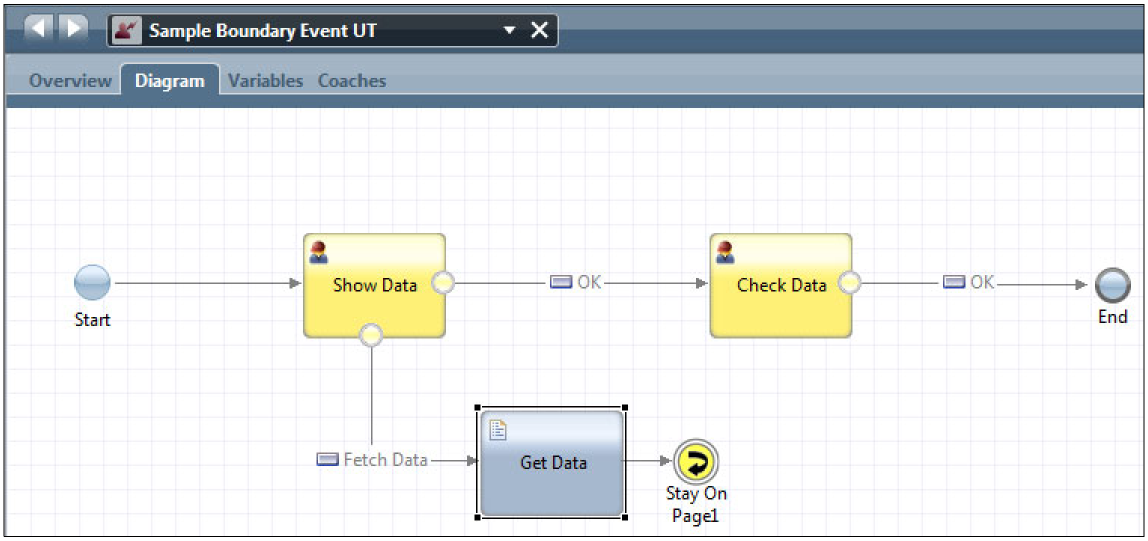


Figure 3-38 Coach flow with Boundary Event

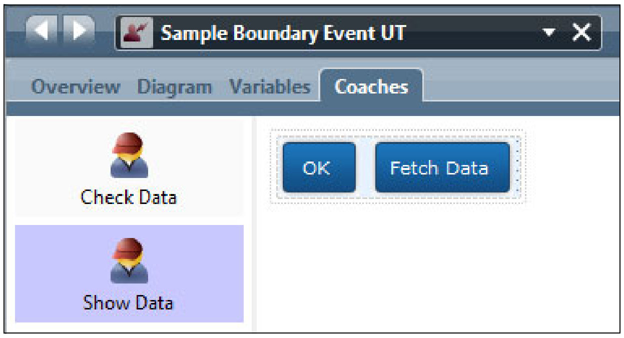
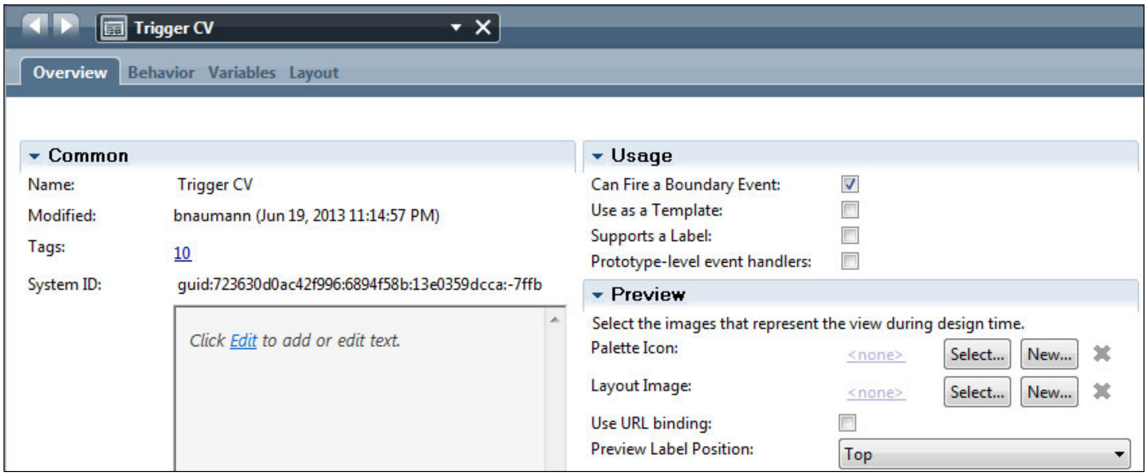


Figure 3-39 Coach designer

Triggering

You can also programmatically trigger a Boundary Event from within a Coach View by selecting the **Can Fire a Boundary Event** option in the Overview tab, as shown in Figure 3-40.



Name: Trigger CV
Modified: bnaumann (Jun 19, 2013 11:14:57 PM)
Tags: 10
System ID: guid:723630d0ac42f996:6894f58b:13e0359dcca:-7ffb
Click Edit to add or edit text.

Can Fire a Boundary Event: ✓
Use as a Template: ☐
Supports a Label: ☐
Prototype-level event handlers: ☐

Select the images that represent the view during design time.
Palette Icon: <none>
Layout Image: <none>
Use URL binding: ☐
Preview Label Position: Top

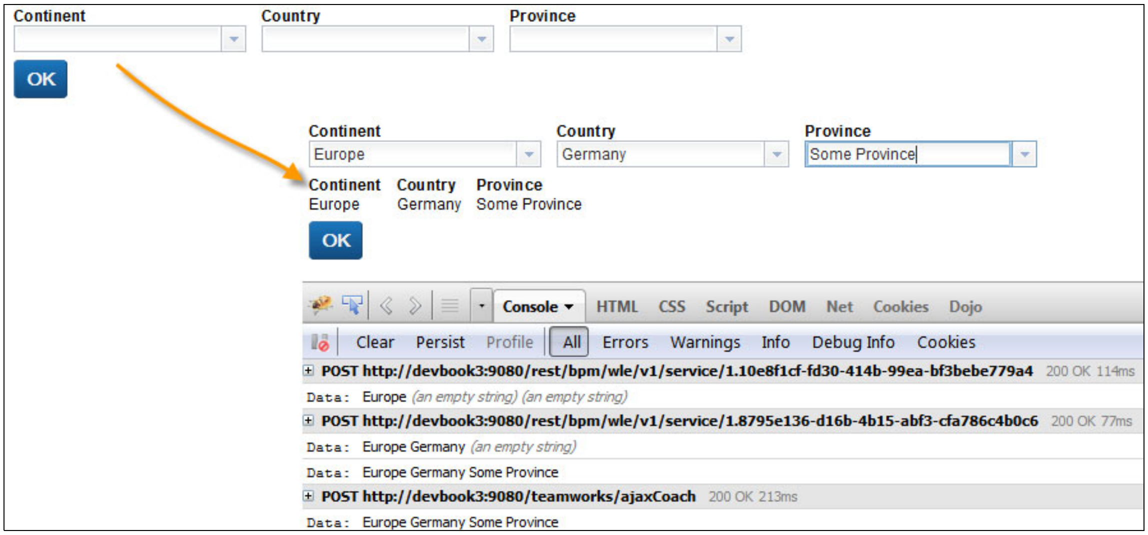
Figure 3-40 Enable triggers

This option triggers the Boundary Event programmatically depending on the conditions that you implement. You can use the following API call to trigger a Boundary Event:

```
this.context.trigger();
```

Sample

Here you can look at a sample usage of the trigger API. In this sample, we want to show only a Coach View if three dependent Select boxes are filled, as shown in Figure 3-41 on page 91.



```
POST http://devbook3:9080/rest/bpm/wle/v1/service/1.10e8f1cf-fd30-414b-99ea-bf3bebe779a4   200 OK 114ms
Data:  Europe (an empty string) (an empty string)
POST http://devbook3:9080/rest/bpm/wle/v1/service/1.8795e136-d16b-4b15-abf3-cfa786c4b0c6   200 OK 77ms
Data:  Europe Germany (an empty string)
Data:  Europe Germany Some Province
POST http://devbook3:9080/teamworks/ajaxCoach   200 OK 213ms
Data:  Europe Germany Some Province
```

Figure 3-41 Trigger sample in Firebug

The Human Service is shown in Figure 3-42. When the trigger API is called, the Change Visibility server script is called.

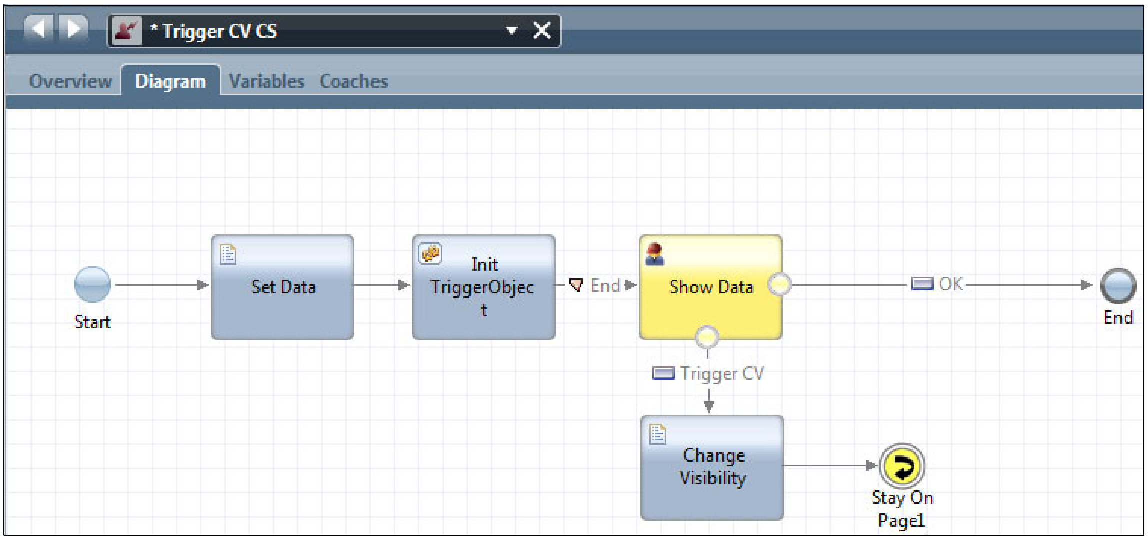


Figure 3-42 Triggers in a Human Service

In the Change Visibility option, toggle the visibility for the lower Coach View as follows:

```
tw.local.visibility = tw.local.visibility == "NONE" ? "DEFAULT" :
"NONE";
```

Inline JavaScript

In this sample, define the local variables in the Inline JavaScript section. as follows:

```
var visibility
   , continent
   , country
   , province;
```

When a value changes, the **`changeBinding`** function is called. First, check if a value changed. Only if a change of a relevant attribute is detected do you update the local variable with the new value and show that a change happened by setting the helper variable `doTrigger` to `true`.

If you see that one of the relevant attributes changed, update the local private variable and update the `doTrigger` variable to show that something changed, as shown in Example 3-6.

Example 3-6 Populating variables

```
this.changeBinding = function (event) {
  var binding = this.context.binding.get("value");
  var doTrigger = false;
  if ( event.type !== "config" &&
       event.newVal  &&
       event.newVal != event.oldVal ) {
    if (event.property === "continent") {
      continent = event.newVal;
      doTrigger = true;
    } else if (event.property === "country") {
      country = event.newVal;
      doTrigger = true;
    } else if (event.property === "province") {
      province = event.newVal;
      doTrigger = true;
    }
  }
  console.log("Data: ", continent, country, province);
```

In the second block, check if all private values have a value and that there was a change of one of the values. Check the helper variable `doTrigger` to detect a change. Only when all the conditions match do you call the trigger API, as shown in Example 3-7.

Example 3-7 Checking for value changes

```
if ( this.context &&
       continent !== null &&
       continent !== "" &&
       country !== null &&
       country !== "" &&
       province !== null &&
      province !== "" &&
       doTrigger ) {
    this.context.trigger();
  }
}
```

Loading

In this sample, gather information from the bindings and initialize the local variables with the bound values, as shown in Example 3-8.

Example 3-8 Initialize local variables

```
var binding = this.context.binding.get("value");
visibility = this.context.options.visibility ?
this.context.options.visibility.get("value") : "";
continent = binding ? binding.get("continent") : null;
country = binding ? binding.get("country") : null;
province = binding ? binding.get("province") : null;
```

Bind to the changes of the three attributes. Call the **`changeBinding`** function that you defined in the Inline JavaScript when a change of the attribute values is detected, as shown in Example 3-9.

Example 3-9 Use of the changeBinding function

```
if (binding) {
  this.continent = binding.bind("continent", this.changeBinding, this);
  this.country = binding.bind("country", this.changeBinding, this);
  this.province = binding.bind("province", this.changeBinding, this);
}
```

Unloading

In this sample, release the handlers in the unbind event handler, as shown in Example 3-10.

Example 3-10 Releasing the handlers

```
if (this.continent) { this.continent.unbind(); }
if (this.country) { this.country.unbind(); }
if (this.province) { this.province.unbind(); }
```

3.4.3 Dojo XMLHttpRequest (XHR)

Instead of using the Ajax Service construct directly, you can also call it through the IBM BPM Representational State Transfer (REST) API. More information about this topic can be found at the following address:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/develop/topics/tajaxservice.html

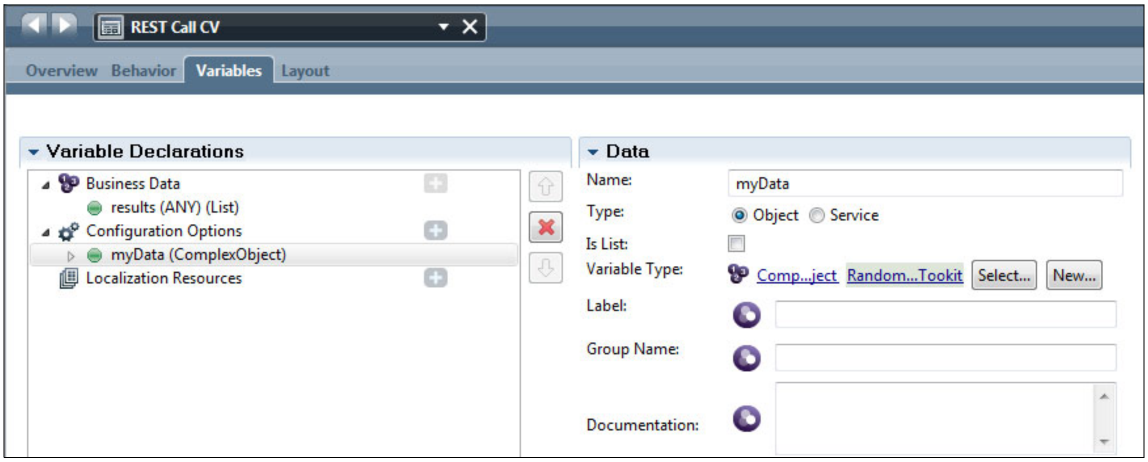
This method is useful if you want to have more control over how the call is done. In this sample, use dojo.xhrPost, which allows you, for example, to define a custom timeout for a call to a REST API. More information about this topic can be found at the following website:

http://dojotoolkit.org/reference-guide/1.8/dojo/xhrPost.html

Let us look at a sample that is the same as the Ajax Service sample before, but this time use Dojo to do the REST call.

Variables

In the variables section, do not define the Ajax Service because you are using the REST API directly, as shown in Figure 3-43 on page 95.



Variable Declarations
- Business Data
  - results (ANY) (List)
- Configuration Options
  - myData (ComplexObject)
- Localization Resources

Data

Name: myData

Type: [✓ Object] Service

Is List:

Variable Type: Comp...ject Random...Tookit Select... New...

Label:

Group Name:

Documentation:

Figure 3-43   Configuration option with complex object type

Inline JavaScript

Define the local variables that you use in the other event handlers:

```
var _this
   , results
   , myData
   , xhrArgs
   , deferred;
```

Specify the **getXhrArgs** function that you use to prepare the arguments for the actual **xhr** call. Specify the URL of the REST API, the parameters that are sent through POST, and how you want to handle the response, as shown in Example 3-11 on page 96.

Note: You can find more information about this IBM BPM REST call in the information center that is found at the following address:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.ref.doc/rest/bpmrest/rest_bpm_wle_v1_service_instanceid_post_start.htm

Preferably, the URL should not be hardcoded because, for example, the /rest/ part of the IBM BPM REST URL might vary by environment.

Example 3-11 shows that you can extend the **xhr** arguments. For example, you can add the timeout setting here.

Example 3-11 Extending the xhr arguments

```
this.getXhrArgs = function(config) {
  return {
      url: "/rest/bpm/wle/v1/service/RBO1@getAddressData"
    , content: { accept: "application/json"
               , params: JSON.stringify(config.inputs)
               , createTask: false
               , parts: "all"
               , action: "start"
               }
    , handleAs: "json"
  }
};
```

Ajax Service as configuration option: If you want to refer to an Ajax Service that was defined as configuration option, you can extract the URL for the service through the following API call:

```
this.context.options.myOptionService.url
```

Loading

During the loading process, populate the local variables with data that is bound to the Coach View instance, as shown in Example 3-12.

Example 3-12 Populating local variables

```
_this = this;
results = this.context.binding ? this.context.binding.get("value") :
null;
myData = this.context.options.myData ?
this.context.options.myData.get("value") : null;
```

Call **dojo.xhrPost** and provide it with the arguments that you prepare by calling the **getXhrArgs** function that was defined in the inline JavaScript, as shown in Example 3-13.

Example 3-13 Calling dojo.xhrPost

```
deferred = dojo.xhrPost(this.getXhrArgs( { inputs: { myData: { name:
myData.get("name") } } } ));
```

The **xhrPost** call returns a deferred object; use this object to define the callback function and an error handler. When **xhr** returns a result, update the local binding with the returned value, as shown in Example 3-14.

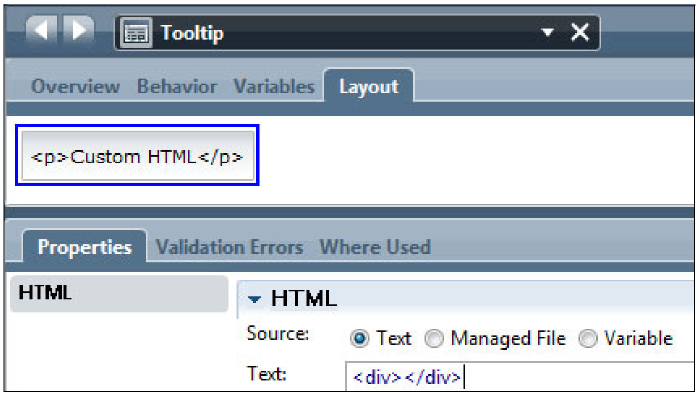
Example 3-14 Updating the local binding

```
deferred.then( function(value){
  _this.context.binding.set("value", value.data.data.results.items);

} , function(error){
    console.error("error", error);
});
```

3.5 DOM manipulation

If you want to manipulate the content of your Coach programmatically, the Dojo dom-construct library can be a significant aid. You can easily, for example, add a span programmatically to your custom Coach View. You can also use any other Ajax frameworks in the DOM manipulation library.

For example, if you want to show an image in a Coach View, you can use the library to create an image tag and to place the image, for example, into a `div`. To build the sample, put a `div` tag into a Custom HTML block in the Layout section, as shown in Figure 3-44.



`<p>Custom HTML</p>`

HTML

Source: [✓ Text] Managed File Variable

Text: `<div></div>`

Figure 3-44 Define a base div element

Inline JavaScript

You can now create the local variables in the Inline JavaScript section:

```
var _this
   , div
   , img;
```

Loading

In the load event handler, take the `div` element and create an `img` element with `domConstruct.create`. Then, you can place the new element in to the existing `div` element, as shown in Example 3-15.

Example 3-15 Creating new elements

```
div = this.context.element.getElementsByTagName("div")[0];
img = domConstruct.create("img");
domConstruct.place(img, div, "last");
```

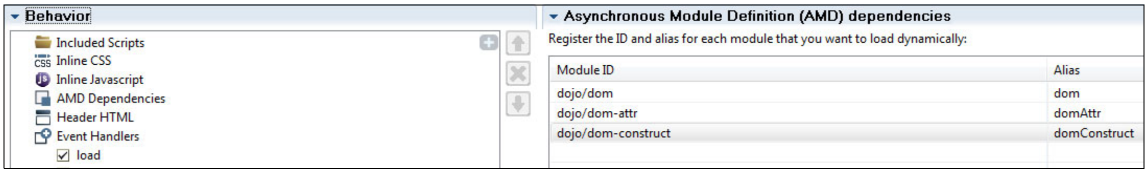
You can use the dom-attr Dojo library to assign the `src` attribute to the `img` element. You can refer to an image in the managed files. You can use the **`com_ibm_bpm_coach.getManagedAssetUrl`** framework function to obtain the URL where the image can be found, as shown in Example 3-16.

Example 3-16 Assigning a source attribute

```
domAttr.set(img, "src", com_ibm_bpm_coach.getManagedAssetUrl(
"dialog-information-2.png" , com_ibm_bpm_coach.assetType_WEB ) );
```

AMD dependencies

You must add the following AMD dependencies before you can run the Coach View the first time, as shown in Figure 3-45.



Asynchronous Module Definition (AMD) dependencies

Register the ID and alias for each module that you want to load dynamically:

Module ID	Alias
dojo/dom	dom
dojo/dom-attr	domAttr
dojo/dom-construct	domConstruct

Figure 3-45 AMD dependencies

Output

When you run a Coach with this Coach View, you see the dialog box that is shown in Figure 3-46 on page 99.

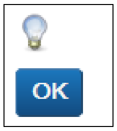


Figure 3-46 Output

3.5.1 Content box

A content box allows you to add content to a custom Coach View at design time. So, if you want to allow a user of your Coach View to specify the content of some parts of the Coach View in Process Designer, you can use the content box control. More information about this topic can be found at the following website:

`http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/buildcoach/topics/ccontrols.html`

To show how you can use the content box, you can extend the sample by adding a tooltip that is shown when you hover your cursor over the image. The content of the tooltip is modeled at design time in Process Designer in a content box, as shown in Figure 3-47.

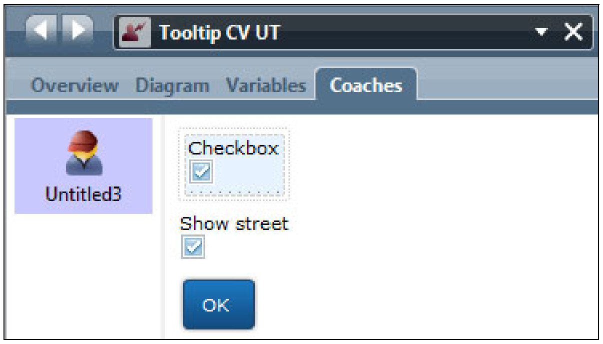


Figure 3-47 Content box

At run time, the content that is specified in the content box is moved in to the tooltip, as shown in Figure 3-48.

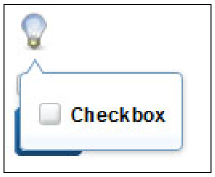


Figure 3-48 Output with tooltip

Inline JavaScript

Define the following local variables:

```
var div
   , img
   , myDialog
   , myButton
   , contentBox;
```

Loading

In the load event handler, you can search for the content box in the scope of your Coach View instance. In this case, use the **querySelector** function and then search for the ContentBox class that is assigned to the content box `div` element. Then, create a TooltipDialog instance following the examples from the Dojo toolkit website:

https://dojotoolkit.org/reference-guide/1.8/dijit/TooltipDialog.html

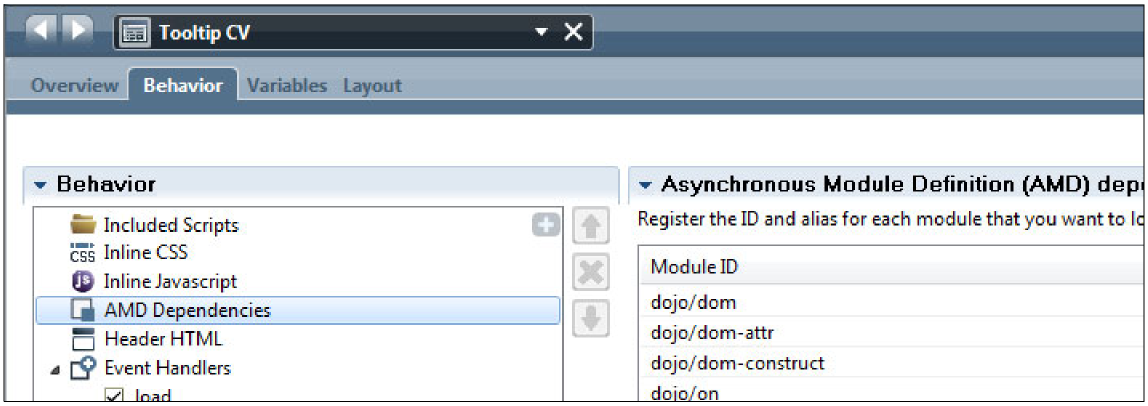
Now, move the content box to the dialog instance by assigning the dom node to the content attribute, as shown in Example 3-17.

Example 3-17 Assigning the dom node to the content attribute

```
div = this.context.element.getElementsByTagName("div")[0];
img = domConstruct.create("img");
domConstruct.place(img, div, "last");
domAttr.set(img, "src", com_ibm_bpm_coach.getManagedAssetUrl(
"dialog-information-2.png"
                                                           ,
com_ibm_bpm_coach.assetType_WEB ) );
contentBox = this.context.element.querySelector(".ContentBox");
myDialog = new TooltipDialog( { onMouseLeave: function(){
popup.close(myDialog); } });
myDialog.attr('content', contentBox);
on( img, 'mouseover', function(){
  popup.open( { popup: myDialog
            , around: img
            });
});
```

AMD dependencies

You must add some additional AMD dependencies for the Tooltip Dialog control. You must do this so that the correct dependencies are imported and accessible within your Coach View, as shown in Figure 3-49 on page 101.



Register the ID and alias for each module that you want to lo

Module ID
dojo/dom
dojo/dom-attr
dojo/dom-construct
dojo/on

Figure 3-49 AMD dependencies for Tooltip Dialog control

3.5.2 Sample download

You can find the samples of this chapter in the IBM BPM wiki at the following website:

`http://bpmwiki.blueworkslive.com/display/samples/Coach+View+Samples`

3.6 Preferred practices for developing Coach Views

This section is a compilation of preferred practices that should be adopted when you create custom Coach Views.

3.6.1 Cascading style sheets

This section covers the guidelines for using CSS in custom Coach Views.

Making all styles extensible

When you create custom Coach Views, you can show the look and feel of your Coach Views through CSS classes. The users of your Coach View can easily change the look and feel of your Coach View without having to change the source code (copy and modify).

When you author Coach Views, do not make style definitions inline, as shown in the following example:

```
<div style="background:red;"> content
</div>
```

Instead, use explicit classes, as shown in the following example:

```
.myClass { background: red; }
```

Then, use these classes when defining your Coach View, as shown in the following example:

```
<div class="myClass">
  content
</div>
```

The users of your Coach View can change the look and feel by overriding the class, as shown in the following example:

```
 .myClassOverride .myClass
 { background: blue; }
```

Inheriting the look and feel from the Stock Coach Views

When you create a Coach View, it is a preferred practice to match the look and feel of the Stock Coach Views (the Coach Views that come with IBM BPM). For example, if you want to match the look and feel of the Stock Coach Views, you must use the IBM BPM Coach View framework CSS style that is included in the `coach_ng_controls.cc` CSS file.

Two different approaches can be taken. Which approach you take depends on whether the Stock Coach View that you want to emulate is implemented as HTML (that is, the Button Coach View) or as a Dojo dijit (that is, the Text Coach View).

Emulating an HTML-based Stock Coach View

If the Stock Coach View you want to emulate was implemented as HTML (that is, the Button Coach View), complete the following steps:

1. Include in your Coach View the CSS that defines the look and feel of the Stock Coach Views. To do so, add `coach_ng_controls.css` to the Included Scripts folder, as shown in Figure 3-50 on page 103.

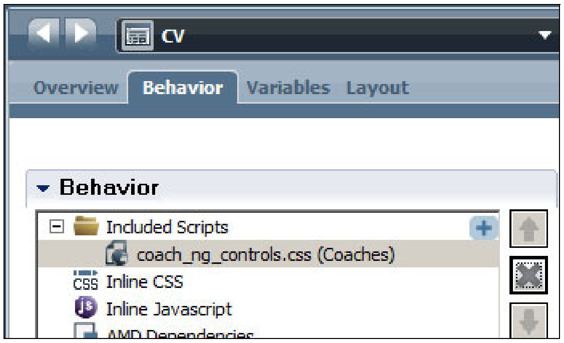


Figure 3-50 Included Scripts

2. To apply the styles you want you use in your Coach View, add the Custom HTML Coach View, as shown in Figure 3-51.

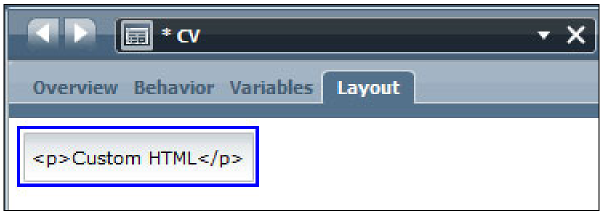
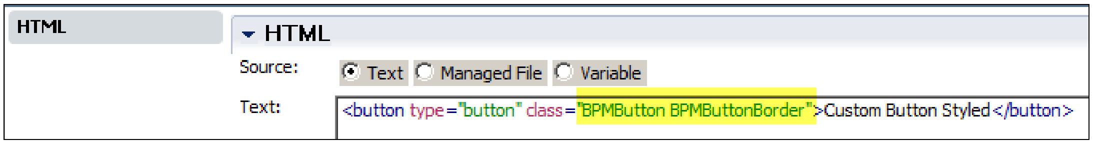


Figure 3-51 Custom HTML

3. Then, use the CSS classes (BPMButton and BPMButtonBorder, as shown in Figure 3-52) that are defined in `coach_ng_control.css` (this file can be found in the Coaches toolkit in the Files folder).



Source: (● Text) ○ Managed File ○ Variable

Text:

```
<button type="button" class="BPMButton BPMButtonBorder">Custom Button Styled</button>
```

Figure 3-52 Adding CSS classes

To determine what class to select, either use browser tools such as Firebug or open `coach_ng_controls.css`.

Formatted content: In Version 8.0.1.2 and Version 8.5.0.1 or higher, this CSS file is put through a Dojo build, so it is compressed and comments are removed. To see the formatted content, use a web browsers' built-in debugger (for example, Firebug) or CSS formatting tools.

Figure 3-53 shows three Coach Views that are rendered in a web browser.

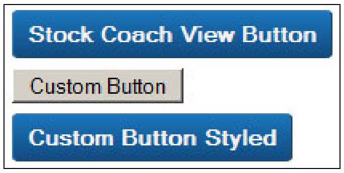


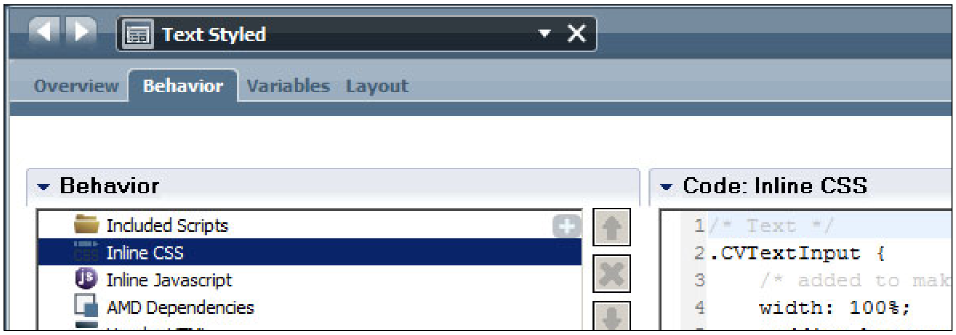
Figure 3-53 Custom Button Coach View - runtime view

The first one is a Stock Coach View Button. The second one is a Custom Button Coach View without any CSS styling applied. The third one is a Custom Button Styled Coach View with CSS classes from the `coach_ng_control.css` file applied so that the custom Coach View mimics the look and feel of the Stock Coach Views.

Emulating a Dojo dijit-based Stock Coach View

If the Stock Coach View that you want to emulate was implemented as a Dojo dijit (that is, Text Coach View), complete the following steps:

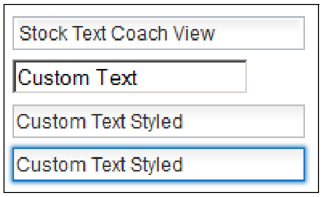
1. Use Firebug to look for the CSS classes and the `div` structure that is used in the Stock Coach View that you want emulate.
2. Open the `coach_ng_controls.css` file (you can find it in the Files folder in the Coaches toolkit) and copy the relevant CSS classes from that file in to the Inline CSS editor, as shown in Figure 3-54.



```
/* Text */
.CVTextInput {
    /* added to mak
    width: 100%;
```

Figure 3-54 AMD dependencies for tooltip

Figure 3-55 on page 105 shows four Coach Views that are rendered in a web browser.


Stock Text Coach View

Custom Text

Custom Text Styled

Custom Text Styled

Figure 3-55 Custom Text Coach View - runtime view

The first one is a Stock Text Coach View. The second one is a Custom Text Coach View without any CSS styling applied. The third one is a Custom Text Styled Coach View with CSS classes from the `coach_ng_control.css` file applied so that the custom Coach View mimics the look and feel of the Stock Coach Views. The fourth one is also a Custom Text Styled Coach View with CSS classes applied, which shows that the selected look and feel mimics the Stock Text Coach View.

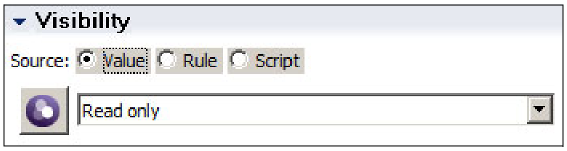
Example 3-18 shows the CSS code that is used to accomplish the look and feel of the Stock Text Coach View.

Example 3-18 CSS code

```
/* Text */
.CVTextInput {
    /* added to make style as close to stock text control */
    width: 100%;
    padding-bottom: 2px;
    padding-top: 2px;

    /* from coach_ng_cnotrols.css */
    background-color:#FFF;
    border:1px solid #bbb;
    color:#2d2d2d;
    background-image: linear-gradient(bottom, rgb(255,255,255) 50%, 
rgb(239,239,239) 100%);
    background-image: -o-linear-gradient(bottom, rgb(255,255,255) 50%, 
rgb(239,239,239) 100%);
    background-image: -moz-linear-gradient(bottom, rgb(255,255,255) 
50%, rgb(239,239,239) 100%);
    background-image: -webkit-linear-gradient(bottom, rgb(255,255,255) 
50%, rgb(239,239,239) 100%);
   background-image: -ms-linear-gradient(bottom, rgb(255,255,255) 50%, 
rgb(239,239,239) 100%);
    background-image: -webkit-gradient(
         linear,
         left bottom,
        left top,
        color-stop(.5, rgb(255,255,255)),
        color-stop(1, rgb(239,239,239))
        );
    /* include rules from .dijitReset */
    font-family: inherit;
    font-size: inherit;
    font-size-adjust: inherit;
    font-stretch: inherit;
    font-style: inherit;
    font-variant: inherit;
    font-weight: inherit;
    line-height: normal;
}
/* from coachng_cnotrols.css */
.CVTextInput:hover {
      border-color:#999;
}
/* from coachng_cnotrols.css */
.CVTextInput:focus {
      border:1px solid #1b75bc;
      -moz-box-shadow: 0 0 2px 1px rgba(27,118,188,.9);
      -webkit-box-shadow: 0 0 2px 1px rgba(27,118,188,.9);
      box-shadow: 0 0 2px 1px rgba(27,118,188,.9);
}
/* Sets the width of the text control */
.CVTextContent {
    width: 15em;
}
.CVTextLabel {
      color:#000;
      font-weight:bold;
      letter-spacing:.3px;
}
```

3.6.2 Coach View visibility

When you create a Coach View, the Visibility property automatically is made available. In the Visibility page, the users of your Coach View can set the runtime visibility properties in the Coach by using Value, Rule, or Script options, as shown in Figure 3-56 on page 107.

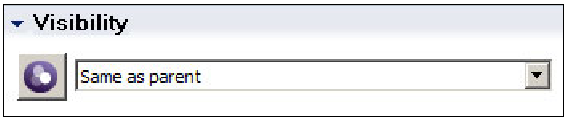


Source: [✓ Value] Rule Script

Read only

Figure 3-56 Visibility - in Coaches

These visibility options are available only for coach view instances in a Coach, and *not* Coach Views. Coach Views have only the “value” visibility style, as shown in Figure 3-57.



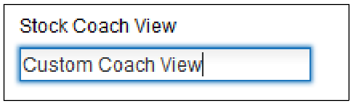
Same as parent

Figure 3-57 Visibility - in Coach Views

Your custom Coach View must implement these visibility settings.

Figure 3-58 shows two Coach Views that are rendered in a web browser. In the parent Coach that includes these Coach Views, the Visibility was set to `Read only` for both.

The top Coach View is a Stock Text Coach View. It acknowledges the visibility setting, but it is not editable. The second one is a custom Text Coach View that did not implement the visibility logic; it is editable even though the Visibility was set to `Read only`.



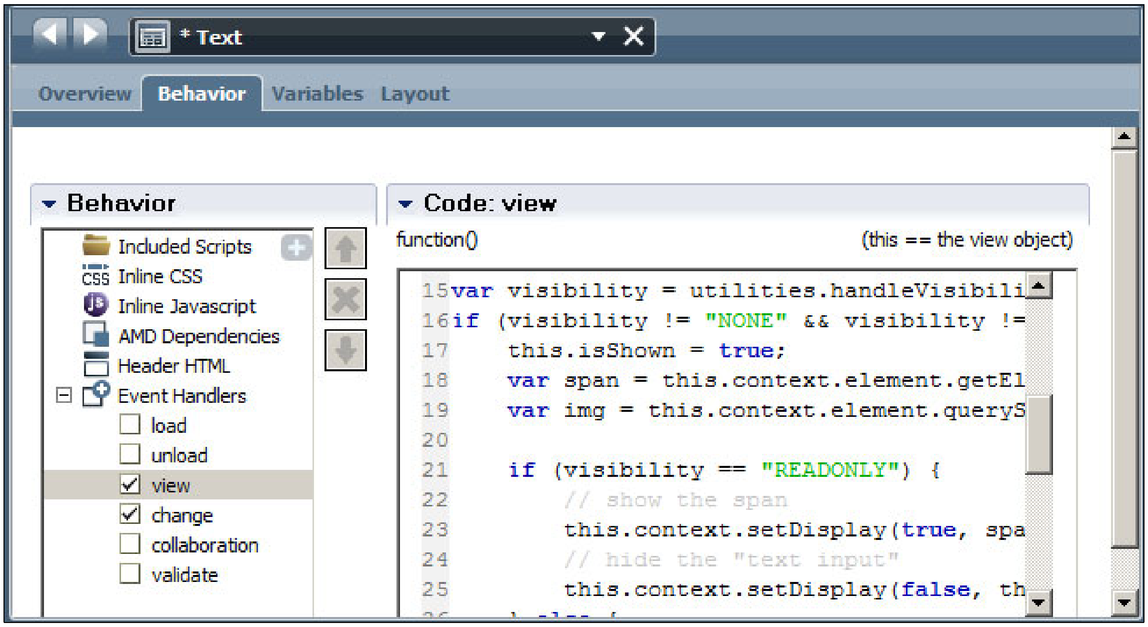
Stock Coach View

Custom Coach View

Figure 3-58 Read only visibility at run time

Implementing visibility in a custom Coach View

Internally, the visibility settings are `DEFAULT` (the code equivalent of same as parent), `EDITABLE`, `REQUIRED`, `READONLY`, `NONE`, and `HIDDEN`. When you create a custom Coach View, you must provide support for the visibility settings in the view() handler, as shown in Figure 3-59.



```
var visibility = utilities.handleVisibili
if (visibility != "NONE" && visibility !=
    this.isShown = true;
    var span = this.context.element.getEl
    var img = this.context.element.queryS

    if (visibility == "READONLY") {
        // show the span
        this.context.setDisplay(true, spa
        // hide the "text input"
        this.context.setDisplay(false, th
```

Figure 3-59 Implementing visibility in a Coach View

A recommended way to implement the visibility logic is to copy the code from a Stock Coach View that is similar to your custom Coach View.

The visibility logic must be implemented in the view() handler. It is not necessary to implement the change() handler. If the change() handler is not implemented, the view() handler is called if a change occurs in the bindings of the Coach View.

As an example, examine the visibility logic in the Text Stock Coach View's view() handler, which is shown in Example 3-19.

Example 3-19 Visibility logic in the Text Stock Coach View's view() handler

```
var visibility = utilities.handleVisibility(this.context, null,
[this.comboBox]);
if (visibility != "NONE" && visibility != "HIDDEN" || this.isShown ==
undefined) {
   this.isShown = true;
   var span = this.context.element.getElementsByTagName("span")[0];
   var img = this.context.element.querySelector("img");
   if (visibility == "READONLY") {
      // show the span
      this.context.setDisplay(true, span.parentNode);
      // hide the "text input"
      this.context.setDisplay(false, this.comboBox.domNode);
   } else {
      // hide the span
      this.context.setDisplay(false, span.parentNode);
      // show the "text input"
      this.context.setDisplay(true, this.comboBox.domNode);
   }
   if (visibility == "REQUIRED") {
      var label = labelDiv.querySelector(".controlLabel");
      domClass.add(label, "BPMRequired");
      this.comboBox.set("required", true);
   } else {
      var label = labelDiv.querySelector(".controlLabel");
      domClass.remove(label, "BPMRequired");
      this.comboBox.set("required", false);
}
```

Note how the code in Example 3-19 on page 108 covers all the states: `DEFAULT`, `EDITABLE`, `REQUIRED`, `READONLY`, `NONE`, and `HIDDEN`.

Note: The code in Example 3-19 on page 108 refers to *utilities*, which come from the `utilities.js` file that is included in IBM BPM, as shown in Figure 3-60. To use the code in `utilities.js`, you must first include it in Included Scripts and then create an AMD alias, as shown in Figure 3-61.

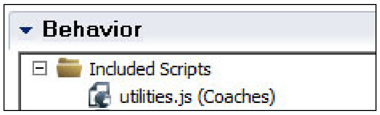
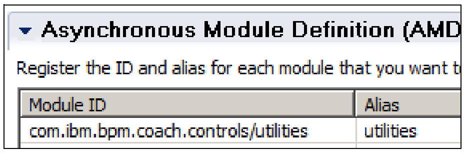


Figure 3-60 utilities.js location



Register the ID and alias for each module that you want t

Module ID	Alias
com.ibm.bpm.coach.controls/utilities	utilities

Figure 3-61 Create an AMD alias for utilities

3.6.3 Providing Coach View information

As with Stock Coach Views, a custom Coach View should supply basic information to make it easier to find and use it.

Tags

Add at least one tag to your custom Coach View, as shown in Figure 3-62.

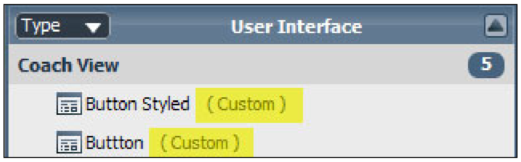


Figure 3-62 Coach View tags

The Coach Editor in Process Designer uses these tags to categorize the Coach View on the palette, as shown in Figure 3-63.

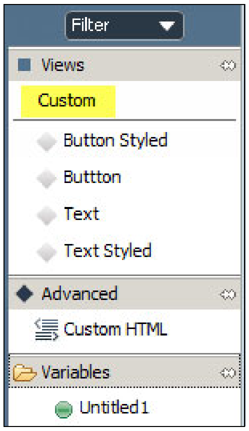


Figure 3-63 Tags in the Coach palette

Coach Editor considerations

Provide meaningful icons for the Coach Editor palette icon and a layout image, as shown in Figure 3-64 on page 111.

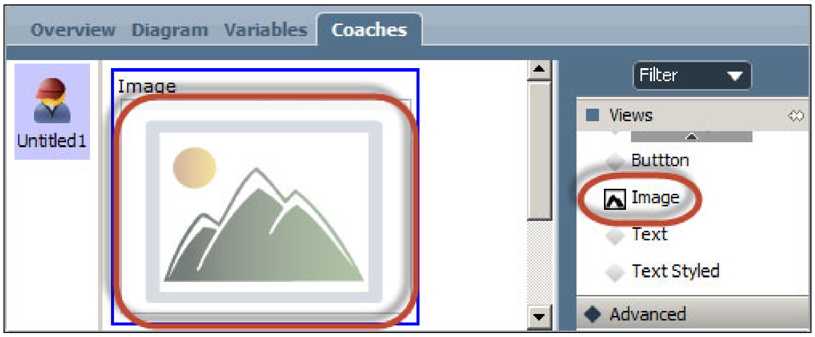
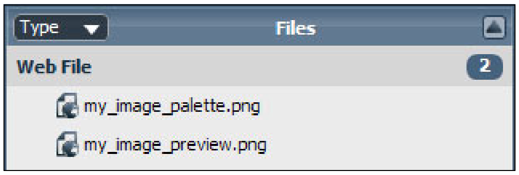


Figure 3-64 Coach View authoring time icons

The icon images should be first imported as web files into your toolkit, as shown in Figure 3-65.



my_image_palette.png
my_image_preview.png

Figure 3-65 Coach View authoring time icons – web files

To set the palette and layout icons in your custom Coach View, click the **Overview** tab. Then, select the icon files in the Preview section, as shown in Figure 3-66.

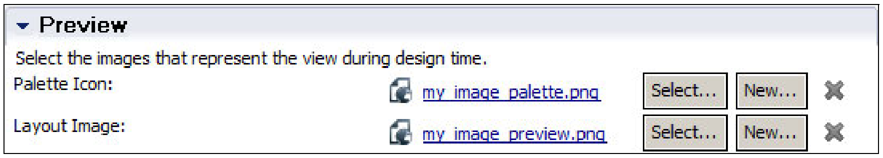

Preview
Select the images that represent the view during design time.
Palette Icon: my_image_palette.png
Layout Image: my_image_preview.png

Figure 3-66 Specifying Coach View authoring time icons

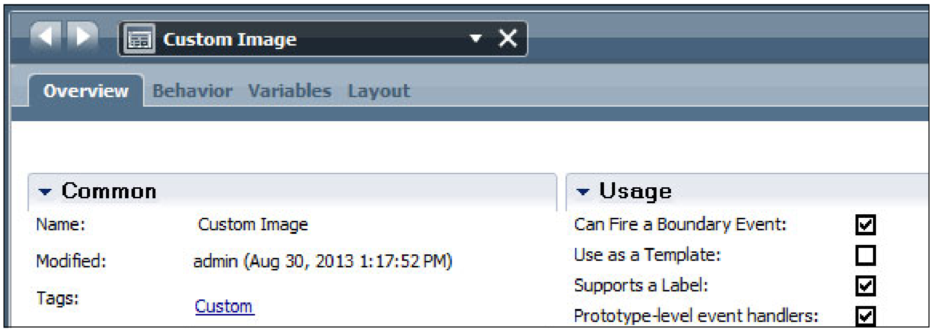
3.6.4 Boundary Events

Boundary Events are client-side browser events that you might want to propagate back to the server. At run time, Boundary Events are implemented as an Ajax call to the server, Boundary Events are recognized in the Human Service editor. At authoring time in the Human Service Editor, you can add wires to Boundary Events to run the client request.

Here are some key uses of Boundary Events:

- ► Navigate to the next step in Human Service flow.
- ► Perform server-side validation.
- ► Get more or new data from the server.

To declare Boundary Events, in the Overview tab in the Coach View Editor, select the **Can Fire Boundary Event** check box, as shown in Figure 3-67.



Common

Name: Custom Image

Modified: admin (Aug 30, 2013 1:17:52 PM)

Tags: Custom

Usage

Can Fire a Boundary Event: ☑

Use as a Template: ☐

Supports a Label: ☑

Prototype-level event handlers: ☑

Figure 3-67 Can Fire a Boundary Event setting

To fire a Boundary Event, add JavaScript code to trigger the event at the appropriate time by using `this.context.trigger(callback, options)`:

```
function()
  {
    // when boundary event completes execute this code
  }
);
```

Note: For more information about the **`trigger()`** function, see the *IBM Business Process Manager V8.5* information center that is found at the following website:

`http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/index.jsp?topic=%2Fcom.ibm.wbpm.wle.editor.doc%2Fdevelop%2Ftopics%2Frcontext.html`

Boundary Events: Performance considerations

Boundary Events have a key advantage when they are used to refresh data from the server. For loopback scenarios where the data is committed and then the same Coach reopens, only the affected data is reloaded in the Coach View instead of the full page. For example, suppose Coach 1 (shown in Figure 3-68) contains a Custom Image Coach View with a Boundary Event that is defined to fire when the user clicks the image. In the Human Service Editor, it is possible to wire the Boundary Event to an Enterprise Content Management Service, which can be used to refresh the image from the server.

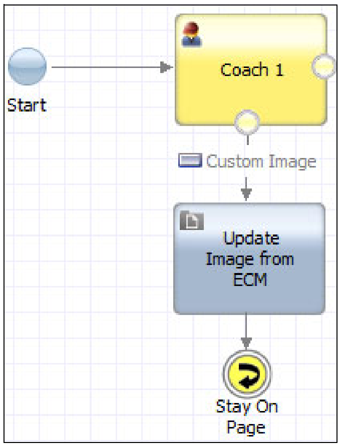
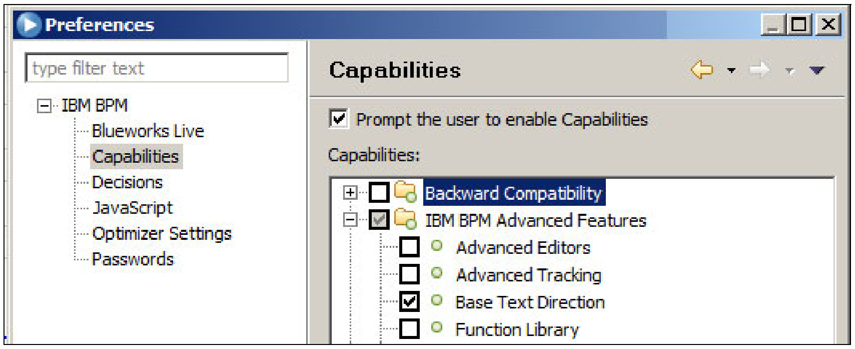


Figure 3-68 Firing a Boundary Event for updates

Boundary Events are useful, but under some circumstances they might negatively impact server performance because the firing of Boundary Events causes a commit of the data from web client to the server. If many concurrent users are interacting with a Coach that is frequently generating Boundary Events, the server might become easily overloaded.

3.6.5 Base Text Direction settings considerations

IBM Business Process Manager can support languages that are written from right to left and languages that are written from left to right. In Process Designer, Base Text Direction support can be enabled by selecting **File** → **Preferences** → **Capabilities** → **Base Text Direction**, as shown in Figure 3-69.


Preferences

type filter text

- IBM BPM
  - Blueworks Live
  - Capabilities
  - Decisions
  - JavaScript
  - Optimizer Settings
  - Passwords

Capabilities

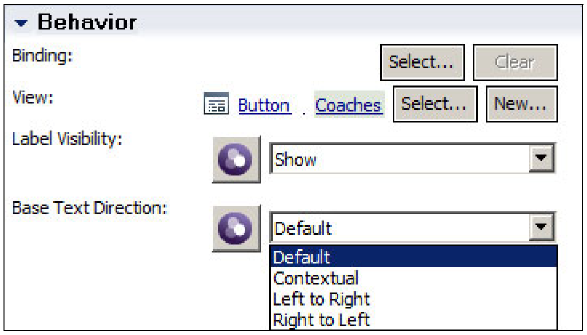
☑ Prompt the user to enable Capabilities

Capabilities:

- ☐ Backward Compatibility
- ☑ IBM BPM Advanced Features
  - ☐ Advanced Editors
  - ☐ Advanced Tracking
  - ☑ Base Text Direction
  - ☐ Function Library

Figure 3-69 Enabling Base Text Direction settings

After Base Text Direction is enabled, the Base Text Direction settings display in the General tab in the Behavior window, as shown in Figure 3-70.


Behavior

Binding: Select... Clear

View: Button Coaches Select... New...

Label Visibility: Show

Base Text Direction: Default

- Default
- Contextual
- Left to Right
- Right to Left

Figure 3-70 Setting the Base Text Direction settings

Table 3-1 on page 115 describes Base Text Direction settings.

Table 3-1 Base Text Direction settings descriptions

Value	Description
Default	Inherit the text direction that is set in the user's profile.
Contextual	Display the text according to the first strong directional character in the string. For example, if the first strong directional character is from a right to left language, display the text from right to left. This setting applies to all text elements that a Coach View shows. For example, a Text stock control has an Arabic label but its contents are English. In this case, the text in the label is right to left and the text in the field is left to right.
Left to Right	Display the text from left to right no matter what characters are in the text.
Right to Left	Display the text from right to left no matter what characters are in the text.

Supporting Base Text Direction

All Stock Coach Views provide Base Text Direction support. To enable Base Text Support in your custom Coach View, you must retrieve the value of the Base Text Direction attribute and write code (typically in load() handler) that looks similar to Example 3-20.

Example 3-20 Base Text Direction

```
var baseTextDirection =
this.context.options._metadata.baseTextDirection &&
this.context.options._metadata.baseTextDirection.get("value") ?
this.context.options._metadata.baseTextDirection.get("value") :
utilities.BiDi.BASE_TEXT_DIRECTION._default;
var textDir = baseTextDirection.toLowerCase() ==
utilities.BiDi.BASE_TEXT_DIRECTION._default ?
         generalPrefTextDirection ?
utilities.BiDi.GENERAL_PREFERENCE_BASE_TEXT_DIRECTION.getValue(generalP
refTextDirection) :
         "" :
      baseTextDirection.toLowerCase()==
      utilities.BiDi.BASE_TEXT_DIRECTION.contextual ?
         "auto" :
         baseTextDirection.toLowerCase();

if (textDir == "ltr" || textDir == "rtl" || textDir == "auto") {
   args.textDir = textDir;
}
```

```
   if (this.context.binding != undefined &&
   this.context.binding.get("value") != undefined) {
   spanViewElement.innerHTML =

   this.context.htmlEscape(this.context.binding.get("value")) :
      "&nbsp;";
utilities.BiDi.applyTextDir(spanDivElement, spanDivElement.innerHTML,
baseTextDirection, generalPrefTextDirection);
this.comboBox.set('value', this.context.binding.get("value"));
}
if (this.context.options._metadata.label != undefined &&
this.context.options._metadata.label.get("value") != "") {
   labelViewElement.innerHTML =
   this.context.htmlEscape(this.context.options._metadata.label.get("va
   lue"));
   utilities.BiDi.applyTextDir(labelViewElement,
   labelViewElement.innerHTML, baseTextDirection,
   generalPrefTextDirection);
}
```

3.6.6 Collaboration

You can use IBM Process Portal to request help from experts and collaborate with experts and other users in real time to complete work on a task. The collaboration feature provides feedback behavior when multiple people work on the same Coach at the same time.

Defining collaboration behavior

In most cases, Coach Views can rely on a Coach Framework to handle Coach Collaboration in IBM Process Portal.

However, there are a few exceptions, and they are usually associated with a container type of custom Coach Views (for example, Table or Tab). Typically, these container types of Coach Views show and hide content one at a time (for example, table pagination or tab transition); some additional logic is required to ensure that the UI shows consistent content during a collaboration session.

Enabling collaboration for container type Coach Views involves the APIs that are shown in Table 3-2 on page 117.

Table 3-2 Collaboration APIs

Collaboration API	Description
`context.triggerCollaboration(payload)`	The Coach View can use this API to emit custom collaboration events to the collaborators (for example, tab transition).
collaboration(evt)	The Coach View should implement this lifecycle API to handle custom collaboration events.

The Stock Table or Tab Coach Views can serve as samples. Example 3-21 shows the collaboration() handler implementation of the Stock Tab Coach View.

Example 3-21 Collaboration() handler implementation

```
if (event.type == 'custom') {
  //do view specific stuff with event.payload, the following       is
what Tab view may do:
  if (!!event.payload && event.payload != "") {
      var panes = this.tabContainer.getChildren();
      for (var i=0; i<panes.length; i++) {
  if (panes[i].id == event.payload) {
    this.tabContainer.selectChild(event.payload);
                break;
         }
      }
   }
} else {
  //call the super class view to handle the collab event
  CoachView.prototype.collaboration.apply(this, arguments);
}
```

3.6.7 Validation

IBM BPM Coach Framework provides a server-side validation capability that you can use to validate the data that is in the Coach before the flow proceeds to the next step in the service flow.

To validate the data in a Coach before the flow proceeds to the next step in the service flow, you can add a validation node, as shown in Figure 3-71. When a Boundary Event is fired, the validation node runs first, and if a validation error is reported, a validation service for the control goes back to the Coach.

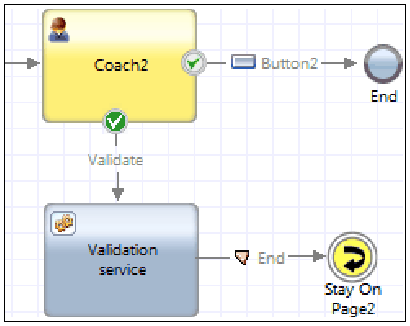


Figure 3-71 Server-side validation in Human Service

Enabling server-side validation

If your custom Coach View receives user input, you must take special steps to enable the server–side validation that is provided by the Coach Framework.

The validate() handler is invoked when the validation service that is associated with the validate Boundary Event returns errors. The handler contains the logic to display error indicators and the error messages that are defined in the validation service. To ensure that your custom Coach View is consistent in the way it displays validation errors, use the Stock Coach Views for examples of presentation logic.

As shown in Figure 3-72 on page 119, a Stock Coach View with a validation error changes color and displays an error icon. When you hover hour cursor over the icon, an error message is shown.

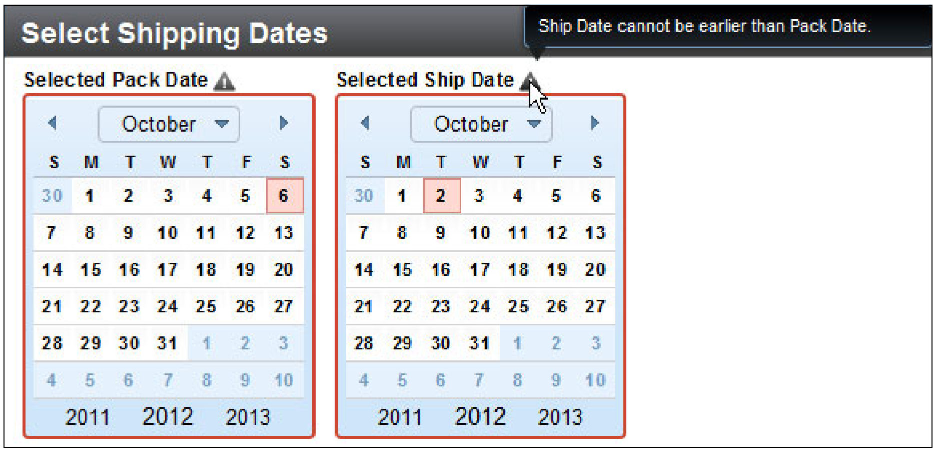


Figure 3-72 Server-side validation - displaying validation errors

The stock Text can serve as an example of how to implement the validate() handler, as shown in Example 3-22.

Example 3-22 How to implement the validate() handler

```
var img = this.context.element.querySelector("img");
var span =
this.context.element.querySelector("span.coachValidationText");
domClass.remove(this.context.element, "coachValidationContainer");
domClass.add(img, "CoachView_hidden");
span.style.visibility = "hidden";
if (this.validateTooltip != undefined) {
  this.validateTooltip.destroyRecursive();
  delete this.validateTooltip;
}
if (event.type == "error") {
  for (var i=0; i<event.errors.length; i++) {
    var error = event.errors[i];
      if (error.view_dom_ids[0] == this.context.element.id) {
      domClass.add(this.context.element, "coachValidationContainer");
      domClass.remove(img, "CoachView_hidden");
      span.style.visibility = "";
      thisClass.add(this.context.element, "coachValidationContainer");
      domClass.remove(img, "CoachView_hidden");
      span.style.visibility = "";
      this.validateTooltip = new Tooltip({
      connectId: [img, span],
         label: error.message,
         position: ["above", "below", "before", "after"]
```

```
        });
        this.validateTooltip.onShow = function(){
        domClass.add(Tooltip._masterTT.domNode,
        "coachValidationTooltip");
        };
      }
    }
}
```

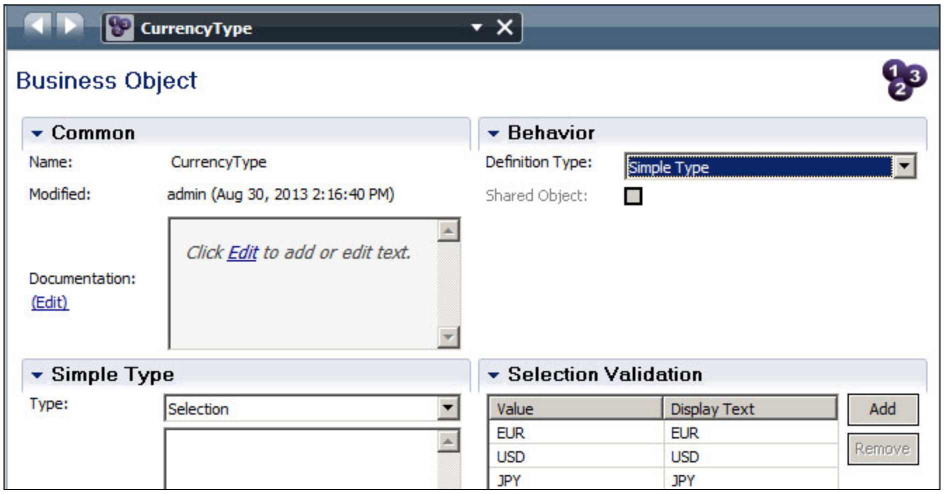
3.6.8 Coach View configuration options

Optionally, Coach View configuration options can be defined to provide users with ways to customize Coach View instances.

Defining a configuration data type

Configuration options can be a set of specific values. For example, your custom Coach View's configuration option might require a currency symbol from a predefined set of supported currency symbols.

In such cases, do not use the built-in types, such as String. Instead, create a Business Object that is called *CurrencyType*, and for Definition Type use Simple Type, as shown in Figure 3-73.

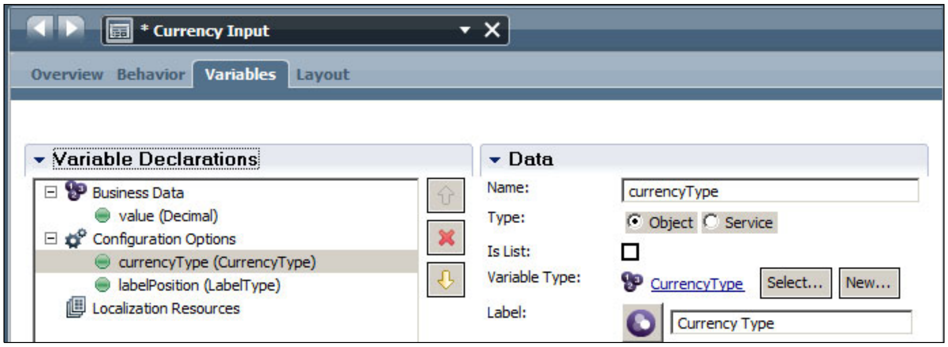


Name: CurrencyType
Modified: admin (Aug 30, 2013 2:16:40 PM)
Documentation: (Edit)
Click Edit to add or edit text.
Definition Type: Simple Type
Shared Object:
Type: Selection

Value	Display Text
EUR	EUR
USD	USD
JPY	JPY

Figure 3-73 Defining a Simple Type

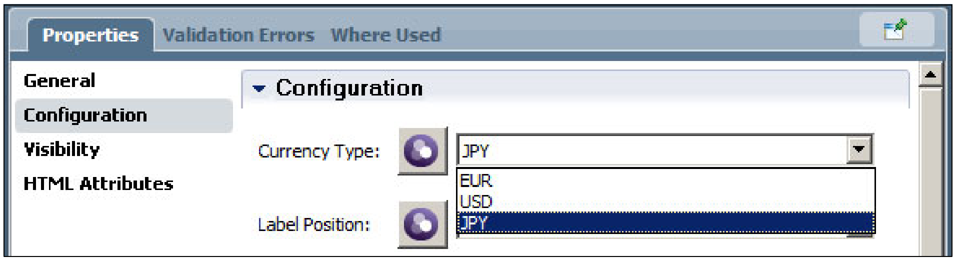
Then, in your custom Coach View, select the CurrencyType for the type of your configuration option, as shown in Figure 3-74 on page 121.



Name: currencyType
Type: [✓ Object] Service
Is List: ☐
Variable Type: CurrencyType
Label: Currency Type

Figure 3-74   Using Simple Types to define Coach View configuration options

When you finish configuring your custom Coach View, the user sees only the valid currency types, as shown in Figure 3-75.



Currency Type: JPY
EUR
USD
JPY

Figure 3-75   Configuration options selection when Simple Types are used

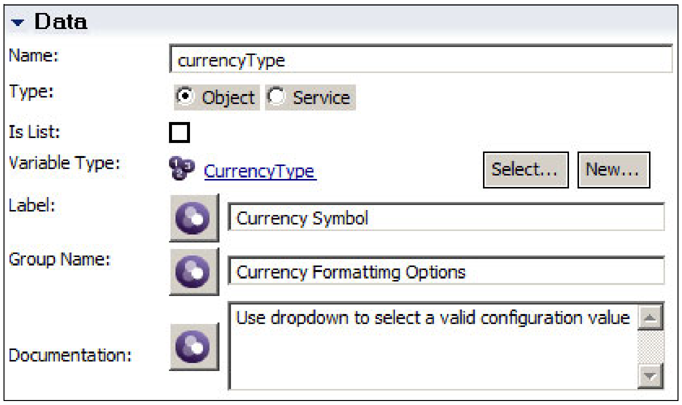
This setting lets you avoid runtime errors and the need to check for a valid currency symbol during run time.

Making configuration options more usable

To make configuration options more usable, complete the following steps

1. In the Label field, provide a meaningful display name for the configuration property.
2. Use the Documentation field to provide hover help text to help users understand how to use the settings.

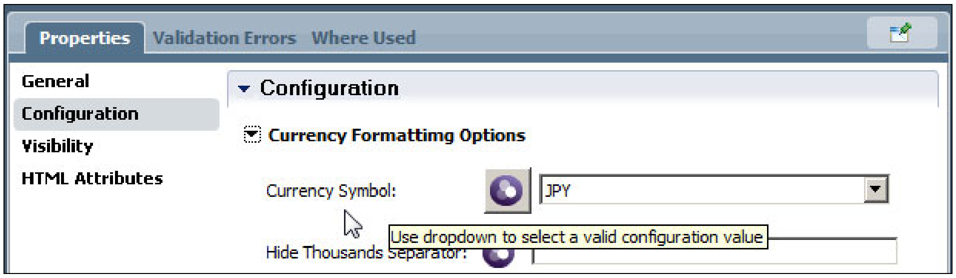
3. Provide a Group Name to group several related configuration options, as shown in Figure 3-76.



Name: currencyType
Type: [✓ Object] Service
Is List: ☐
Variable Type: CurrencyType
Label: Currency Symbol
Group Name: Currency Formattimg Options
Documentation: Use dropdown to select a valid configuration value

Figure 3-76 Configuration options - defining documentation

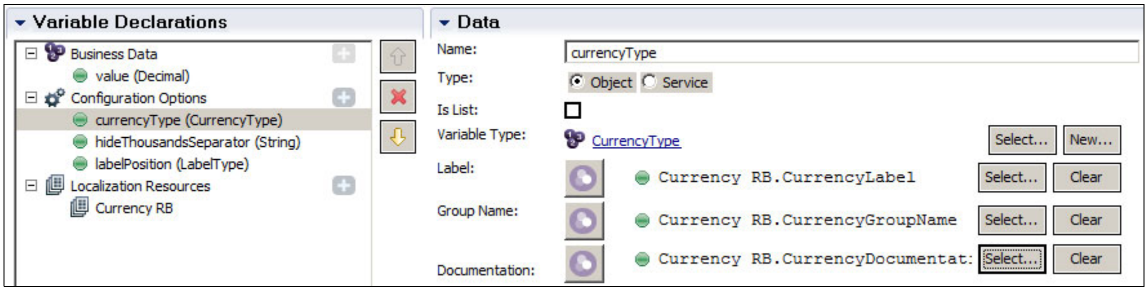
Note the Group Name (Currency Formatting Options) and Documentation (*Use dropdown to select a valid configuration value*) in Coach Editor, as shown in Figure 3-77.



Currency Symbol: JPY
Use dropdown to select a valid configuration value
Hide Thousands Separator:

Figure 3-77 Configuration options - documentation at authoring time

To ensure that your custom Coach Views can be used internationally, make sure to provide translations for the text of your Coach View. These translations can be provided by creating Localization Resources and then by adding these resources to the Localization Resources folder on the Variables page in a custom Coach View (Currency RB in Figure 3-78 on page 123).


Variable Declarations

- Business Data
  - value (Decimal)
- Configuration Options
  - currencyType (CurrencyType)
  - hideThousandsSeparator (String)
  - labelPosition (LabelType)
- Localization Resources
  - Currency RB

Data

Name:	currencyType
Type:	[✓ Object] Service
Is List:	☐
Variable Type:	CurrencyType
Label:	Currency RB.CurrencyLabel
Group Name:	Currency RB.CurrencyGroupName
Documentation:	Currency RB.CurrencyDocumentat:

Figure 3-78 Coach View globalization

Notes: For more information about creating Localization Resources, see the IBM Business Process Manager V8.5 information center that is found at the following website:

http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/topic/com.ibm.wbpm.wle.editor.doc/buildcoach/topics/example_localizingacoach.html?resultof=%22%4c%6f%63%61%6c%69%7a%61%74%69%6f%6e%22%20%22%6c%6f%63%61%6c%22%20

Promoting your configuration options

You can promote your custom Coach View's configuration options. A promoted configuration option appears in higher-level Coach Views, which vastly simplifies and enables Coach View reuse.

Suppose in your custom Coach View (Parent) that you need to reuse another Coach View (Child). If you want the Child Coach View configuration options to be exposed to the Coaches that include it (or other Coach Vies that must include your Parent Coach View), you must promote the Child's Coach Views configuration options.

To accomplish this task, you must explicitly click **Expose this configuration in the parent Coach View** button in your custom Coach View, as shown in Figure 3-79

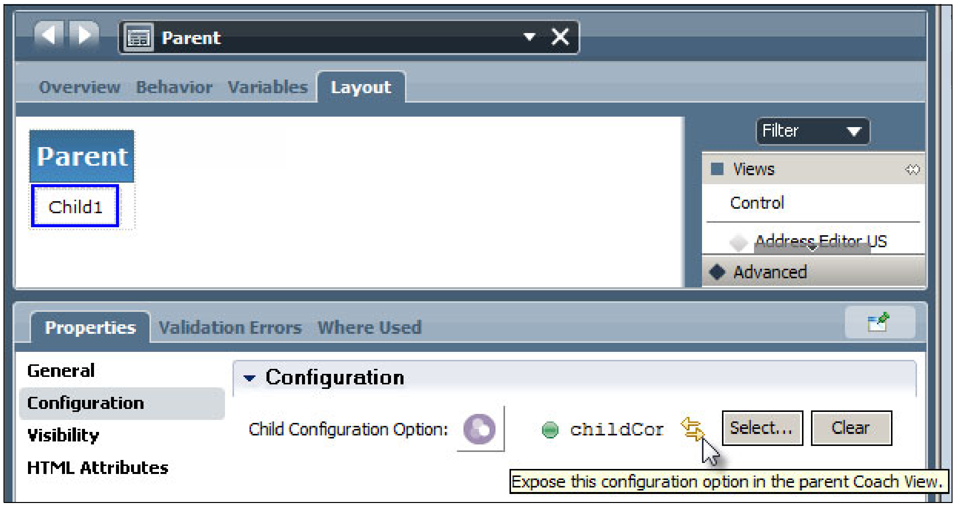


Figure 3-79 Promoting configuration options

Clicking this button creates a variable in your custom Coach View and binds it to the configuration option. Additionally, the user of your custom Coach View (Grandparent) can see and set the configuration options of the child Coach Views you include in your custom Coach View (Parent), as shown in Figure 3-80.

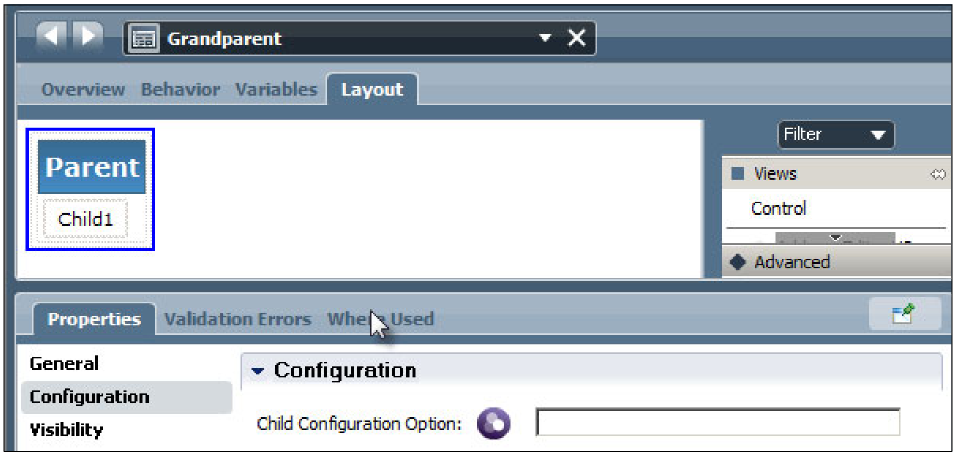


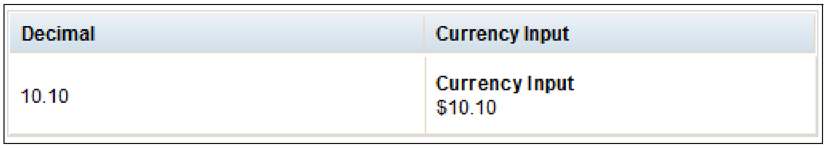
Figure 3-80 Promoted configuration option

3.6.9 Label support

If your Custom Coach View has a label, you must add support for it to ensure that your Coach View behaves as other Stock Coach Views do during run time (for example, when the custom Coach View is included inside the Table and Tab Coach View), and at authoring time in the Coach Editor.

Runtime label support

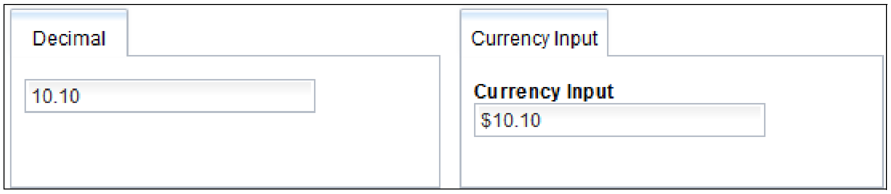
When a custom Coach View is included in the Table or Tab Coach View, the label should appear only once. In Figure 3-81, the Decimal Stock Coach View on the left shows the label only in the table header, and a poorly constructed Custom Coach View on the right shows the label inside the table cell.



Decimal	Currency Input
10.10	Currency Input $10.10

Figure 3-81 Runtime label support in Table Coach View

The same incorrect behavior is shown in the Tab Coach View, as shown in Figure 3-82.



Decimal

10.10

Currency Input

Currency Input
$10.10

Figure 3-82 Runtime label support in Tab Coach View

To enable label “hiding” in container type of Coach Views, such as Table and Tab in the view() handler of a custom Coach View, you must add code that is similar to the code that is shown in Example 3-23.

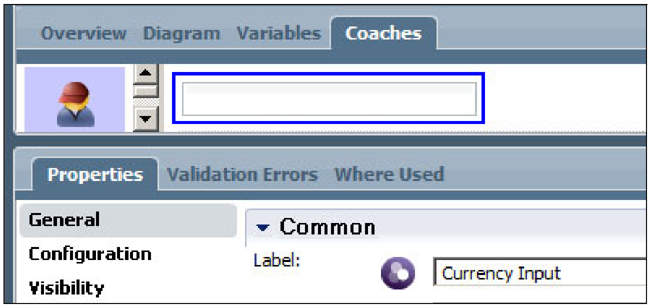
Example 3-23 Label hiding

```
var labelDiv = this.context.element.querySelector(".outputTextLabel");
if (this.context.options._metadata.label == undefined ||
    this.context.options._metadata.label.get("value") == "" ||
    (this.context.options._metadata.labelVisibility != undefined &&
     this.context.options._metadata.labelVisibility.get("value") ==
"NONE")) {
  // hide the label div
  this.context.setDisplay(false, labelDiv);
} else {
  // show the label div
  this.context.setDisplay(true, labelDiv);
}
var visibility = handleVisibility(this.context);
```

The code in Example 3-23 on page 125 uses the `labelVisibility` property to determine whether the label should be shown or hidden.

Authoring label support

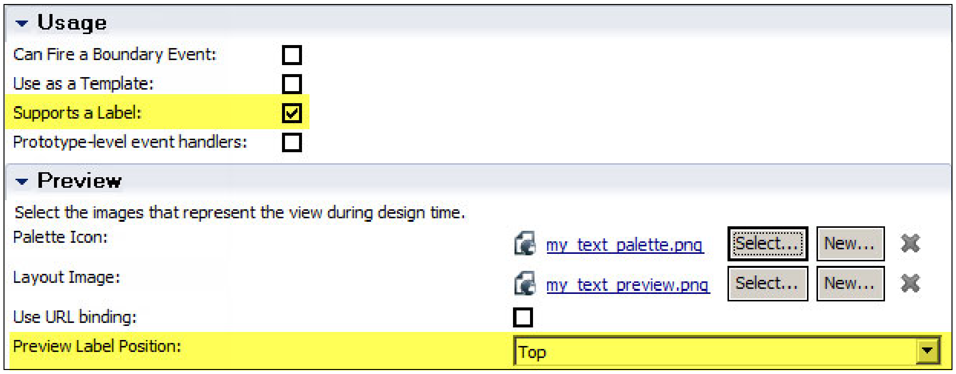
If your Custom Coach View has a label, the label does not appear in the Coach Editor or Coach View Editor by default. Even though the label value is defined, it does not appear in the Coach View Editor, as shown in Figure 3-83.



Label: Currency Input

Figure 3-83 Authoring time label support

For the label to be visible at authoring time, you must select the **Support a Label** check box in the Overview tab of the Coach View Editor. Additionally, you should specify the default label position by using the Preview Label Position drop-down menu, as shown in Figure 3-84.



Can Fire a Boundary Event: ☐
Use as a Template: ☐
Supports a Label: ☑
Prototype-level event handlers: ☐

Select the images that represent the view during design time.
Palette Icon: my_text_palette.png
Layout Image: my_text_preview.png
Use URL binding: ☐
Preview Label Position: Top

Figure 3-84 Configuring runtime label support

This setting specifies only the authoring time preview support. If you have a configuration option to adjust the label position, the design time authoring does not reflect the setting; it should show the default. If you do not supply a configuration option for your label position and you have a hardcoded runtime label position, make sure that the preview label support matches the runtime support.

3.6.10 Coach Views runtime performance considerations

This section describes runtime performance considerations when you are authoring custom Coach Views.

Excessive change events

Avoid unnecessary change event firing. Always check the value of configuration / binding before setting a new value. If a new value is the same as the old value, do not set the value again.

Tab Coach View

Tabs are all loaded up front (not lazy-loaded). Avoid putting too many tabs with many Coach Views in a single Coach, as it takes much time to render all the content even though only one tab is visible up front.

If you cannot avoid many tabs with many Coach Views in each tab, implement a custom Tab Coach View that can handle lazy-load of the Dojo dijit controls.

Lazy-load for the Tab Coach View: If you are using IBM BPM V8.0.1.1, V8.0.2.1, or V8.5.0.1, you can download and install APAR JR47845. This APAR provides lazy-load support for the Tab Coach View. Only the selected tab has all the Dojo dijit controls instantiated. If you are using a IBM BPM version higher than Version 8.5, this lazy-loading feature is included in your version.

Table Coach View

Avoid showing many rows and columns when you use the Table Coach View. The pagination support in Table Coach View is only client-side. So, all data is still loaded on to the client, but only a portion of it is shown to the user.

If you cannot avoid many rows with many Coach Views in each row, implement a custom Table Coach View that can handle lazy-instantiation of the Dojo dijit controls.

Lazy load for the Table Coach View: If you are using IBM BPM V8.0.1.1, V8.0.2.1, or V8.5.0.1, you can download and install APAR JR47171. This APAR provides lazy-load support for the Table Coach View. Initially, data in the table is rendered as HTML. Only when you click a row are the Dojo dijit controls instantiated, assuming the user intends to edit the data. If you are using a IBM BPM version higher than Version 8.5, this lazy-loading feature is included in your version.

Excessive Ajax calls

Avoid using too many Ajax calls during the initial load to populate data. Browsers have a limited number of concurrent connections available and their logic to handle concurrent connections varies.

Instead of using a data initialization Ajax call to the server, use a setup script step to preinstall the data before the Coach step, as shown in Figure 3-85.

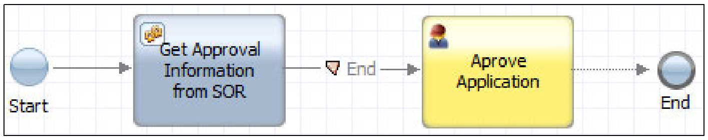
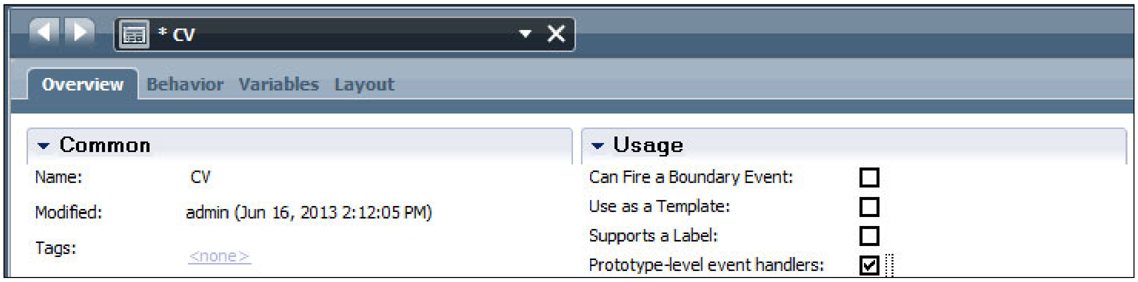


Figure 3-85 Initializing Coach data in a Human Service

Reducing the Coach View memory footprint

Select the **Prototype-level event handlers** check box (available in Version 8.5 or higher) to reduce memory footprint and improve performance when you implement frequently reusable Coach Views, especially if multiple instances of the Coach Views are being used in a single Coach.

Selecting this option means that the event handlers for the Coach View are in the prototype and not in every instance, as shown in Figure 3-86.



Name: CV
Modified: admin (Jun 16, 2013 2:12:05 PM)
Tags: <none>

Can Fire a Boundary Event: ☐
Use as a Template: ☐
Supports a Label: ☐
Prototype-level event handlers: ☑

Figure 3-86 Enabling the Prototype Event Handler

However, the JavaScript code that you use to create and access variables differs between Coach View instance-level event handlers and prototype-level handlers, as shown in Table 3-3.

Table 3-3 Prototype-level event handler programming considerations

Instance-level event handlers	Prototype-level event handlers
Define the variable in the inline JavaScript of the Coach: `View:var myVariable = "123";`	Define the variable in the inline JavaScript of the Coach: `View:this.myVariable = "123";`
Access the variable in the load event: `handler:if(myvariable == "123") { ... }`	Access the variable in the load event: `handler:if(this.myvariable == "123") { ... }`

3.6.11 Coach Views development considerations

This section describes development and coding tips when you are authoring custom Coach Views.

Debugging

Always test your Coach / Human Services by clicking the orange Run Service button (followed by using native web browser debug tools, such as Firebug), and not the Debug Service button, as shown in Figure 3-87.

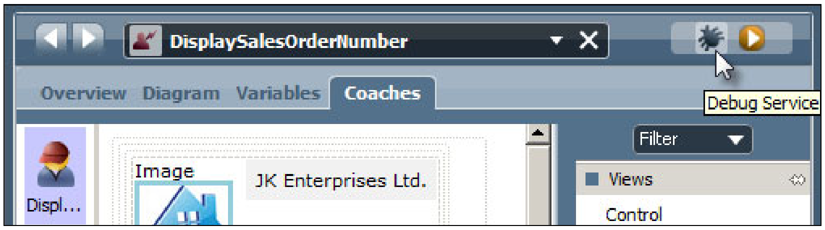


Figure 3-87 Debug Service

When you use Debug Service, some artifact-loading side effects might occur, which might hide certain coding errors. This situation occurs because a “step” in the Debug Service debugger shows and loads the variable page in the debugger. As a result, any Boundary Events that use stay-on page return control back to the Coach.

Using bind() and bindAll() APIs correctly

Here are some preferred practices to use bind() and bindAll() APIs correctly:

- Avoid using event handler change() as the callback; instead, use a different function as a callback handler.
- Do *not* use lifecycle event change() as the callback handler, as shown in the following example:

  ```
  code:this.context.binding.get("value").bindAll(this.change);
  ```

  Instead, use a different function as the callback handler, as shown in the following example:

  ```
  this.context.binding.get("value").bindAll(myCallbackFunc);
  ```

- If you use a different function as the callback handler when the change() handler is invoked, it is guaranteed that the change is on either the data binding or one of the configuration options. To further distinguish between these two items in the view() handler, add the following code to check event types:

  ```
  if(event.type == "config") {
  // the change is on one of the configure options
     } else {
  // the change is on data binding
  }
  ```

- Add rebind logic when the base object is changed.

  For example, suppose that a Coach View is bound to the `personInfo.phoneList` variable, and the Coach View is interested in any change (add or delete set) of any items of phoneList. To correctly implement this behavior, add the following logic to the load() handler:

  ```
  this.bindAllHandle =
  this.context.binding.get("value").bindAll(this.callbackFuncForListIt
  emChange);
  ```

  Add this logic in the unload() handler:

  ```
  if(this.bindAllHandle) {
     this.bindAllHandle.unbind();
     this.bindAllHandle = null;
  }
  ```

  The above code works for most scenarios. However, when `this.context.binding` gets a new value, `this.context.binding.get("value")`; returns a different object from the one you called on bindAll(). As a result, `this.callbackFuncForListItemChange` is associated with a stale object.

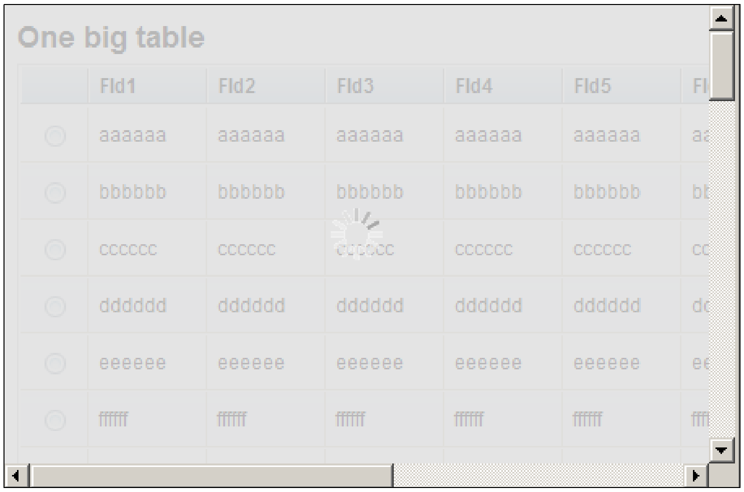
This issue can be resolved if, as described earlier, you do *not* use lifecycle event change() as the callback in the change() handler, but instead use a different function as the callback handler:

```
if(event.type == "config") {
  // the change is on one of the configure options
} else {
  // the change is on data binding, so we need to rebind
  // unbind current handle
  if(this.bindAllHandle) {
     this.bindAllHandle.unbind();
  }
  // rebind
  this.bindAllHandle =
this.context.binding.get("value").bindAll(this.callbackFuncForListIt
emChange);
}
```

Using a Loading Curtain pattern

Depending on the size and the complexity of a web page, JavaScript and CSS might take a long time to complete initialization. The speed of initialization depends on the browser's rendering engine. During this initialization, a web page is not ready for user interactions and might flicker as the page elements are loaded and initialized.

To avoid this flickering, you can implement a Loading Curtain as a Coach View that includes a progress indicator and a CSS style with an opaque background color high z-index, as shown in Figure 3-88.



One big table

	Fld1	Fld2	Fld3	Fld4	Fld5	Fl
○	aaaaaa	aaaaaa	aaaaaa	aaaaaa	aaaaaa	aa
○	bbbbbb	bbbbbb	bbbbbb	bbbbbb	bbbbbb	bb
○	cccccc	cccccc	cccccc	cccccc	cccccc	cc
○	dddddd	dddddd	dddddd	dddddd	dddddd	dd
○	eeeeee	eeeeee	eeeeee	eeeeee	eeeeee	ee
○	ffffff	ffffff	ffffff	ffffff	ffffff	fff

Figure 3-88 Loading Curtain to hide excessive flickering

To implement a Loading Curtain Coach View, complete the following steps:

1. Add Content Box in the Layout tab. This content box must be the outer-most Coach View that contains all other Coach Views in your Coach, as shown in Figure 3-89.

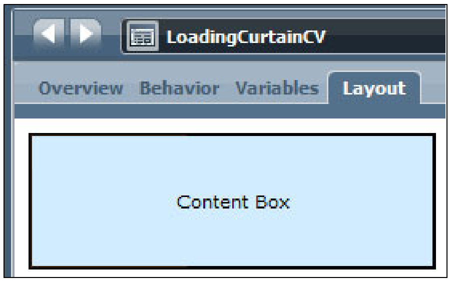
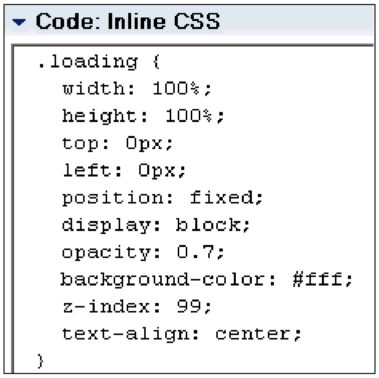


Figure 3-89 Loading Curtain - Content Box Coach View

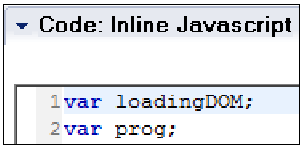
2. Add Inline CSS, as shown in Figure 3-90.



```
.loading {
  width: 100%;
  height: 100%;
  top: 0px;
  left: 0px;
  position: fixed;
  display: block;
  opacity: 0.7;
  background-color: #fff;
  z-index: 99;
  text-align: center;
}
```

Figure 3-90 Loading Curtain - Inline CSS

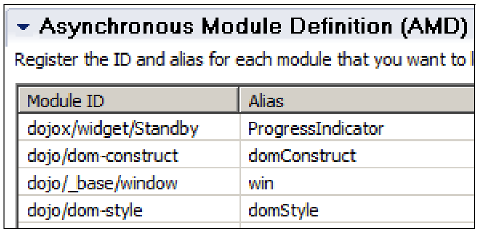
3. Add Inline JavaScript, as shown in Figure 3-91.



```
var loadingDOM;
var prog;
```

Figure 3-91 Loading Curtain - Inline JavaScript

4. Add AMD dependencies, as shown in Figure 3-92 on page 133.

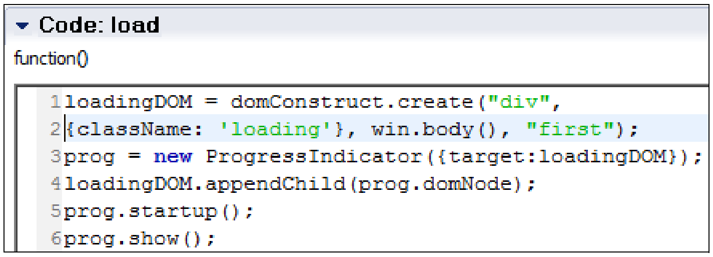


Register the ID and alias for each module that you want to l

Module ID	Alias
dojox/widget/Standby	ProgressIndicator
dojo/dom-construct	domConstruct
dojo/_base/window	win
dojo/dom-style	domStyle

Figure 3-92 Loading Curtain - AMD

5. Implement the load() event handler, as shown in Figure 3-93.

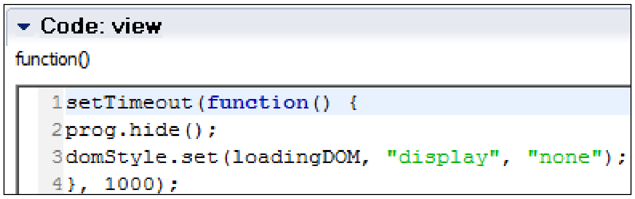


function()

```
loadingDOM = domConstruct.create("div",
{className: 'loading'}, win.body(), "first");
prog = new ProgressIndicator({target:loadingDOM});
loadingDOM.appendChild(prog.domNode);
prog.startup();
prog.show();
```

Figure 3-93 Loading Curtain - load() handler

6. Implement the view() event handler, as shown in Figure 3-94.



function()

```
setTimeout(function() {
prog.hide();
domStyle.set(loadingDOM, "display", "none");
}, 1000);
```

Figure 3-94 Loading Curtain – view() handler

Note: The loading curtain sample is included in the `SampleLoadingCurtain - SN01.twx` file.

Caching binding objects

Here are some preferred practices to cache binding objects:

- Avoid caching binding objects or configuration option objects and use them later, as they might change.
- When you are developing Coach Views, use `this.context.binding` to access the binding object of a Coach View instance, and use `this.context.options` to access the options of a coach view instance.
- For convenience, it is customary to define a local variable as the shortcut to the current value of `this.context.binding` or `this.context.options` as follows:

  ```
  var myBinding = this.context.binding;
  var myBindingValue = this.context.binding.get("value");
  var myOptionX = this.context.options.pptionX;
  var myOptionXValue = this.context.options.optionX.get("value");
  ```

  However, this practice might lead to stale objects. Here are two such examples:

  – The `myBindingValue` variable becomes stale when `this.context.binding` receives a new value, which occurs when `this.context.binding.set("value", newValue)` is called.

  – The `myBinding` variable becomes stale because `this.context.binding` might point to a new binding object. For example, suppose a Coach View is bound to *`personInfo`*`.address`. Later, when the *`personInfo`* variable itself is changed, `this.context.binding` now points to a new binding object.

Support for multiple Coach View instances

In some situations, you might want to use the `id` attribute for your DOM elements within a Coach View. Because Coach Views are reusable, it is common to have several Coach Views that are instantiated in a single Coach. To ensure that your Custom Coach View can be instantiated more than once in a single Coach, your Custom Coach View must have a unique DOM ID. To ensure a unique ID, you can use the *`$$viewDOMID$$`* placeholder keyword. At run time, this keyword is replaced by the Coach View DOM ID.

To generate a unique DOM ID for your Custom Coach View, in the Layout page, create a custom HTML control and add HTML code that is similar to Example 3-24.

Example 3-24   Custom HTML control code

```
<div id="$$viewDOMID$$_myId1">
  <span id="$$viewDOMID$$_myId2"></span>
  <input id="$$viewDOMID$$_myId3" type="button" class="Mybutton"
name=myjbtnName" value="default">
  </input>
</div>
```

Table behavior

Test your Custom Coach View's runtime behavior in the Table Coach View.

One common issue that can occur is when a Coach View inside a table is selected to enter data. Unless correct clipping behavior is implemented, clipping might occur, as shown in Figure 3-95.

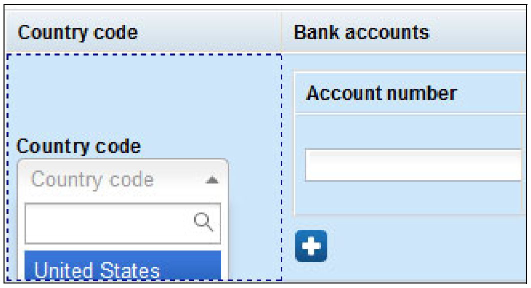


Figure 3-95   Custom Coach View clipping inside the Table Coach View

Notice that a user has no access to all the drop-down list items. The correct behavior is to display the drop-down menu at the top of the table, as shown in Figure 3-96.

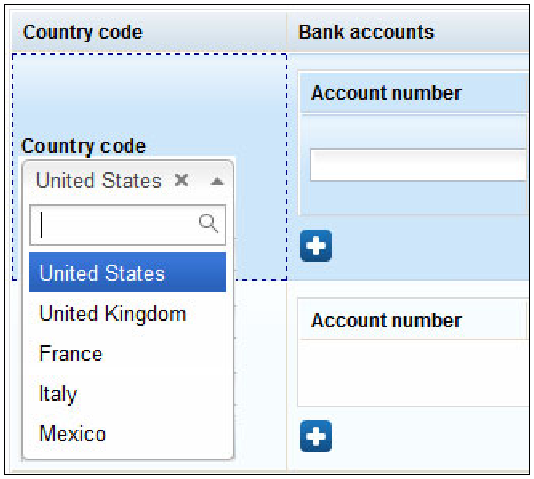


Figure 3-96 Correct Custom Coach View behavior inside the Table Coach View

4

Advanced performance considerations

This chapter presents advanced performance considerations for both designing and developing Coaches, and also runtime deployment and usage guidance. This chapter is a follow-on to the Coach development performance considerations in 2.8, “Performance considerations” on page 52.

4.1 Design and development considerations

This section discusses how to design and develop Coaches that perform well at run time.

4.1.1 Minimize network requests to the server

For optimal performance, make as few network requests as possible to the Process Server. This is particularly true if the production runtime environment will have high network latency between the Coach clients and the Process Server, or the Process Server and its database. This is also important if the production runtime environment will use a relatively low-bandwidth network.

The number of network requests can be determined via Firebug, if Firefox is the browser, or a similar tool for other browsers.

The following techniques minimize network requests:

- Combining multiple JavaScript files into one (or a few)
- Combining multiple CSS files into one
- Utilizing CSS image sprites to reduce the number of requests to retrieve image files
- Designing coarse-grained API calls that minimize the number of API calls that flow over the network to the Process Server.

4.1.2 Execute long-running calls asynchronously

When making service calls that are expected to take a long time to complete, such as calls to back-end systems, perform these calls asynchronously so control can be returned to the user as quickly as possible.

4.1.3 Maximize browser concurrency

Most browsers utilize multiple channels in order to process concurrent network requests between the browser and the Process Server. However, the Coach author must structure the Coach to be able to take full advantage of this capability. Specific techniques are identified in the rest of this section.

Load CSS Files before JavaScript Files

JavaScript files can have dependencies on CSS files, such as layout options. If while processing a JavaScript file such a dependency is found and the CSS file is not yet loaded, processing will be stopped for all JavaScript files on all browser channels. This can produce higher, and more variable, Coach load times.

To avoid this issue, where possible, load CSS files before JavaScript files to ensure all CSS dependencies are satisfied before processing JavaScript files. This is primarily applicable to custom Coach code.

Load CSS and JavaScript files from HTML

Browsers only have visibility to files that are loaded directly from the parent HTML file; these are the only files that can be downloaded concurrently on multiple browser channels. Structure your Coach HTML file accordingly.

As a corollary to the above, avoid loading JavaScript files from other JavaScript files, which results in sequentially loading each JavaScript file on the same browser channel.

4.2 Deployment considerations

This section discusses performance at runtime deployment considerations for Coaches.

4.2.1 Use a high performing browser

Business Process Manager supports many browsers. The specific browser releases supported vary by Business Process Manager release, and include Chrome, Firefox, Internet Explorer, and Safari. Performance can vary dramatically depending on which browser is used. Typically the most recent releases of these browsers offer the best performance. This is particularly true for Internet Explorer.

4.2.2 Enable browser caching

Coaches rely heavily on Ajax and JavaScript code; their performance depends heavily on the browser’s efficiency in processing this code. To improve Coach client response times, you might need to adjust both HTTP server and client browser settings. This section covers ensuring that browser caching is enabled. HTTP server caching is addressed in 4.2.5, “Tune the HTTP server” on page 142.

By default, Business Process Manager allows the browser to cache static information such as style sheets and JavaScript files for 24 hours. However, if the browser is configured in a way that does not cache static files such as these, the same resources are loaded repeatedly, resulting in long page load times.

The next sections describe how to enable browser caching for Internet Explorer and Firefox.

Ensuring browser cache is enabled in Internet Explorer

To enable caching, use the following steps:

1. Click **Tools** → **Internet Options** → **General** → **Browsing History**.
2. Set the cache size, indicated by *Disk space to use*, to at least 50 MB.

Ensure browser cache is not cleared on logout

Modern browsers have options to clear the cache on exit; make sure that this setting is not enabled. Use the following procedure to disable cache clearing:

1. Click **Tools** → **Internet Options**.
2. Under the General tab, make sure that the *Delete browser history on exit* check box is cleared.
3. Under the Advanced tab, ensure that the *Empty Temporary Internet Files folder when browser is closed* check box is cleared. Click **OK** to save the settings.
4. Restart your browser to ensure that the changes have taken effect.

Ensuring browser cache is enabled in Firefox

To enable caching, use the following steps:

1. Click **Tools** → **Options** → **Advanced** → **Network** → **Offline Storage**.
2. Set the *Offline storage* to at least 50 MB.

Use the following procedure to enable cache clearing:

1. Click **Tools** → **Options**.
2. Click the Privacy tab and make sure that it reads *Firefox will: Remember History*. Click **OK**. This setting enables caching. If you select, *Firefox will: Never remember history*, caching is disabled and you must change the setting.

   If you select *Use custom settings for history*, you have additional options that still allow caching:

   – If you select *Automatically start Firefox in a private browsing session*, caching is enabled. However, after the browser is closed, everything is erased (not only cache, but also browsing history) as though you were never using the browser.

   - If private browsing is not selected, make sure the *Clear history when Firefox closes* check box is cleared.
3. Restart the browser to apply the changes.

4.2.3 Optimize network quality between browser clients and servers

As noted previously, multiple network requests are made between the Coach in the browser and the Process Server. Therefore, ensuring an optimal network connection between these systems is crucial. There are two primary factors that influence network speed:

- One factor that influences network latency is the physical distance between servers and clients, and also the distance between the Business Process Manager servers in a cluster and their database (for example, between the Process Server and its databases). When practical, locate Business Process Manager servers physically close to each other, and also to the Coach browser clients to minimize the latency for requests.
- The other key factor is the quality of the network itself: the speed and the bandwidth of the connection. Ensure that the browser client is using the fastest possible network with the highest bandwidth.

4.2.4 Warm-up frequently used Coaches

Coaches are generated in two stages on the Process Server:

- Stage 1 does the vast majority of the generation of the coach runtime code. This stage is only done once during the lifetime of a Process Server Instance.
- Stage 2 only performs substitution of dynamic data in the coach runtime code that was generated during Stage 1. It is done every time a Coach is opened.

This is a good design trade-off, since most of the Coach generation is only done once during a Process Server runtime instance. However, this does mean the first usage of a Coach will take much longer than subsequent uses. To mitigate this effect, where possible, *warm up* commonly used Coaches one time whenever the Process Server is started. To warm up a Coach, open an instance of that Coach in a client browser. The Coach instance does not need to be completed; Stage 1 code generation will be done when the Coach instance is opened.

If warming up a Coach is not practical for business reasons, such as not being able to have a *test process instance* and task in the system to use to warm up the Coach, recognize the first usage of a Coach can take longer than subsequent usages.

4.2.5 Tune the HTTP server

Coaches are deployed to Process Servers in production scenarios that often use a topology that includes an HTTP server or an HTTP server plug-in. Coaches send multiple requests to their respective server, which are then handled by this HTTP server. To minimize the number of requests and the size of the response data to obtain good user response times, tune your HTTP server to efficiently handle the expected load.

In particular, ensure that caching is effectively enabled in the HTTP server. Much of the content that is requested by coaches is static (for example images and JavaScript files), and can be cached in the browser. Of course, ensure that browser caching is enabled first, but also tune the HTTP server to support caching at that level. Specifically in the `httpd.conf` file, use the following caching and compression settings:

- ExpiresActive on
- ExpiresByType image/gif A86400
- ExpiresByType image/jpeg A86400
- ExpiresByType image/bmp A86400
- ExpiresByType image/png A86400
- ExpiresByType application/x-javascript A86400
- ExpiresByType text/css A86400

Tip: 86400 seconds = 1 day

By default, static files (JavaScript, CSS, images, and so on) are set to expire in 24 hours. If increasing the expiration is desirable, one can change the expiration header in the HTTP server. This can have a side effect when new Business Process Manager (BPM) product code is applied which updates any of these files, since the files will *not* be reloaded until the expiration period elapses.

Note: For more information about tuning the IBM HTTP Server, which is included as part of the Business Process Manager product, see the IBM HTTP Server Performance Tuning page:

http://publib.boulder.ibm.com/httpserv/ihsdiag/ihs_performance.html

5

Real world lessons from Apex Process Consultants

In this chapter, we introduce Apex Process Consultants, an IBM Premier Business Partner, and their palette of tools to support the development of IBM Business Process Manager (BPM) Coach Views.

- Apex overview
- Designing Coach Views for a business technical user
- Leveraging client-side data
- Packaging for reuse
- Optimizing performance

5.1 Apex overview

We introduce Apex Process Consultants and take a look at their Coach View tools and offerings.

5.1.1 Apex Process Consultants

Apex Process Consultants is an IBM Premier Business Partner focused 100% on IBM BPM. Our mission is to partner with our customers to build Business Process Engineering capability that delivers business performance through agility. We provide a wide range of business process engineering services built around IBM BPM to meet our customers' needs including:

- BPM Program Services: Apex can help a customer design an enterprise-wide BPM program that can scale delivery capability for repeatable success. This includes helping customers crystallize their BPM vision, defining a BPM strategy aligned to their corporate objectives, and creating a Center of Excellence to build on success of initial projects.
- BPM Enablement Services: The Apex approach is designed to enable customers to take ownership of their BPM efforts by providing knowledge transfer of industry best practices. This includes ensuring teams are up to speed quickly with tailored BPM training, providing dedicated or on-demand mentoring services to help teams continue to grow, and creating powerful tools and methodologies to accelerate BPM implementations.
- BPM Delivery Services: Incubator projects build business consensus and provide context for targeting and estimating a first implementation in a specific business process area. Apex can provide expert staffing and experienced leadership as needed to complement the customer's resources. Apex can also manage a customer's BPM project end-to-end, taking total project delivery responsibility for the BPM solution.

In addition to services, Apex develops and markets a range of products that complements the capabilities of IBM BPM and helps our customers achieve their BPM objectives.

5.1.2 Evolution of Apex user interface tools

User experience is critical to the success of any IBM BPM project. From the beginning, Apex has used patterns and toolkits to help enhance the look and feel of IBM BPM Coaches. The earliest toolkits were JavaScript libraries that would insert Dojo grids and dialogs into IBM BPM 7.x Coaches using blocks of custom

HTML. Although effective, these were restricted to specific use cases and the modeling usability was not what we wanted it to be.

In late 2010, we collaborated with a customer that required highly interactive, client-side user interfaces using IBM BPM 7.x. To meet these requirements, we built a new client-side, Dojo-based Coach transform from the ground up. During the next year and several production releases of the process application, we discovered and refined a long list of UI patterns and implementation approaches. While the user experience was outstanding, the heritage Coach designer in 7.x limited the modeling experience.

In late 2011, we shared our UI experiences with the IBM BPM product team and they introduced us to the BPM 8 Coach View framework that was under development then. The new framework addressed the limitations that we experienced in 7.x and allowed us to fully realize the potential for the UI patterns that we had developed.

In early 2012, we joined the beta program for BPM 8.0 and started development of Apex Coach Views. The framework delivered on the promise and we quickly had our first prototype running. We were honored when John Reynolds invited us to co-present the new IBM BPM Coach View Framework at Impact 2012. John showed the framework and we demonstrated an industrial example based on our prototype.

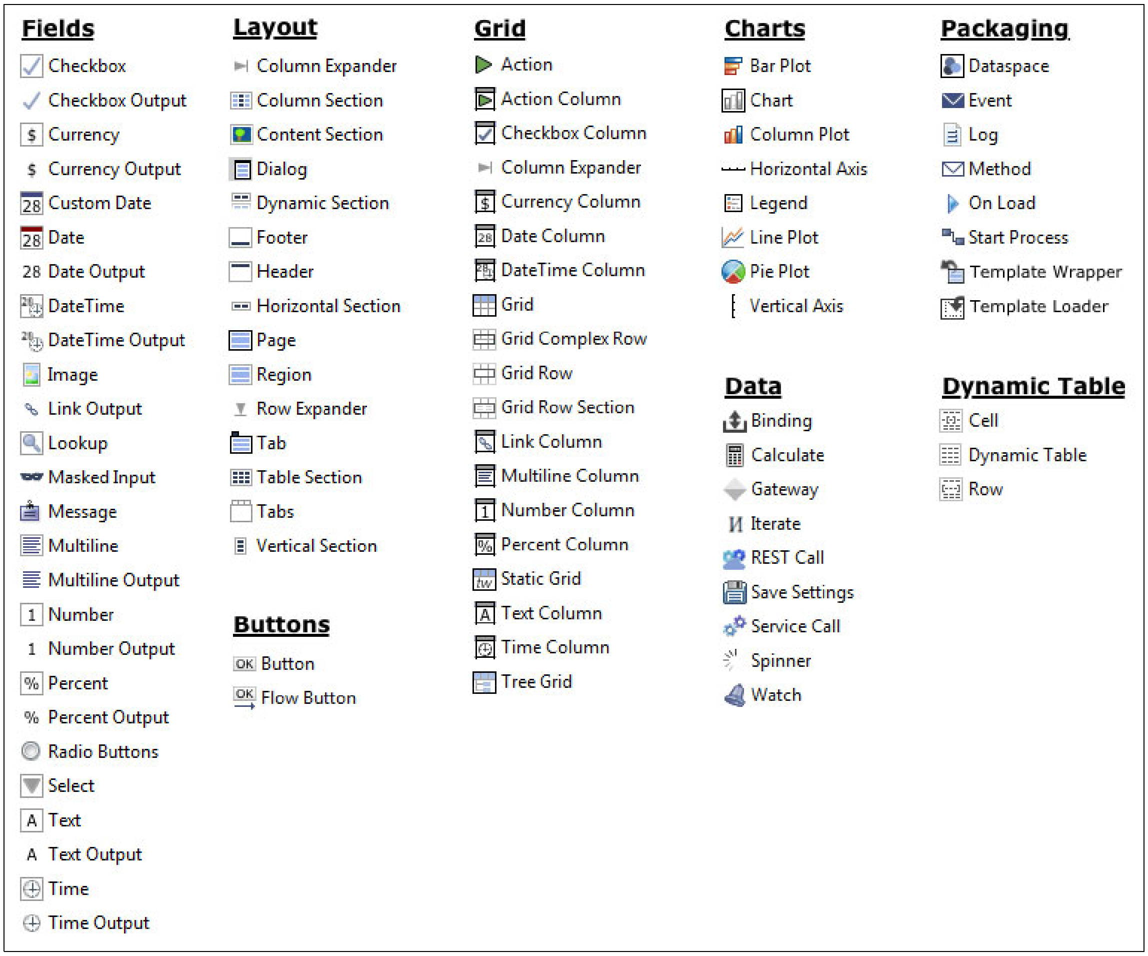
Development continued through the summer culminating in the release of Apex Coach Views 1.0 in the fall of 2012. Five additional releases were completed by the end of 2013, with most of the features driven by our customers' continuing evolution of advanced UI patterns through real industrial use.

5.1.3 Apex Coach Views

Apex Coach Views is a library of 90 ready-to-use layouts and controls designed for business technical users. They enable rich user experience with the latest dynamic client-side technologies under the hood and are ideal for both task completion and ad hoc information management user interfaces. Apex Coach Views provides the following benefits:

- Improved usability
- Reduced development cost
- Improved performance
- Provides a more consistent user experience

The complete set of Apex Coach Views is shown in Figure 5-1 on page 146.



Fields
- Checkbox
- Checkbox Output
- Currency
- Currency Output
- Custom Date
- Date
- Date Output
- DateTime
- DateTime Output
- Image
- Link Output
- Lookup
- Masked Input
- Message
- Multiline
- Multiline Output
- Number
- Number Output
- Percent
- Percent Output
- Radio Buttons
- Select
- Text
- Text Output
- Time
- Time Output

Layout
- Column Expander
- Column Section
- Content Section
- Dialog
- Dynamic Section
- Footer
- Header
- Horizontal Section
- Page
- Region
- Row Expander
- Tab
- Table Section
- Tabs
- Vertical Section

Buttons
- Button
- Flow Button

Grid
- Action
- Action Column
- Checkbox Column
- Column Expander
- Currency Column
- Date Column
- DateTime Column
- Grid
- Grid Complex Row
- Grid Row
- Grid Row Section
- Link Column
- Multiline Column
- Number Column
- Percent Column
- Static Grid
- Text Column
- Time Column
- Tree Grid

Charts
- Bar Plot
- Chart
- Column Plot
- Horizontal Axis
- Legend
- Line Plot
- Pie Plot
- Vertical Axis

Data
- Binding
- Calculate
- Gateway
- Iterate
- REST Call
- Save Settings
- Service Call
- Spinner
- Watch

Packaging
- Dataspace
- Event
- Log
- Method
- On Load
- Start Process
- Template Wrapper
- Template Loader

Dynamic Table
- Cell
- Dynamic Table
- Row

Figure 5-1 Apex Coach Views

The remainder of this chapter provides insight into the design of Apex Coach Views and the patterns that can be implemented using them.

5.2 Designing Coach Views for a business technical user

This section shares experiences in designing Coach Views for business technical users. It includes the definition of a business technical user, definition of three levels of user interface design work, and an explanation of design patterns and decisions that help design Coach Views for a business technical user.

5.2.1 Business technical user defined

IBM BPM was designed with the business technical user in mind. Coach views should be designed in the same way. For the design of Apex Coach Views, we define a *business technical user* as a person with domain knowledge who is comfortable building complex spreadsheets and possibly simple databases using Microsoft Office tools. The business technical user is not a professional software developer but is comfortable working with technical tools.

5.2.2 User interface design skill levels defined

At Apex, we divide user interface development work into three skill levels:

- Level 1: Building Coaches by dragging and dropping pre-existing atomic and composite Coach Views.
- Level 2: Building composite Coach Views by dragging and dropping pre-existing atomic and composite Coach Views.
- Level 3: Building atomic Coach Views using HTML, JavaScript, and CSS.

Apex Coach Views enable business technical users to become adept at level 1 and 2 modeling.

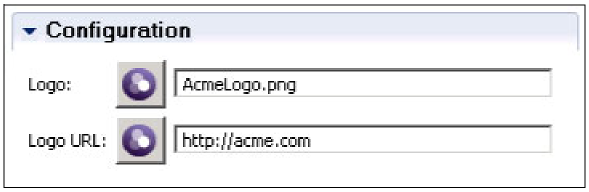
Level 3 modeling requires deep web development skills and is usually beyond the capabilities of a business technical user. Apex Coach Views are designed to reduce or even eliminate the need for level 3 skills. The following sections explore key design patterns and decisions using examples from Apex Coach Views.

5.2.3 Consolidate common patterns

Coach design efficiency can be improved by consolidating common patterns into new atomic Coach Views. Apex Coach Views includes layout Coach Views that control the way information is presented on the page. Layout views include page-related views, container views, and section views. Page-related views include:

- Page provides the main container for the page and establishes the liquid layout
- Header displays a logo and text at the top of the page
- Footer displays text at the bottom of the page

A header is an example of consolidating a common pattern. The same result could have been accomplished with combinations of other Coach Views (Horizontal Section, Image, and Text Output in this example). Providing a dedicated Coach View-like header makes modeling more efficient (single view versus multiple views) and improves ease of use through simple configuration options (see Figure 5-2).



Logo: AcmeLogo.png

Logo URL: http://acme.com

Figure 5-2 Simple configuration options on Header Coach View

5.2.4 Specialize by behavior

A key design decision is whether to make a single general-purpose Coach View or a set of Coach Views that are specialized by behavior. To illustrate this decision, look at an example from Apex Coach Views. Container views are used to subdivide the page container into smaller liquid layout containers or containers with special behavior. Container views include:

- Tabs define a set of containers that are presented as tabs
- Tab defines a container within the Tabs set of containers
- Region subdivides a container into subcontainers
- Dialog defines a container that is presented in a modal dialog

A tab is an example of specializing by behavior. The library could have been designed with Vertical Section to be used as a tab. Table 5-1 compares the Vertical Section and Tab Coach Views.

Table 5-1 Comparing Vertical Section and Tab

	Vertical Section	**Tab**
Configuration Options	Region Show Label onValidationStateChange	onShow onHide Load On onValidationStateChange
Properties[a]	none	isLoaded isSelected

	Vertical Section	Tab
Methods[b]	disable enable hide show reset setLabel	disable enable hide show reset setLabel select

a. Properties are part of the public interface for an Apex Coach View and are accessible on the client using expressions like “views.myTab.isSelected”.
b. Methods are part of the public interface for an Apex Coach View and are accessible on the client using expressions like “views.myTab.select()”.

While most of the methods are the same for the views, there are two behavioral differences that justify the specialized Tab view:

- A tab can be shown by the user clicking it or using the select() method
- The load of the contents of a tab can be deferred to improve page load performance

Neither of these behaviors is applicable to a Vertical Section. Using a Vertical Section as a tab would have made it more complex and less intuitive to use.

5.2.5 Clarify layout using helper views

Helper views are Coach Views that modify the presentation or behavior of another Coach View. To illustrate the concept of helper views, look at an example from Apex Coach Views. Section views organize a set of fields into a row, column, or table inside of a container. Section views include:

- A Vertical Section organizes the fields into a vertical column with the labels and fields aligned.
- A Horizontal Section organizes the fields into a horizontal row.
- A Dynamic Section is a repeating Horizontal Section based on a static or progressively loaded list.
- A Table Section organizes the fields within its Column Sections into a tabular view.
- A Column Section organizes a set of fields into a column within a Table Section.
- A Column Expander makes a field to span two or more columns in a Table Section.
- A Row Expander makes a field to span two or more rows in a Table Section.

A Column Expander and a Row Expander are examples of helper views that clarify the layout. Consider the example in Figure 5-3. The street address field spans three columns to provide more space to enter the value.

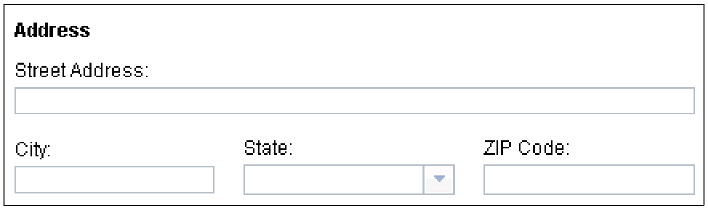


Figure 5-3   Street address spanning three columns

This behavior could have been achieved by adding a Column Span configuration option to each of the field views. This is not ideal because it would make the field views more complex and would also make the layout confusing (see Figure 5-4). The State and ZIP code fields appear as if they would be on the same line as the Street Address.

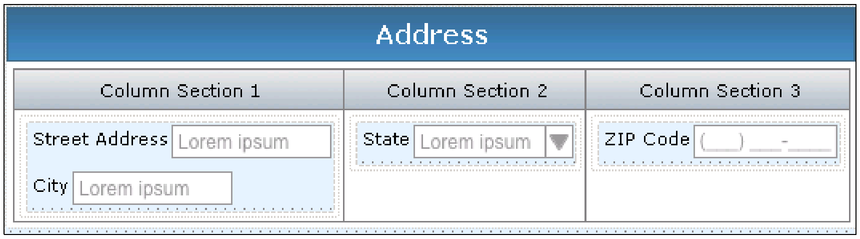


Figure 5-4   Address table section without Column Expanders

Apex Coach Views include the Column Expander helper view to make the Street Address field span three columns as shown in Figure 5-5.

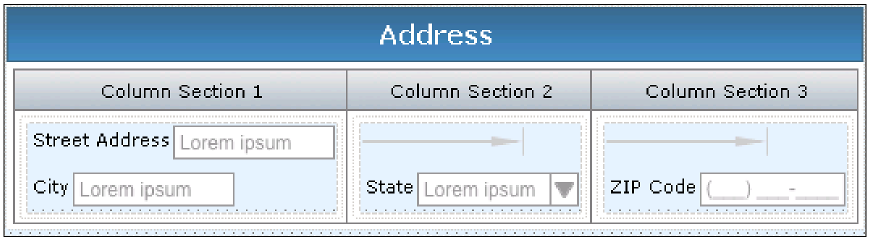


Figure 5-5   Address table section with Column Expanders

The Column Expanders do not have any configuration options. They are simply dragged from the palette onto the canvas.

5.2.6 Specialize by data type

Specializing Coach Views by data type improves usability. The Field views in Apex Coach Views illustrate specialization by data type. Field views are used to present a value with an optional label on a Coach. Field views include:

- Text values are presented using Text, Multiline, and Masked Input fields.
- Numeric values are presented using Number, Percent, and Currency fields.
- Date and time values are presented using Date, Time, DateTime, and Custom Date fields.
- Discrete values are presented using Checkbox, Radio Buttons, Select, and Lookup fields.

Define common configuration options

When designing a set of field views, work from the general to the specific. Start by establishing a set of configuration options that are common to all data types. The field views in Apex Coach Views share these configuration options:

- Dataspace Binding defines the client-side binding for the field (see 5.3, “Leveraging client-side data” on page 156 for more details)
- Format:
  - Show Label defines the position of the label (left, top, none)

Define configuration options by presentation

Next, specialize the configuration options by presentation. Apex Coach Views include three presentations: input, output, and grid column. The output presentation does not have any additional configuration options beyond the common set. The input presentation adds the following configuration options:

- Format:
  - Placeholder Text defines text (for example, Enter part number) that is shown when the field has no value.
- Validation:
  - Validation Message defines the message that will be displayed if the field is not valid.
  - Validation Service defines an optional Ajax service for server-side field (and cross-field) validation.
- Inline Edit:
  - Inline Edit turns inline edit (click to edit) on or off.
  - Update Service defines the Ajax service that will persist the changes.

  - Identifier defines the ID that will be passed to the update service.
  - Attribute defines the attribute name that will be passed to the update service.
- onChange defines a JavaScript snippet that will be executed when the field value changes.

The column presentation adds the following configuration options:

- Field defines the parameter or expression based on parameters that will give the column its value.
- Width defines the optional width of the column.
- Selectable defines whether the column value is selectable or not.

Define configuration options by data type

Next, define additional configuration options for each data type.

Configuration options for text fields

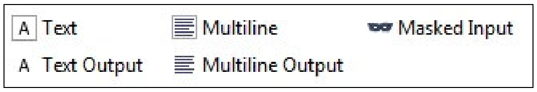


Figure 5-6 Text fields in Apex Coach Views

The text fields in Apex Coach Views as shown in Figure 5-6 have configuration options as follows:

- Text Coach View presents a single line of text and includes these additional configuration options:
  - *Is Password* defines whether the field should be masked for password input.
  - *Validation* defines an optional regular expression that will be used to validate field values.
- Multiline Coach View presents multiple lines of text and includes these additional configuration options:
  - Number of Rows option defines the vertical size of the input area.
  - Rich Text option defines whether the field will include rich text formatting.
  - Plug-ins option defines an optional list of rich text field plug-ins that will be included.
  - Validation option defines an optional regular expression that will be used to validate field values.

- Masked Input Coach View presents a single line of text using a format mask and includes these additional configuration options:
  - Mask defines a pattern of characters for the mask ("(200) 200-000)" for example).
  - Mask Definition defines the meaning of each mask character ("2" means 2-9 and "0" means 0-9 for example).

Configuration options for numeric fields

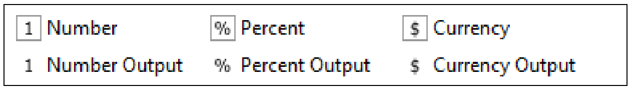


Figure 5-7 Numeric fields in Apex Coach Views

The numeric fields in Apex Coach Views as shown in Figure 5-7 have configuration options as follows:

- Number Coach View presents a numeric value and adds these additional configuration options:
  - Decimal Places option defines the maximum number of decimal places allowed.
  - Use 1000's Separator option indicates whether the separator (defined by the locale) will be included.
  - Negative Numbers option defines how negative numbers are presented (parentheses or minus sign).
- Percent Coach View option presents a percentage value and includes this additional configuration option:

  Decimal Places option defines the maximum number of decimal places allowed.
- Currency Coach View presents a currency value and adds these additional configuration options:
  - Fractional option indicates whether fractional values will be included or not.
  - Currency option is a drop-down selection of one of the ISO 4217 currency codes.
- Number, Percent, and Currency each include these additional validation configuration options:
  - Minimum defines the minimum value.
  - Maximum defines the maximum value.

Configuration options for date and time fields

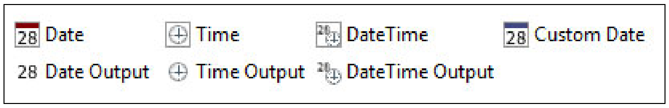


Figure 5-8 Date and time fields in Apex Coach Views

The date and time fields in Apex Coach Views as shown in Figure 5-8 have configuration options as follows:

- Date Coach View presents a date value and adds one additional configuration option:

  Format defines the presentation of the date as short, medium, long, or full.

- Custom Date Coach View allows entry of dates using different formats and adds one additional configuration option:

  Formats option defines a comma-separated list of date formats that can be used.

- Time Coach View presents a time value and adds these additional configuration options:
  - Format defines the presentation of the time as short, medium, or long.
  - Increment defines the granularity of the drop-down values (every 15 minutes for example).
  - Range defines how many values will be shown in the drop-down (4 hours for example).

- DateTime Coach View presents both a date and a time value and adds these additional configuration options:
  - Format defines the presentation of the date and time as short, medium, long, or full.
  - Increment defines the granularity of the drop-down values (every 15 minutes for example).
  - Range defines how many values will be shown in the drop-down (4 hours for example).

- Date, Custom Date, Time, and DateTime all include these additional validation configuration options:
  - Minimum defines the minimum value allowed.
  - Maximum defines the maximum value allowed.

Configuration options for discrete fields

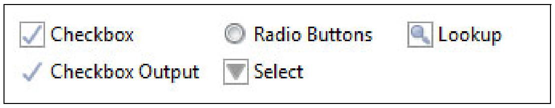


Figure 5-9 Discrete fields in Apex Coach Views

The discrete fields in Apex Coach Views as shown in Figure 5-9 have configuration options as follows:

- Checkbox does not add anything beyond the common fields.
- Radio Buttons Coach View presents a set of mutually exclusive options and adds these additional configuration options:
  - Orientation option defines whether the options are shown horizontally or vertically.
  - Name Value Pairs (List) option defines the options using a data entry table.
  - Name Value Pairs (JSON) option define the options using a JSON string.
- Select Coach View presents a set of mutually exclusive options as a drop-down and adds these additional configuration options:
  - Name Value Pairs (List) option defines the options using a data entry table.
  - Dataspace List Source defines the options using a client-side array.
- Lookup Coach View presents a set of mutually exclusive options as a drop-down and adds these additional configuration options:
  - List Service option specifies the Ajax service that returns the filtered list of name value pairs.
  - Data Source option specifies an Apex node that will return the filtered list of name value pairs.
  - Filters option specifies the comma-separated list of data space values for the filters.
  - Display Attribute option identifies the attribute to be displayed if it is not “name”.

5.3 Leveraging client-side data

The IBM BPM Coach View framework includes support for binding fields to values defined in the Human Service. This section describes a set of patterns that can be used to leverage client-side data.

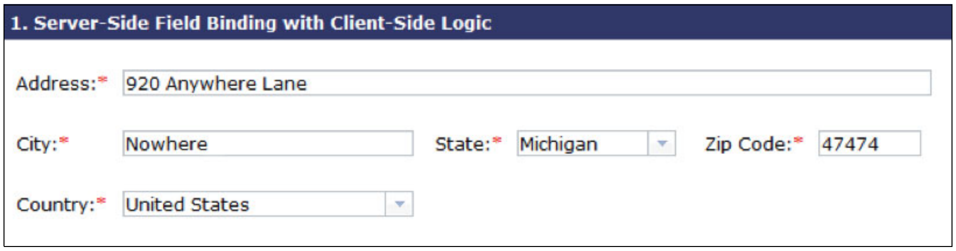
- Simplify the Human Service by reducing or eliminating Boundary Events for the interaction between Coach Views.
- Reduce the server-side session size by retrieving data dynamically on the client.
- Improve responsiveness by progressive loading of large data sets.

5.3.1 Combining server-side and client-side data

This section introduces the concept of client-side data through three patterns that combine server-side and client-side data.

Using client-side binding for cross-field interactions

Coach designs frequently require interaction between fields on a Coach. Cross-field interaction is dramatically simplified by using client-side data. Consider the example of an address section as shown in Figure 5-10.


1. Server-Side Field Binding with Client-Side Logic

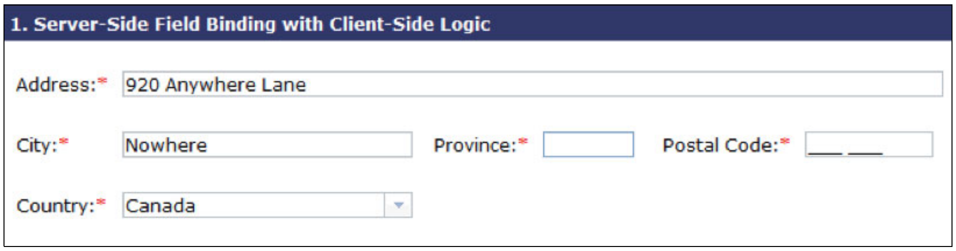
Address:* 920 Anywhere Lane

City:* Nowhere State:* Michigan Zip Code:* 47474

Country:* United States

Figure 5-10 Address section for the United States

When the user changes the country from USA to Canada, the State field is replaced by a Province Field and the Zip Code field is replaced by a Postal Code field as shown in Figure 5-11 on page 157.


1. Server-Side Field Binding with Client-Side Logic

Address:* 920 Anywhere Lane

City:* Nowhere Province:* Postal Code:* ___ ___

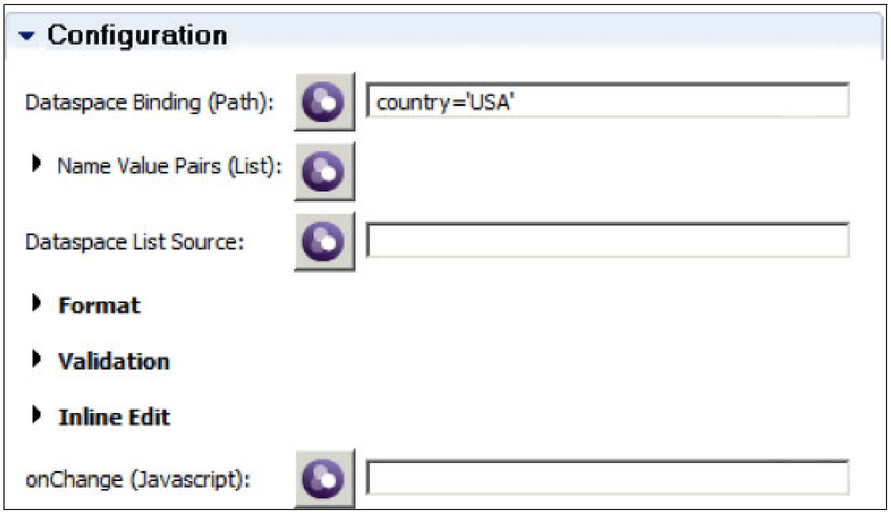
Country:* Canada

Figure 5-11 Address section for Canada

In versions before 8.5, this could be achieved using a Boundary Event and a server script that sets the field visibility using JavaScript, but that would clutter the Human Service diagram and require a round trip to the server. IBM BPM 8.5 offers rule-based visibility, but the functionality stops at visibility. Client-side binding offers a simpler, more complete approach.

Add client-side binding to the Country field

Like the other fields, the Country is bound to a property of the address object: “address.country” in this case. Fields in Apex Coach Views can also be bound to a client-side variable as shown in Figure 5-12.


Configuration

Dataspace Binding (Path): country='USA'

Name Value Pairs (List):

Dataspace List Source:

Format

Validation

Inline Edit

onChange (Javascript):

Figure 5-12 Dataspace binding for the Country field

A *data space* in Apex Coach Views is a pool of client-side data that is independent from the `tw.local` variables that come from the server. The data space binding configuration option defines the name that will be used to access the value in the page’s data space.

Add client-side logic

Now that the value is addressable on the client, logic needs to be added to react to changes in that value. Apex Coach Views includes the Gateway Coach View to define client-side logic. The Gateway Coach View provides the same capabilities as a gateway in a process diagram. Figure 5-13 shows the Coach Views that are used to present the Zip Code and Postal Code.

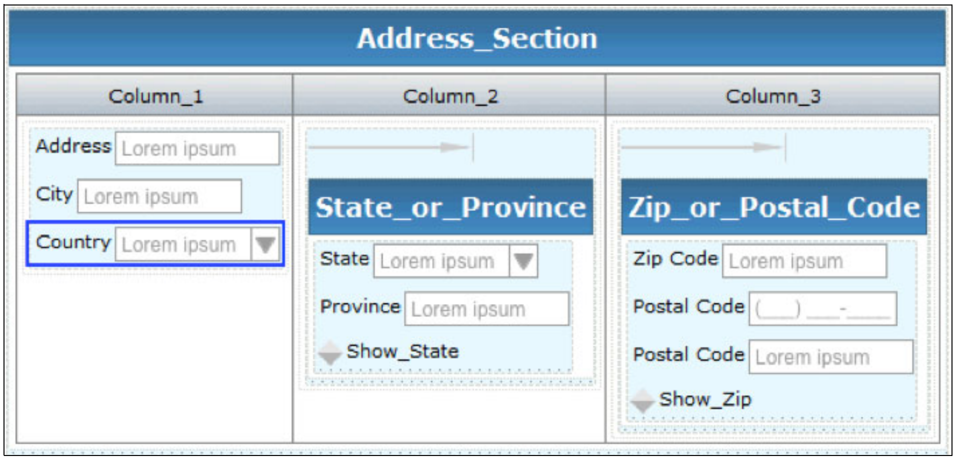


Figure 5-13 Coach views for Zip Code and Postal Codes

Table 5-2 shows the configuration options of the Show_Zip gateway.

Table 5-2 Show_Zip configuration options

Option	**Value**	**Comments**
Operands	Country	This gateway monitors the country client-side value.
Evaluate On	Load and Change	This gateway is evaluated both on load of the page and as the country changes.
Condition 1	value1 == 'USA'	When the value is USA.
Action 1	views.Zip_Code.show().enable(); views.Postal_Code_Canada.hide ().disable(); views.Postal_Code_Other.hide(). disable();	The Zip Code field is shown/enabled and the others are hidden/disabled.
Condition 2	value1 == 'Canada'	When the value is Canada.

Option	Value	Comments
Action 2	views.Zip_Code.hide().disable(); views.Postal_Code_Canada.show().enable(); views.Postal_Code_Other.hide().disable();	The Canadian Postal Code field is shown/enabled and the others are hidden/disabled.
Default Action	views.Zip_Code.hide().disable(); views.Postal_Code_Canada.hide().disable(); views.Postal_Code_Other.show().enable();	Otherwise, the other Postal Code field is shown/enabled and the others are hidden/disabled.

The actions shown in Table 5-2 on page 158 are JavaScript snippets that are executed on the client. “views.Zip_Code” is a reference to an Apex node that is created for each Apex Coach View. “.show()” and “.enable()” are examples of helper functions that are included in Apex Coach Views to simplify client-side logic.

Retrieving data on the client

A more powerful example of client-side data is retrieving whole Business Objects on the client. Consider the example of a Human Service to prepare an order. The business wants to show some of the store details at the top of the Prepare Order Coach. Without client-side binding, a service would be called before the Coach to read the store from the database. This has several potential issues:

- As more Business Objects are added, the service diagram and variables become more complex.
- The Business Objects are read before the Coach is displayed thus slowing the load of the Coach.
- The Business Objects are held in memory on the server consuming more server resources.
- The Business Objects held on the server can become stale if other users are modifying them at the same time.

Apex Coach Views addresses all these issues by making it easy to read Business Objects from the client.

Manage only business keys in the Human Service

In this example, storeId is an input to the service and orderId is an output. No private variables are required because they will be managed on the client side as shown in Figure 5-14 on page 160.

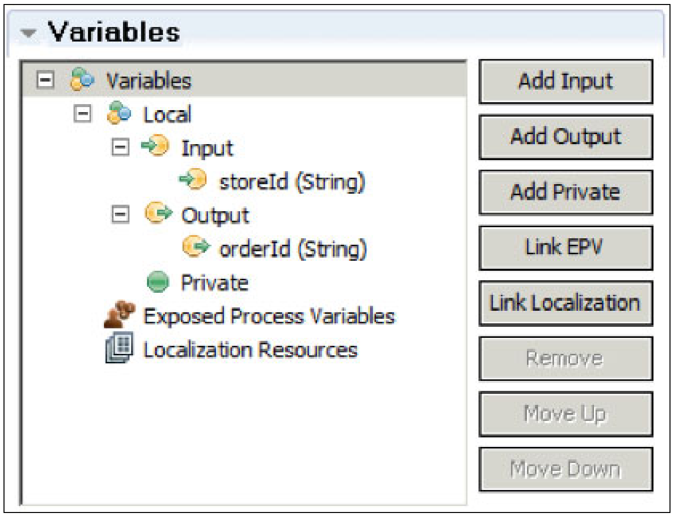


Figure 5-14 Only business keys in the Human Service

Get the storeId to the client

In this example, storeId is a GUID that is not shown to the user. Apex Coach Views includes the Binding Coach View to handle cases like this. It is a non-visual control that binds a value on the server side to a value on the client side.

Read the store on the client

Apex Coach Views includes a Service Call Coach View to call an Ajax service. The configuration options are shown in Table 5-3.

Table 5-3 Service call configuration options

Configuration Option	Example Value	Comments
Service	Read Store Ajax	The Ajax service that will be called
Execute On	On Demand or On Load	When the service should be called
onExecuteBegin	views.Spinner.show()	JavaScript snippet to execute before the call starts
onExecuteComplete	views.Spinner.hide()	JavaScript snippet to execute after the call completes
onError	views.Errror_Dialog.show()	JavaScript snippet to execute if there is an error

Notice that there are no configuration options for inputs and outputs. The Service Call Coach View inspects the inputs to the Ajax service and pulls them from the client-side data space. When the service call is complete, the outputs are put into the data space. This *auto mapping* of inputs and outputs makes using Ajax services to retrieve data quick and easy. In this example, the Read_Store Call Service Coach View is set to read the store on load. It pulls the storeId from the data space, makes the call, and puts the store output into the data space.

Bind the fields on the client

The fields on the Coach are bound to the store object like this:

- Name field data space binding is “store/name”
- Number field data space binding is “store/number”

When the page loads, the fields are shown and as soon as the Ajax call completes, the fields are automatically updated with the values that came from the service.

5.3.2 Managing large data sets

Large tables of information are very common both in task completion user interfaces, dashboards, and administrative user interfaces. The server-side approach to displaying a large grid is to fetch the rows on the server before the Coach and display them in a repeating table. This has several drawbacks:

- The records are fetched before the Coach, which increases response time.
- The records are held in memory on the server, which increases server memory requirements.
- Coach views are created for each cell in the table resulting in poor client performance.

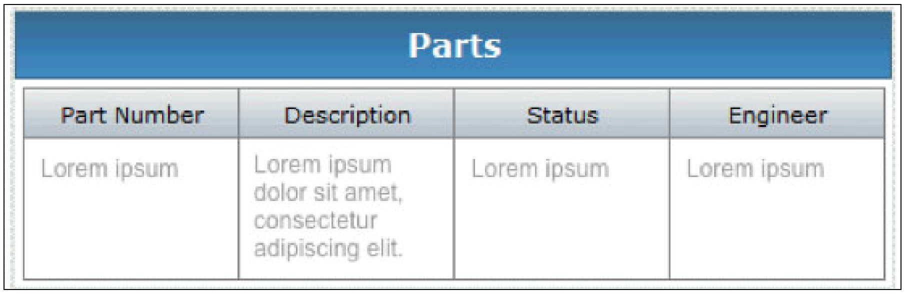
Apex Coach Views address these issues with the Grid Coach View and its associated Column views.

Design the grid

Start by creating the basic grid:

- Drag a Grid view onto the canvas, set the label to “Parts” and leave the configuration options set to defaults.
- Drag a Text Column into the grid, set the label to “Part Number” and the field configuration option to “name”.
- Drag a Multiline Column into the grid, set the label to “Description” and the field configuration option to “Description”.

- Drag a Text Column into the grid, set the label to “Status” and the field configuration option to “status”.
- Drag a Text Column into the grid, set the label to “Engineer” and the field configuration option to “engineer” as shown in Figure 5-15.

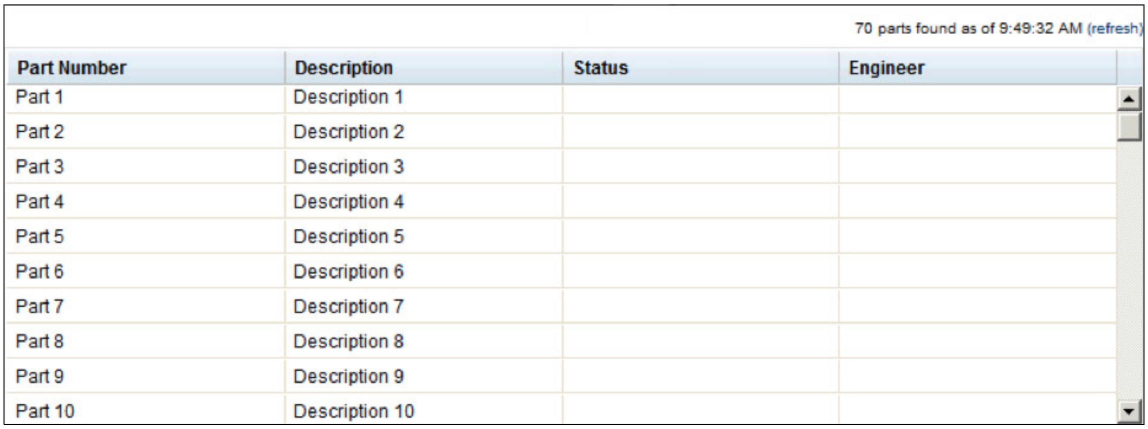


Parts

Part Number	Description	Status	Engineer
Lorem ipsum	Lorem ipsum dolor sit amet, consectetur adipiscing elit.	Lorem ipsum	Lorem ipsum

Figure 5-15 Completed grid design

The grid has a default Human Service that returns fake data with name and description as shown in Figure 5-16. This allows you to test the visual design before the underlying Ajax service is available.



70 parts found as of 9:49:32 AM (refresh)

Part Number	Description	Status	Engineer
Part 1	Description 1		
Part 2	Description 2		
Part 3	Description 3		
Part 4	Description 4		
Part 5	Description 5		
Part 6	Description 6		
Part 7	Description 7		
Part 8	Description 8		
Part 9	Description 9		
Part 10	Description 10		

Figure 5-16 Resulting grid with fake data

This UI-first approach helps solidify the design before investments are made in back-end services.

Build the Ajax service

The Grid Coach View calls an Ajax service to retrieve a logical page of data. Each call returns 25 rows as well as the total number of rows. As the user scrolls down, the Grid Coach View automatically retrieves the next 25 rows with another Ajax call. The scroll bar always reflects the total number of rows so that the user has the impression that they are working with the entire data set, when in fact, they are rarely working with more than 25 or 50 rows.

Standard inputs

The Ajax service has standard inputs:

- Start indicates the index of the first row to return
- Count indicates how many rows to return
- Sort indicates what column the results should be sorted by
- Additional inputs are defined for filters that will be applied

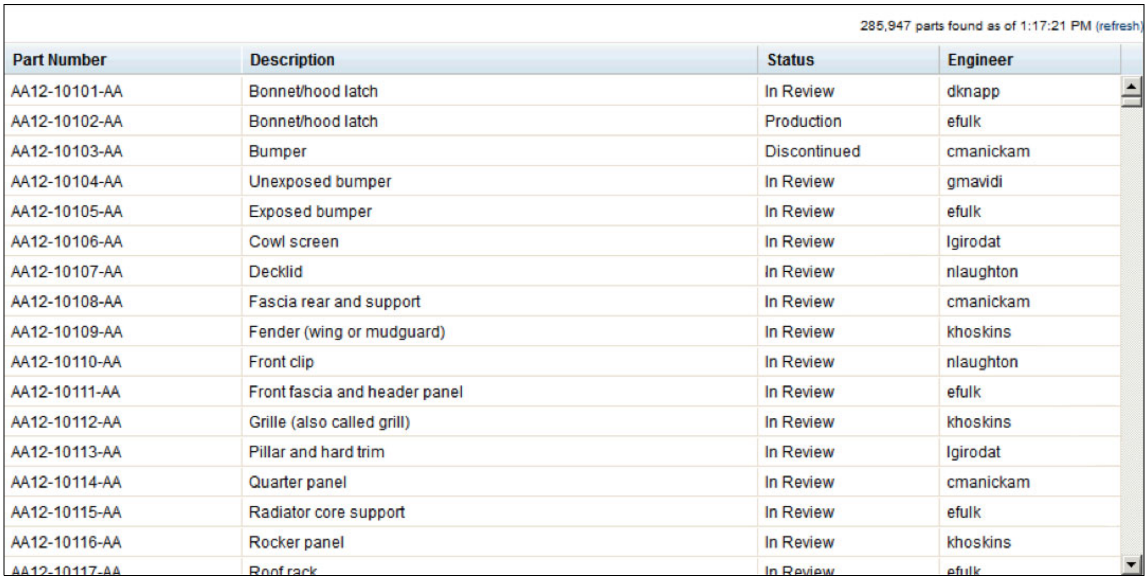
Standard outputs

The Ajax service has standard outputs:

- Items is the array of results that will be displayed in the grid
- Identifier is the name of the attribute that uniquely identifies a row
- numRows is the total number of rows that match the filter criteria

Set the list service

After developing the Ajax service, go to the Grid Coach View and change the configuration option from the default service to your new service. In this example, the service is called “List Page of Parts”. Change the “name” field on the first column to “partNumber” and run the service to see the results as shown in Figure 5-17.



285,947 parts found as of 1:17:21 PM (refresh)

Part Number	Description	Status	Engineer
AA12-10101-AA	Bonnet/hood latch	In Review	dknapp
AA12-10102-AA	Bonnet/hood latch	Production	efulk
AA12-10103-AA	Bumper	Discontinued	cmanickam
AA12-10104-AA	Unexposed bumper	In Review	gmavidi
AA12-10105-AA	Exposed bumper	In Review	efulk
AA12-10106-AA	Cowl screen	In Review	lgirodat
AA12-10107-AA	Decklid	In Review	nlaughton
AA12-10108-AA	Fascia rear and support	In Review	cmanickam
AA12-10109-AA	Fender (wing or mudguard)	In Review	khoskins
AA12-10110-AA	Front clip	In Review	nlaughton
AA12-10111-AA	Front fascia and header panel	In Review	efulk
AA12-10112-AA	Grille (also called grill)	In Review	khoskins
AA12-10113-AA	Pillar and hard trim	In Review	lgirodat
AA12-10114-AA	Quarter panel	In Review	cmanickam
AA12-10115-AA	Radiator core support	In Review	efulk
AA12-10116-AA	Rocker panel	In Review	khoskins
AA12-10117-AA	Roof rack	In Review	efulk

Figure 5-17 Parts grid with the Ajax service

Experience excellent performance

The page and grid are displayed before the rows are fetched for improved response time. In this example, the Parts grid shows over 280,000 parts. The data is displayed in a few seconds because only 25 rows are actually sent to the client. Subsequent scrolling and sorting operations are also very fast. Regardless of the size of the grid or the number of users, no information is residing in memory on the server. Finally, only five Coach View instances were created on the client so performance on the client is very good.

Add co-dependent filters

Working with large data sets usually involves filters to focus on the items of interest. The best usability comes from a set of co-dependent filters. Co-dependent filters filter the grid and each other.

Add the filters

In this example, we add three filters:

- Base Part Number
- Status
- Engineer

The filters are Lookup fields that are placed in a Horizontal Section above the grid as shown in Figure 5-18.

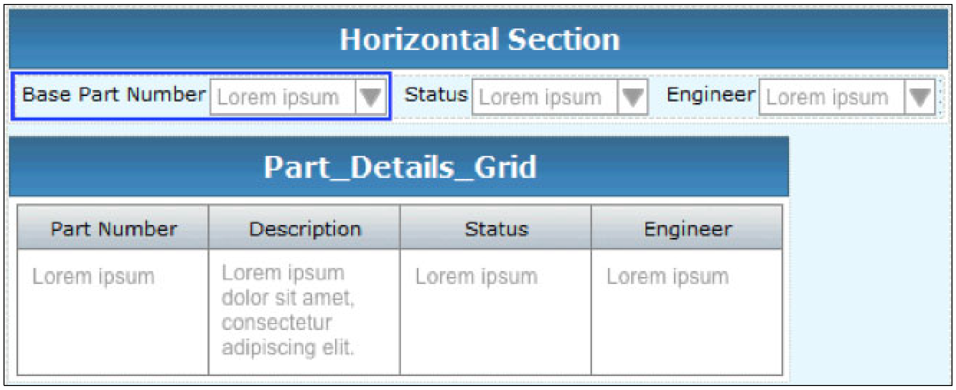
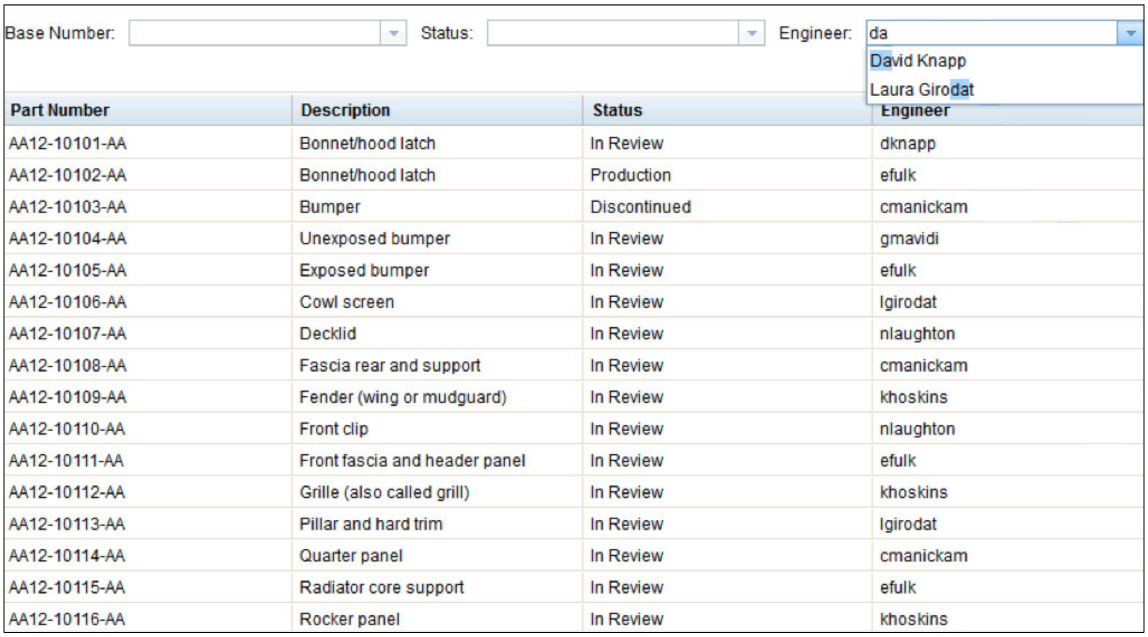


Figure 5-18 Grid with filters

Each filter has two key configuration options:

- Dataspace Binding defines the client-side variable where the filter value will be stored (“status” for example).
- List Service identifies the Ajax service that retrieves the values (“List Page of Part Engineers” for example).

The user can select the drop-down arrow to see the first page of engineers. More commonly, the user enters a few characters and the system finds the matching values, which are shown in Figure 5-19.



Part Number	Description	Status	Engineer
AA12-10101-AA	Bonnet/hood latch	In Review	dknapp
AA12-10102-AA	Bonnet/hood latch	Production	efulk
AA12-10103-AA	Bumper	Discontinued	cmanickam
AA12-10104-AA	Unexposed bumper	In Review	gmavidi
AA12-10105-AA	Exposed bumper	In Review	efulk
AA12-10106-AA	Cowl screen	In Review	lgirodat
AA12-10107-AA	Decklid	In Review	nlaughton
AA12-10108-AA	Fascia rear and support	In Review	cmanickam
AA12-10109-AA	Fender (wing or mudguard)	In Review	khoskins
AA12-10110-AA	Front clip	In Review	nlaughton
AA12-10111-AA	Front fascia and header panel	In Review	efulk
AA12-10112-AA	Grille (also called grill)	In Review	khoskins
AA12-10113-AA	Pillar and hard trim	In Review	lgirodat
AA12-10114-AA	Quarter panel	In Review	cmanickam
AA12-10115-AA	Radiator core support	In Review	efulk
AA12-10116-AA	Rocker panel	In Review	khoskins

Figure 5-19 Type ahead values in filters

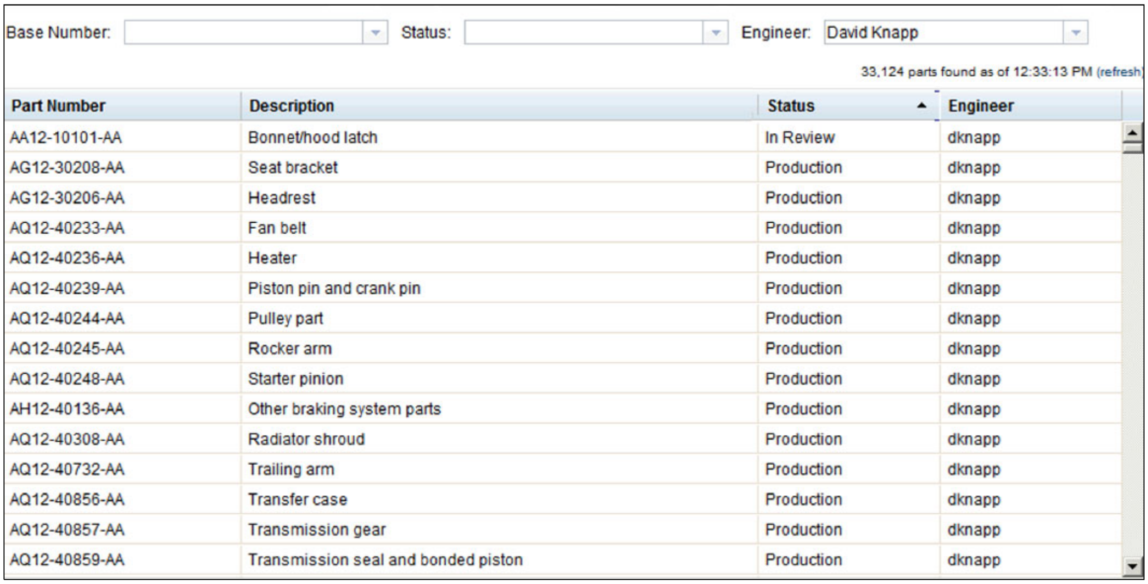
Connect the filters to the grid

In this example, the three filters have data space binding of “basePartNumber”, “status”, and “engineer” respectively. To connect the filters to the grid, simply give the list of values to use as filters and set the grid to refresh automatically as shown in Table 5-4.

Table 5-4 Connecting filters to the grid

Configuration Option	Value	Comments
Filters	“basePartNumber, status, engineer”	The comma-separated list of data space binding values that will be used as filters
Refresh	Automatically	Any time that any of the filter values change, the grid will automatically be refreshed

When a value is selected for a filter, the grid is automatically filtered as shown in Figure 5-20 on page 166.



Base Number: Status: Engineer: David Knapp

33,124 parts found as of 12:33:13 PM (refresh)

Part Number	Description	Status	Engineer
AA12-10101-AA	Bonnet/hood latch	In Review	dknapp
AG12-30208-AA	Seat bracket	Production	dknapp
AG12-30206-AA	Headrest	Production	dknapp
AQ12-40233-AA	Fan belt	Production	dknapp
AQ12-40236-AA	Heater	Production	dknapp
AQ12-40239-AA	Piston pin and crank pin	Production	dknapp
AQ12-40244-AA	Pulley part	Production	dknapp
AQ12-40245-AA	Rocker arm	Production	dknapp
AQ12-40248-AA	Starter pinion	Production	dknapp
AH12-40136-AA	Other braking system parts	Production	dknapp
AQ12-40308-AA	Radiator shroud	Production	dknapp
AQ12-40732-AA	Trailing arm	Production	dknapp
AQ12-40856-AA	Transfer case	Production	dknapp
AQ12-40857-AA	Transmission gear	Production	dknapp
AQ12-40859-AA	Transmission seal and bonded piston	Production	dknapp

Figure 5-20 Grid with filter applied

Make the filters co-dependent

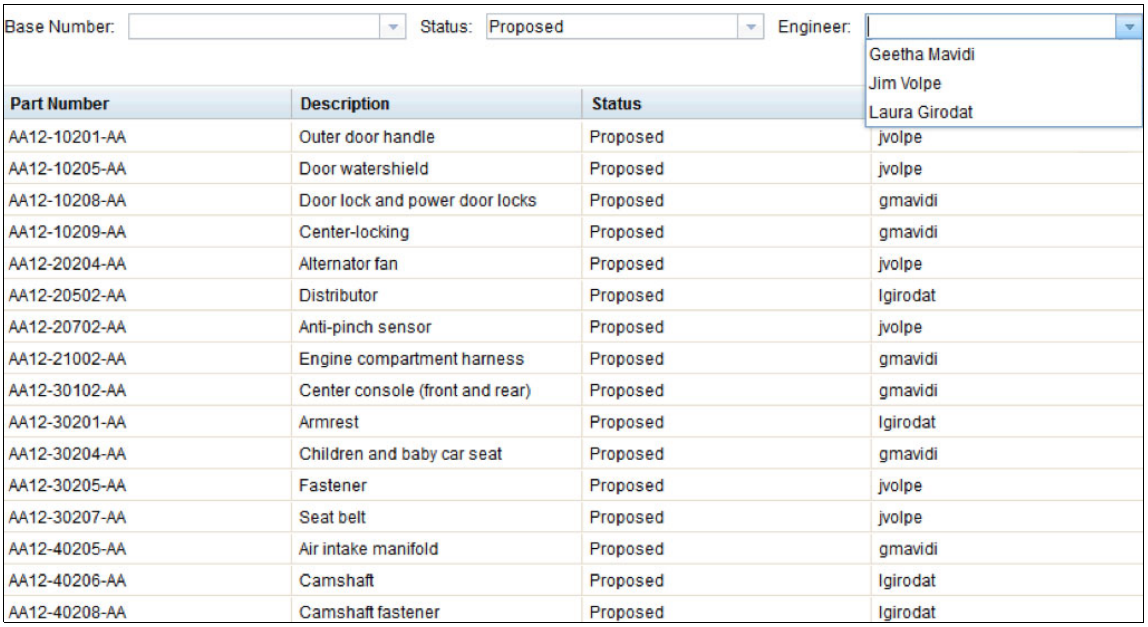
At this point, each of the filters work independently. The grid is filtered by the intersection of the filters but each filter ignores the other filters and presents the full list of values. This results in a bad user experience because selecting some filter combinations will result in no records being displayed in the grid.

To make the filters co-dependent, simply enter the bindings of the other views in the Filter configuration option as shown in Table 5-5.

Table 5-5 Filter configuration options

Coach View	Filter Value
Base Number Filter	status, engineer
Status Filter	baseNumber, engineer
Engineer Filter	baseNumber, status
Parts Grid	baseNumber, status, engineer

After making the filters co-dependent, if a value is selected in one filter, the results of the other filters are reduced as shown in Figure 5-21.


Base Number: | Status: Proposed | Engineer: (Geetha Mavidi, Jim Volpe, Laura Girodat)

Part Number	Description	Status	
AA12-10201-AA	Outer door handle	Proposed	jvolpe
AA12-10205-AA	Door watershield	Proposed	jvolpe
AA12-10208-AA	Door lock and power door locks	Proposed	gmavidi
AA12-10209-AA	Center-locking	Proposed	gmavidi
AA12-20204-AA	Alternator fan	Proposed	jvolpe
AA12-20502-AA	Distributor	Proposed	lgirodat
AA12-20702-AA	Anti-pinch sensor	Proposed	jvolpe
AA12-21002-AA	Engine compartment harness	Proposed	gmavidi
AA12-30102-AA	Center console (front and rear)	Proposed	gmavidi
AA12-30201-AA	Armrest	Proposed	lgirodat
AA12-30204-AA	Children and baby car seat	Proposed	gmavidi
AA12-30205-AA	Fastener	Proposed	jvolpe
AA12-30207-AA	Seat belt	Proposed	jvolpe
AA12-40205-AA	Air intake manifold	Proposed	gmavidi
AA12-40206-AA	Camshaft	Proposed	lgirodat
AA12-40208-AA	Camshaft fastener	Proposed	lgirodat

Figure 5-21 Co-dependent filter results

Local actions

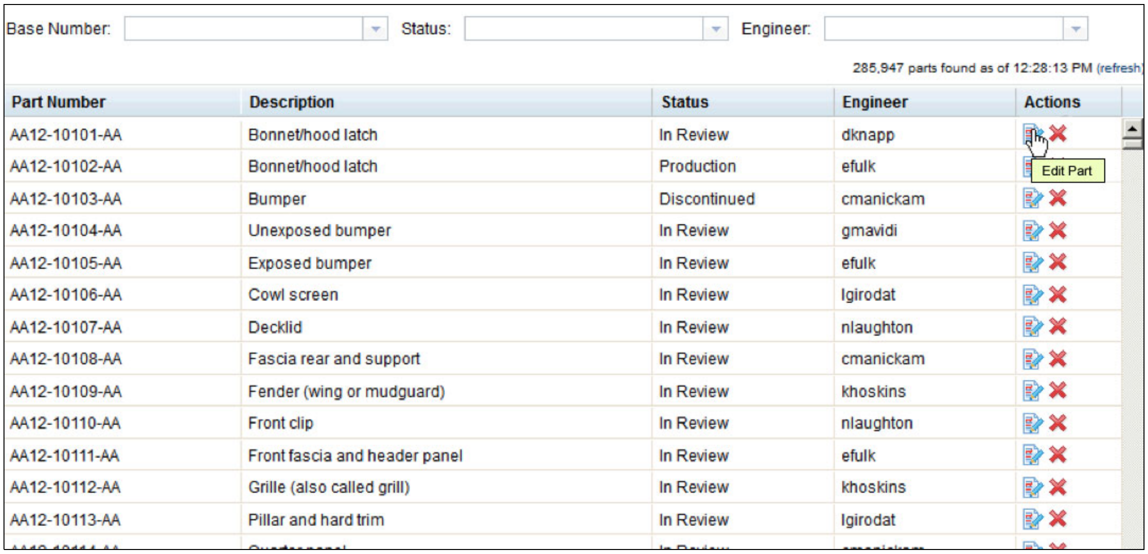
Local actions are operations that a user can do to a specific item in a grid. In this example, we add an Edit and Delete local action to our Parts grid.

Add the actions column and actions

Add the actions column and local actions as follows:

- Drag the Actions Column Coach View into the grid and set the label to "Actions."
- Drag the Action Coach View into the actions column, set the label to "Edit" and the icon to "Edit.png."
- Drag the Action Coach View into the actions column, set the label to "Delete" and the icon to "Delete.png."

Run the service and the grid will now have actions as shown in Figure 5-22 on page 168.


Base Number: Status: Engineer:

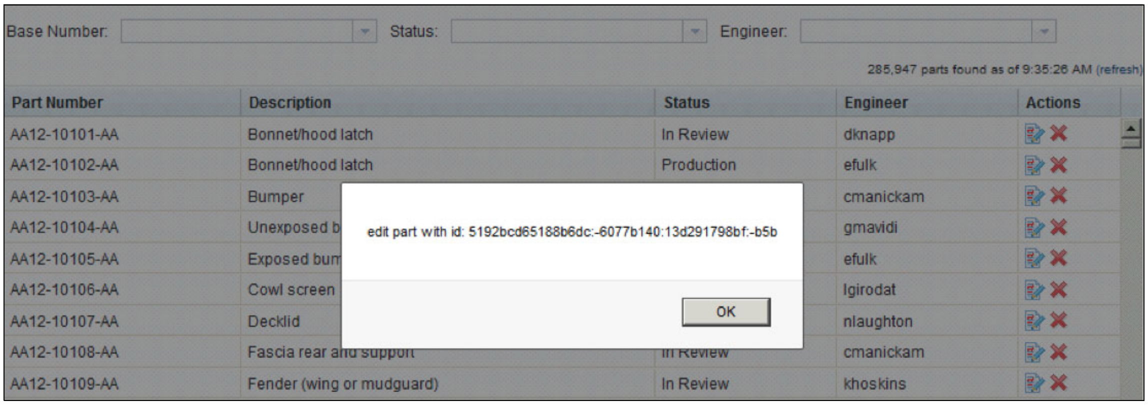
285,947 parts found as of 12:28:13 PM (refresh)

Part Number	Description	Status	Engineer	Actions
AA12-10101-AA	Bonnet/hood latch	In Review	dknapp	
AA12-10102-AA	Bonnet/hood latch	Production	efulk	Edit Part
AA12-10103-AA	Bumper	Discontinued	cmanickam	
AA12-10104-AA	Unexposed bumper	In Review	gmavidi	
AA12-10105-AA	Exposed bumper	In Review	efulk	
AA12-10106-AA	Cowl screen	In Review	lgirodat	
AA12-10107-AA	Decklid	In Review	nlaughton	
AA12-10108-AA	Fascia rear and support	In Review	cmanickam	
AA12-10109-AA	Fender (wing or mudguard)	In Review	khoskins	
AA12-10110-AA	Front clip	In Review	nlaughton	
AA12-10111-AA	Front fascia and header panel	In Review	efulk	
AA12-10112-AA	Grille (also called grill)	In Review	khoskins	
AA12-10113-AA	Pillar and hard trim	In Review	lgirodat	

Figure 5-22 Parts grid with actions

Make the actions do something

Each Action Coach View has an onClick configuration option that is used to specify a JavaScript snippet that will be executed when the user clicks the action. The "id" and "item" options are available in this context. For a quick test, set the onClick configuration option of the Edit action to "alert('edit part with id: '+id);" and run the service. Clicking the edit action shows the alert as shown in Figure 5-23.


Base Number: Status: Engineer:

285,947 parts found as of 9:35:26 AM (refresh)

Part Number	Description	Status	Engineer	Actions
AA12-10101-AA	Bonnet/hood latch	In Review	dknapp	
AA12-10102-AA	Bonnet/hood latch	Production	efulk	
AA12-10103-AA	Bumper		cmanickam	
AA12-10104-AA	Unexposed b		gmavidi	
AA12-10105-AA	Exposed bum		efulk	
AA12-10106-AA	Cowl screen		lgirodat	
AA12-10107-AA	Decklid		nlaughton	
AA12-10108-AA	Fascia rear and support	In Review	cmanickam	
AA12-10109-AA	Fender (wing or mudguard)	In Review	khoskins	

edit part with id: 5192bcd65188b6dc:-6077b140:13d291798bf:-b5b

OK

Figure 5-23 Edit with alert

Display the dialog

In this example, the edit action will be accomplished using an Edit Part dialog. The design of this is shown in Figure 5-24.

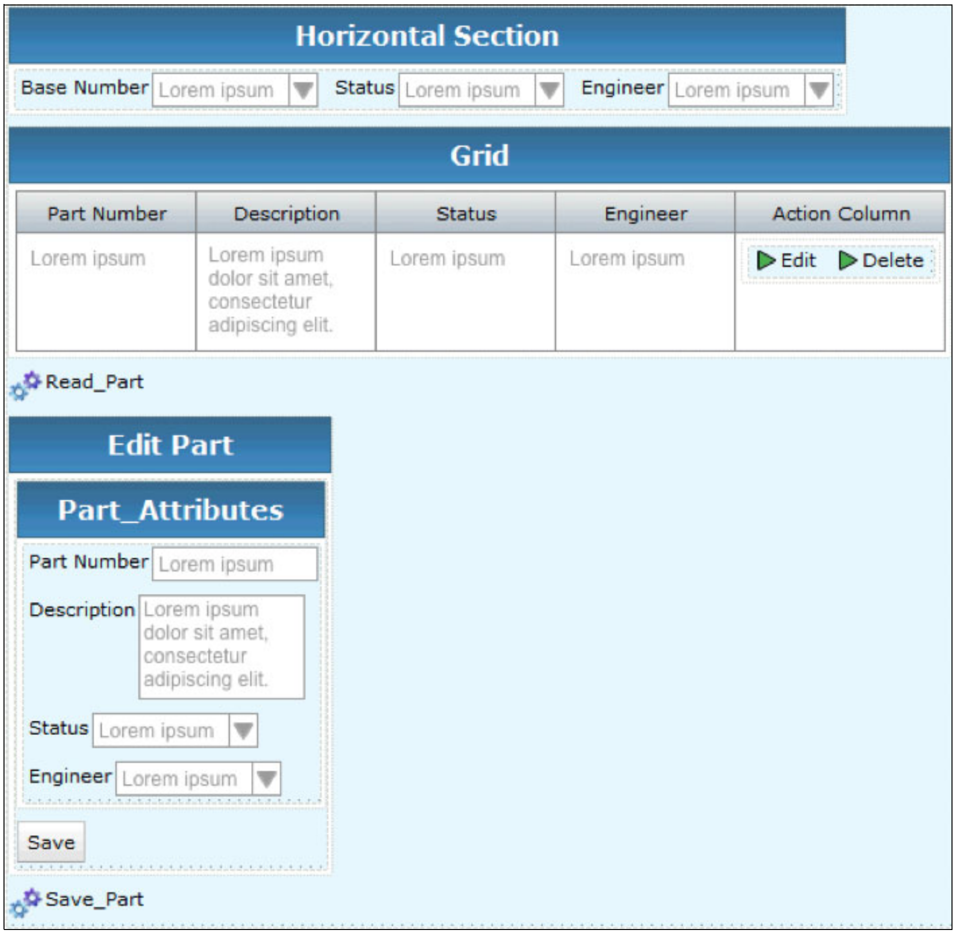


Figure 5-24 Edit part dialog design

The Coach Views will interact with each other using events. When the user clicks the Edit action, the onClick configuration option executes this JavaScript snippet:

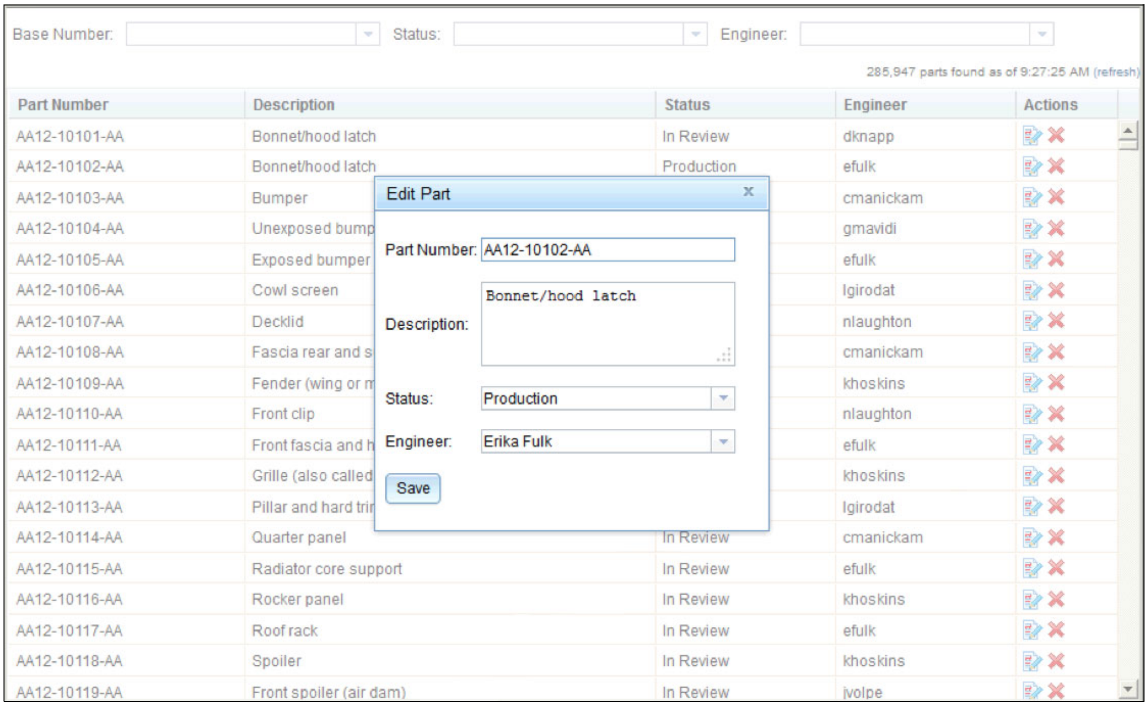
```
views.Read_Part.execute({partId: id});
```

The .execute() method initiates the read of the part. In the previous Service Call, the inputs were automatically pulled from the data space. In this case, the inputs are explicitly specified using the JSON string “{partId: id}”. The Action Coach View makes the id available in the context that the JavaScript snippet is run.

When the Read_Part Coach View finishes reading the part, the outputs are put into the data space and the onExecuteComplete configuration option executes this JavaScript snippet:

```
views.Edit_Part_Dialog.show();
```

The .show() method of the dialog opens it with the fields values completed as shown in Figure 5-25.



Base Number: Status: Engineer:

285,947 parts found as of 9:27:25 AM (refresh)

Part Number	Description	Status	Engineer	Actions
AA12-10101-AA	Bonnet/hood latch	In Review	dknapp	✎ ✗
AA12-10102-AA	Bonnet/hood latch	Production	efulk	✎ ✗
AA12-10103-AA	Bumper		cmanickam	✎ ✗
AA12-10104-AA	Unexposed bump		gmavidi	✎ ✗
AA12-10105-AA	Exposed bumper		efulk	✎ ✗
AA12-10106-AA	Cowl screen		lgirodat	✎ ✗
AA12-10107-AA	Decklid		nlaughton	✎ ✗
AA12-10108-AA	Fascia rear and s		cmanickam	✎ ✗
AA12-10109-AA	Fender (wing or m		khoskins	✎ ✗
AA12-10110-AA	Front clip		nlaughton	✎ ✗
AA12-10111-AA	Front fascia and h		efulk	✎ ✗
AA12-10112-AA	Grille (also called		khoskins	✎ ✗
AA12-10113-AA	Pillar and hard tri		lgirodat	✎ ✗
AA12-10114-AA	Quarter panel	In Review	cmanickam	✎ ✗
AA12-10115-AA	Radiator core support	In Review	efulk	✎ ✗
AA12-10116-AA	Rocker panel	In Review	khoskins	✎ ✗
AA12-10117-AA	Roof rack	In Review	efulk	✎ ✗
AA12-10118-AA	Spoiler	In Review	khoskins	✎ ✗
AA12-10119-AA	Front spoiler (air dam)	In Review	jvolpe	✎ ✗

Edit Part

Part Number: AA12-10102-AA

Description: Bonnet/hood latch

Status: Production

Engineer: Erika Fulk

Save

Figure 5-25 Edit part dialog in action

Save the part

Edit the values and the click the **Save** button. The onClick configuration option of the Save button executes this JavaScript snippet:

```
views.Save_Part.execute();
```

Hide the dialog and refresh the grid

The Save_Part service all calls the Ajax service to save the part and when it is complete, the onExecuteComplete configuration option executes the following JavaScript snippet:

```
views.Edit_Part_Dialog.hide().reset();   views.Parts_Grid.refresh();
```

The .hide() method closes the dialog and the .reset() method clears the values for the next time that the dialog is used. The methods .reset(), .hide(), .show(), .optional(), and .required(), can be called either on specific fields or on a container to apply to all fields. The .refresh() method will refresh the grid to reflect the new values.

Global actions

Global actions are operations that do not apply to a specific item in the grid. In this example, we add an Add Part global action for the grid.

Add the Add Part button

Create a Horizontal Section above the grid and set Show Label to “No” and Align to “Right”. Add a Button to the section, set the Label to “Add Part” and the Icon to “Create.png”. The design is shown in Figure 5-26 on page 172.

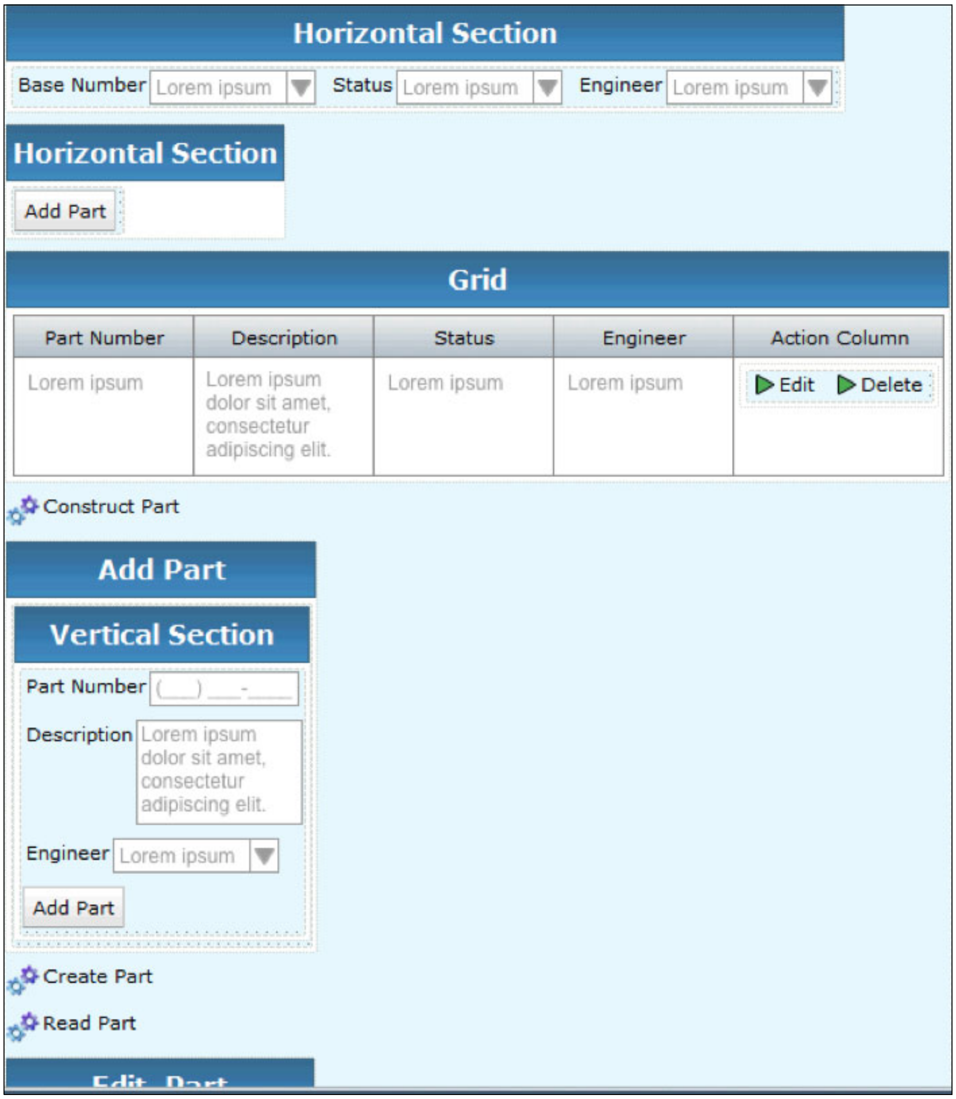
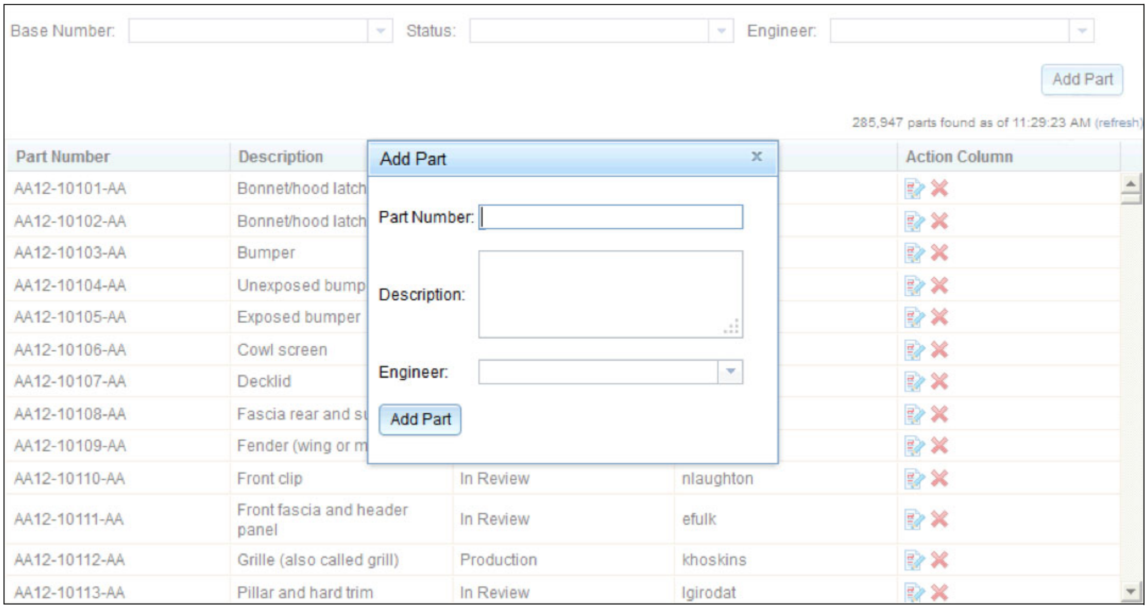


Figure 5-26 Add Part design

The result is shown in Figure 5-27 on page 173.



Base Number: Status: Engineer:

Add Part

285,947 parts found as of 11:29:23 AM (refresh)

Part Number	Description			Action Column
AA12-10101-AA	Bonnet/hood latch			
AA12-10102-AA	Bonnet/hood latch			
AA12-10103-AA	Bumper			
AA12-10104-AA	Unexposed bump			
AA12-10105-AA	Exposed bumper			
AA12-10106-AA	Cowl screen			
AA12-10107-AA	Decklid			
AA12-10108-AA	Fascia rear and su			
AA12-10109-AA	Fender (wing or m			
AA12-10110-AA	Front clip	In Review	nlaughton	
AA12-10111-AA	Front fascia and header panel	In Review	efulk	
AA12-10112-AA	Grille (also called grill)	Production	khoskins	
AA12-10113-AA	Pillar and hard trim	In Review	lgirodat	

Add Part

Part Number:

Description:

Engineer:

Add Part

Figure 5-27 Add Part global action

Make the button add a part

The pattern for Add Part is very similar to Edit Part. The onClick of the Add Part button initiates the construction of a part:

```
views.Construct_Part.execute();
```

When the part is constructed, the onExecuteComplete shows the dialog:

```
views.Add_Part_Dialog.show();
```

When the user clicks the Add Part button, it initiates the create:

```
views.Create_Part.execute();
```

When the part is created, the dialog is hidden and reset and the grid is refreshed:

```
views.Add_Part_Dialog.hide().reset();   views.Parts_Grid.refresh();
```

5.3.3 Integrate directly with a system of record

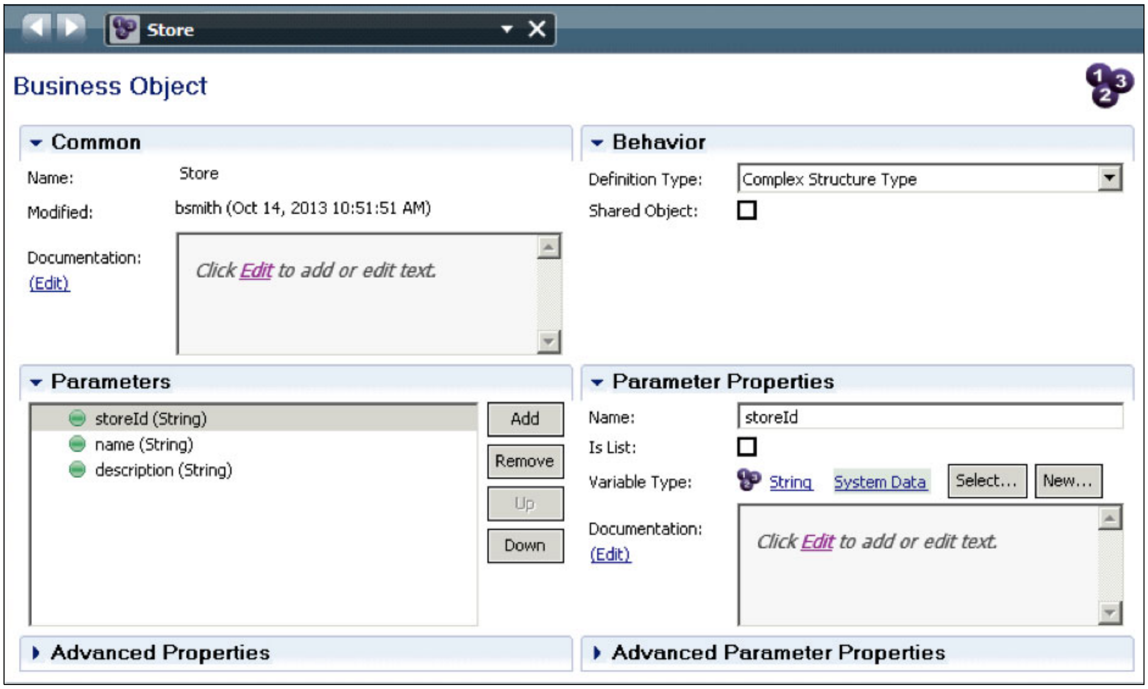
The previous section illustrated how Apex Coach Views leverage client-side data to handle interactions with large sets of data. The patterns include two types of Ajax services:

- "List Page of Objects" services that returned a logical page of Business Objects from the system of record.
- "CRUD" services that create, read, update, or delete Business Objects in the system of record.

While the pattern works very well, implementation effort is still required to build a set of Ajax services for each Business Object in the system of record. This section shows how Apex is using Coach Views to eliminate this work and do direct integration with the system of record on the client.

Set up the Store Business Object

The example that we will use is managing stores for a store order process. Create a new Business Object called "Store" and add the parameters as shown in Figure 5-28.



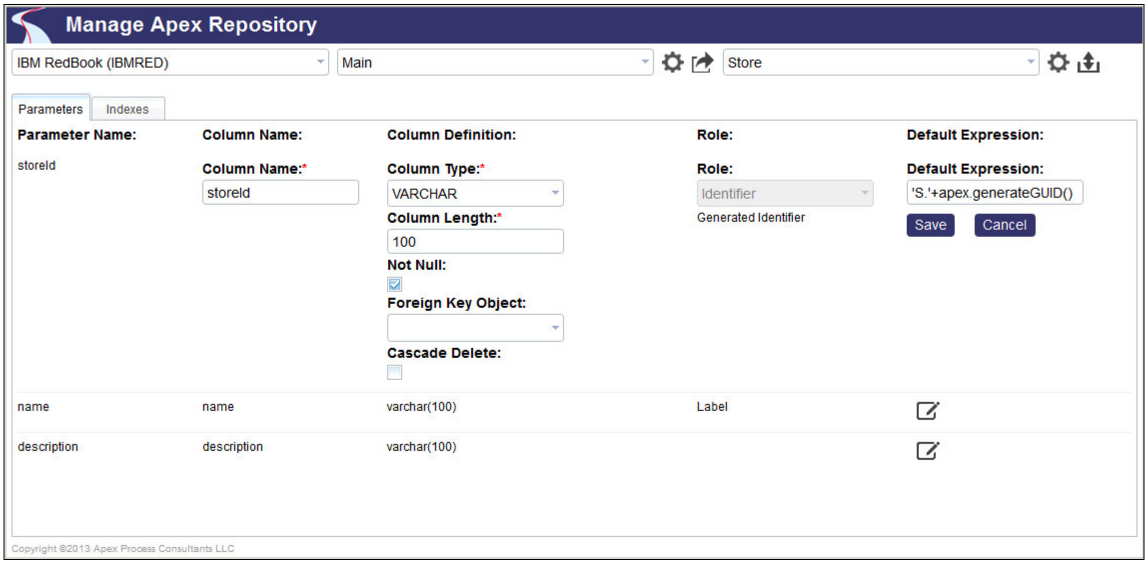
Name: Store
Modified: bsmith (Oct 14, 2013 10:51:51 AM)
Documentation: (Edit) Click Edit to add or edit text.
Definition Type: Complex Structure Type
Shared Object: ☐

storeId (String)
name (String)
description (String)

Name: storeId
Is List: ☐
Variable Type: String System Data
Documentation: (Edit) Click Edit to add or edit text.

Figure 5-28 Store Business Object

Next, run the Manage Repository service in the Apex Repository to configure the Business Object persistence. As you can see in Figure 5-29, rules have been applied to configure the persistence.


Manage Apex Repository

IBM RedBook (IBMRED) | Main | Store

Parameters | Indexes

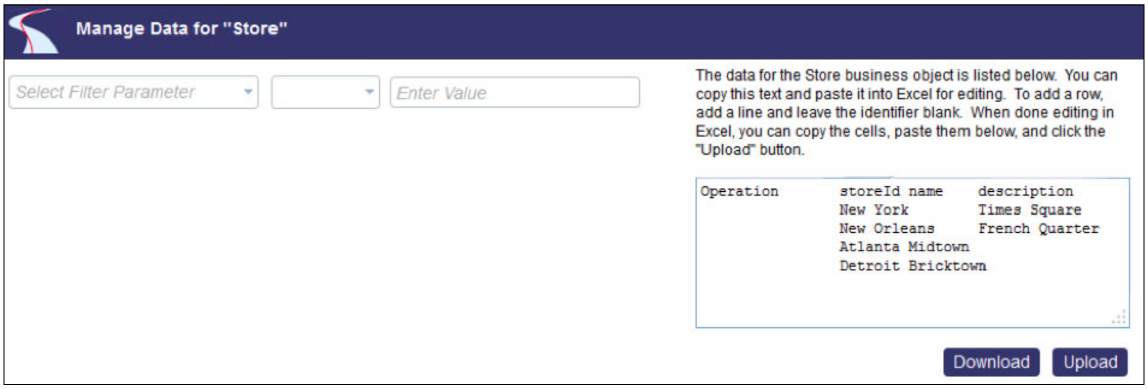
Parameter Name:	Column Name:	Column Definition:	Role:	Default Expression:
storeId	Column Name:* storeId	Column Type:* VARCHAR; Column Length:* 100; Not Null: ✓; Foreign Key Object: ; Cascade Delete: ☐	Role: Identifier; Generated Identifier	Default Expression: 'S.'+apex.generateGUID()
name	name	varchar(100)	Label	
description	description	varchar(100)		

Copyright ©2013 Apex Process Consultants LLC

Figure 5-29 Persistence settings for the Store Business Object

After the settings are confirmed, the system automatically creates the tables, views, indexes, and foreign keys that are required to manage the Store Business Object.

Finally, to support development and testing, store data is copied from Excel and pasted into the Manage Data window to create the Store Business Objects (see Figure 5-30).


Manage Data for "Store"

The data for the Store business object is listed below. You can copy this text and paste it into Excel for editing. To add a row, add a line and leave the identifier blank. When done editing in Excel, you can copy the cells, paste them below, and click the "Upload" button.

```
Operation           storeId name       description
                    New York           Times Square
                    New Orleans        French Quarter
                    Atlanta Midtown
                    Detroit Bricktown
```

Figure 5-30 Uploading store data

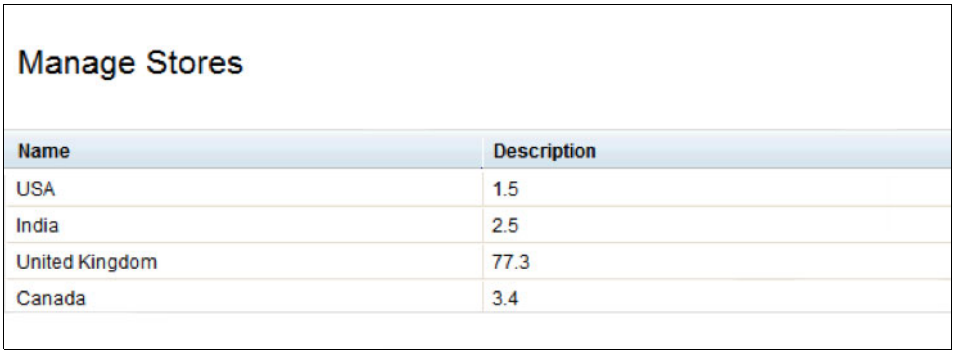
With the Business Object created and the persistence configured, we can move on to the user interface for managing the store Business Objects.

Create a grid of stores

Start by creating a Human Service called "Manage Stores" and expose it as a URL service to the appropriate set of users. Add a Coach with the following structure:

- Page
  - Header
  - Grid
    - Name (Text Column)
    - Description (Multiline Column)
  - Footer

Run the service and you should see the grid with dummy data as shown in Figure Figure 5-31.



Manage Stores

Name	Description
USA	1.5
India	2.5
United Kingdom	77.3
Canada	3.4

Figure 5-31 Stores grid with dummy data

Connect the grid to the system of record

Instead of writing a new Ajax service called "List Page of Stores" and connecting that to the grid, Apex Repository Coach Views will be used to retrieve the data for the grid.

Apex Repository Coach Views include:

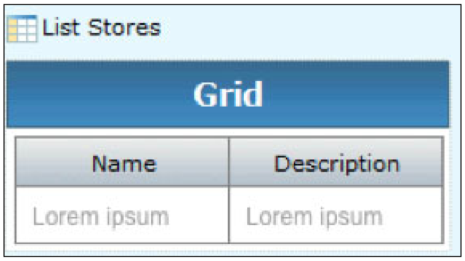
- Construct Coach View initializes a new Business Object
- Create Coach View creates a Business Object in the repository
- Read Coach View reads a Business Object from the repository
- Update Coach View updates a Business Object in the repository
- Delete Coach View removes a Business Object from the repository
- Save Coach View either creates or updates a Business Object in the repository

- Find Coach View finds a Business Object in the repository
- List Objects Coach Views lists Business Objects

In this case, we simply drag the List Objects Coach View onto the canvas and:

- Change its ID to “List_Stores”
- Set the Business Object Name configuration option to “Store”

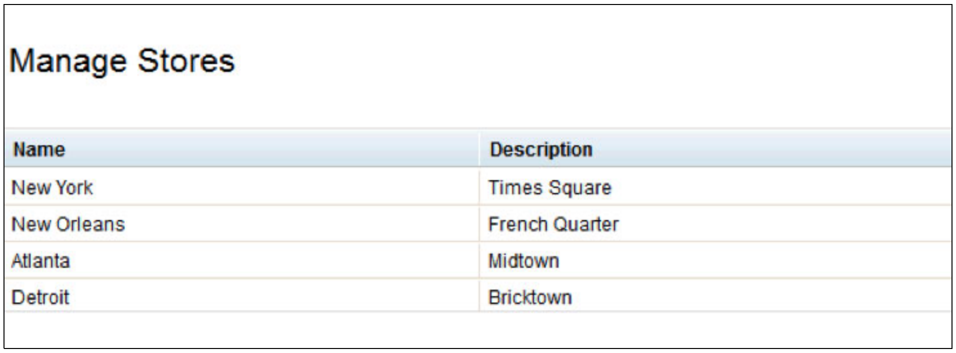
Finally on the Stores Grid, set the Data Source configuration option to “views.List_Stores”. The final design is shown in Figure 5-32.



Name	Description
Lorem ipsum	Lorem ipsum

Figure 5-32 Stores grid design

Run the service and you see the list of stores displayed in your grid as shown in Figure 5-33.


Manage Stores

Name	Description
New York	Times Square
New Orleans	French Quarter
Atlanta	Midtown
Detroit	Bricktown

Figure 5-33 Stores grid with real data

Add the Add Store dialog

The Add Store dialog is modeled using the same pattern that was used in the Add Part dialog. Instead of a Service Call Coach View to construct the store, use the Construct Coach View with the following configuration options:

- Business Object Name is set to “Store”
- Output is set to “store”

The Create Coach View is used in the place of a Service Call Coach View with the following configuration option:

- ► Input is set to “store”

The completed design is shown in Figure 5-34.

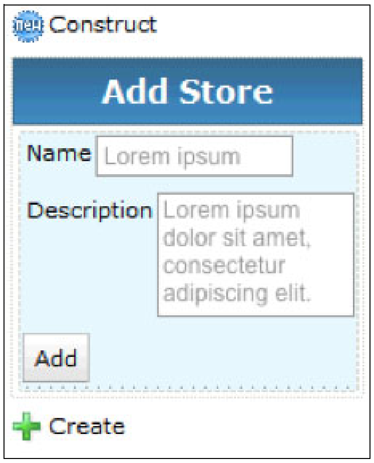


Figure 5-34 Add Store dialog design

Add the Edit Store dialog

The Edit Store dialog is modeled using the same pattern that was used in the Edit Part dialog. Instead of a Service Call Coach View to read the store, use the Read Coach View with the following configuration options:

- ► Business Object Name is set to “Store”
- ► Output is set to “store”

The Update Coach View is used in the place of a Service Call Coach View with the following configuration option:

- ► Dataspace Binding is set to “store”

The completed design is shown in Figure 5-35 on page 179.

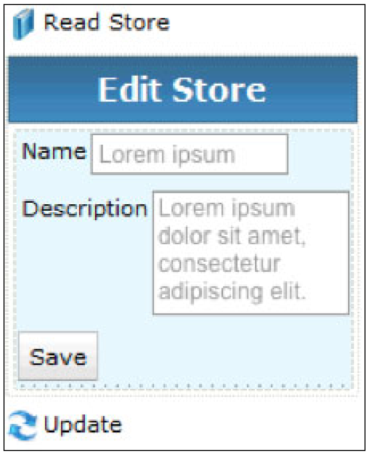


Figure 5-35 Edit Store dialog design

Summary

Using Apex Coach Views layout, grid, and fields with the Apex Repository Coach Views, a complete UI and persistence for a Business Object was built and operational in a fraction of the time that would be required using object-specific persistence services.

5.3.4 Integrate with an external data source

Sections 5.3.2, “Managing large data sets” on page 161and 5.3.3, “Integrate directly with a system of record” on page 174 illustrated client-side data management leveraging information retrieved through Ajax services on the IBM BPM server. Some companies have very stringent security requirements and do not want any data to be in the IBM BPM server at any time.

To meet these requirements, Apex Coach Views include a Representational State Transfer (REST) Call Coach View. This Coach View is designed to retrieve data from, and save data to, external systems through REST application programming interfaces (APIs).

This section shows the REST Call Coach View in action by accessing and updating information in a time and expense Software as a Service (SAAS) application called *Harvest*.

Create the Manage Users grid

First, create a Human Service called "Manage Users" and expose it as a URL to the appropriate users. Add a Coach with the following page structure:

- Page
  - Header
  - Grid
    - First Name (Text Column)
    - Last Name (Text Column)
    - Email (Text Column)
    - Is Admin (Checkbox Column)
  - Footer

Connect the grid to the REST Call

Instead of specifying an IBM BPM Ajax service that retrieves the data, add a REST Call Coach View with ID "List_Users" to the design as shown in Figure 5-36 and Figure 5-37 on page 181.

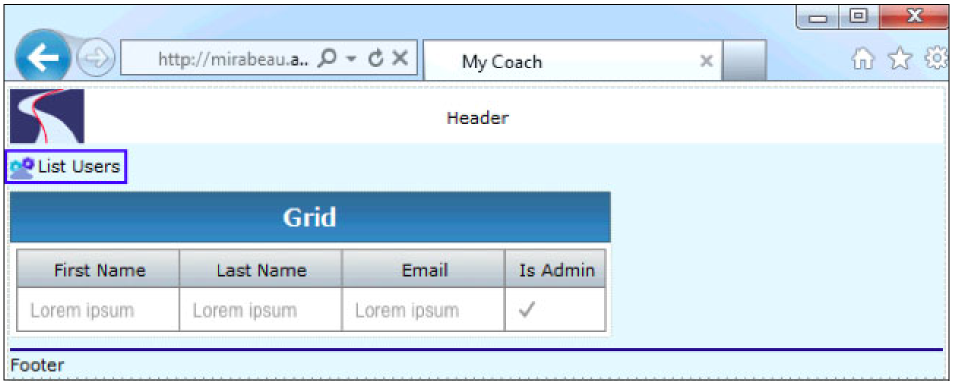
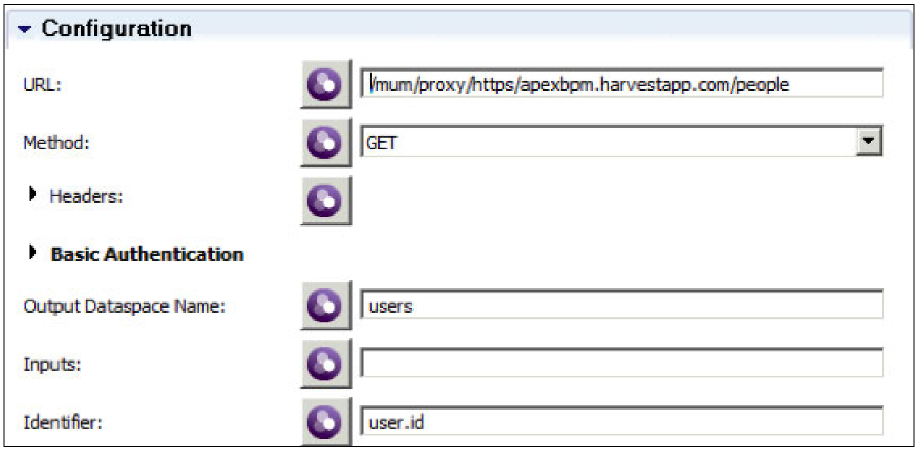


Figure 5-36 List_Users REST Call



URL: /mum/proxy/https/apexbpm.harvestapp.com/people

Method: GET

Output Dataspace Name: users

Inputs:

Identifier: user.id

Figure 5-37 List_Users REST Call configuration options

The configuration options are as follows:

- URL specifies the URL of the REST API that will be called
- Operation specifies that a GET operation will be used
- Output Dataspace Name specifies that the results of the REST API call will be put into “users”
- Identifier specifies that the unique identifier for each user in the list is “user.id”

On the grid, set the Datasource configuration option to “views.List_Users”. This tells the grid to direct its list operations to the List_Users REST API call.

In this example, the REST API from the external system returns information in three layers:

- A list of objects
- A user object
- Attributes of the user object

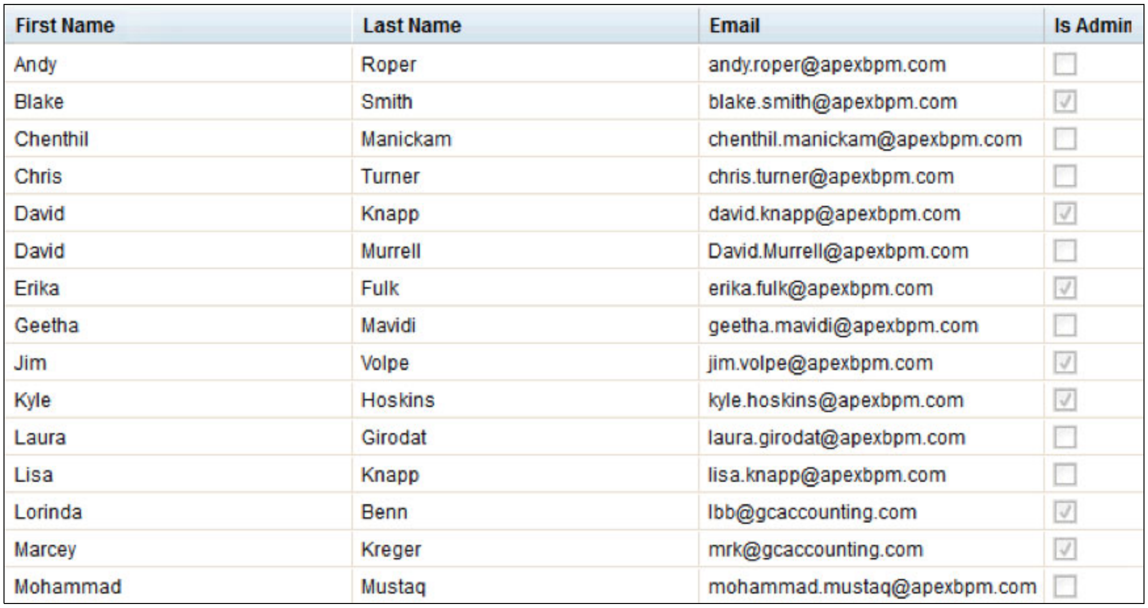
By default, the grid expects to receive a two-layer information structure:

- A list of objects
- Attributes of the object

This is handled by adding the extra layer in the field configuration option on each column:

  – user.first_name
  – user.last_name
  – user.email
  – user.is_admin

The results are shown in Figure 5-38.



First Name	Last Name	Email	Is Admin
Andy	Roper	andy.roper@apexbpm.com	☐
Blake	Smith	blake.smith@apexbpm.com	✓
Chenthil	Manickam	chenthil.manickam@apexbpm.com	☐
Chris	Turner	chris.turner@apexbpm.com	☐
David	Knapp	david.knapp@apexbpm.com	✓
David	Murrell	David.Murrell@apexbpm.com	☐
Erika	Fulk	erika.fulk@apexbpm.com	✓
Geetha	Mavidi	geetha.mavidi@apexbpm.com	☐
Jim	Volpe	jim.volpe@apexbpm.com	✓
Kyle	Hoskins	kyle.hoskins@apexbpm.com	✓
Laura	Girodat	laura.girodat@apexbpm.com	☐
Lisa	Knapp	lisa.knapp@apexbpm.com	☐
Lorinda	Benn	lbb@gcaccounting.com	✓
Marcey	Kreger	mrk@gcaccounting.com	✓
Mohammad	Mustaq	mohammad.mustaq@apexbpm.com	☐

Figure 5-38 Manage Users grid

Add the local and global actions

The Edit and Delete local actions and Add global action are added using the same patterns illustrated in previous sections. The Edit User dialog uses two REST Call Coach Views: Read_User and Update_User.

The Read_User REST Call is configured as follows:

- URL: `/mum/proxy/https/apexbpm.harvestapp.com/people/[[id]]`
- Operation: GET

The URL is parameterized using “[[id]]”. This signifies that the value of “id” in the data space should be put in the URL before the call.

The Update_User REST Call is configured as follows:

- URL: `/mum/proxy/https/apexbpm.harvestapp.com/people/[[id]]`
- Operation: PUT
- Inputs: user

The “Inputs” configuration option specifies that the user object in the data space should be sent in the body of the message.

The Add User dialog uses one REST Call Coach View Add_User that is configured as follows:

- URL: `/mum/proxy/https/apexbpm.harvestapp.com/people`
- Operation: POST
- Inputs: user

Finally, the Delete Confirmation dialog uses one REST Call Coach View (Delete_User) that is configured as follows:

- URL: `/mum/proxy/https/apexbpm.harvestapp.com/people/[[deleteId]]`
- Operation: DELETE

“deleteId” is put into the data space by the Delete local action on the grid before the REST Call is executed.

Summary

The REST Call Coach View, along with the client-side data handling capabilities in Apex Coach Views, gives a powerful combination for designing client-side user interfaces that integrate multiple systems.

5.4 Packaging for reuse

As previous chapters have highlighted, composite views are an excellent way to package presentation. Just create a new composite view, drag atomic views into the layout, and you have a reusable Coach View.

5.4.1 Modeling behavior without Apex Coach Views

While these composite views are useful in making portions of the UI visually consistent, they do not include any behavior. Using the Coach Views delivered with IBM BPM, behavior is modeled using Boundary Events. To appreciate the impact of this, study how to model the Parts Grid without Apex Coach Views.

Parts Grid behavior

The Parts Grid would be implemented using a repeating table. Since it is impractical to load all 280,000 parts into memory at once, paging would need to be simulated and the grid would require three Boundary Events:

- Forward button to load the next page of data
- Backward button to load the previous page of data
- Apply button to reload the current page of data after changes

The Edit Part local action and dialog would add two more Boundary Events:

- Edit Part button to read the part and open the dialog
- Save Part button to update the part in the database and refresh the grid

The Add Part global action and dialog would add two more Boundary Events:

- Add Part button to construct a new part and open the dialog
- Create Part button to insert the part in the database and refresh the grid

The Delete Part local action would add one more Boundary Event:

- Delete button to delete the part in the database and refresh the grid

Completing the functionality by adding a Delete Part local action would add two more Boundary Events resulting in a total of eight Boundary Events. The resulting Human Service diagram is shown in Figure 5-39 on page 185.

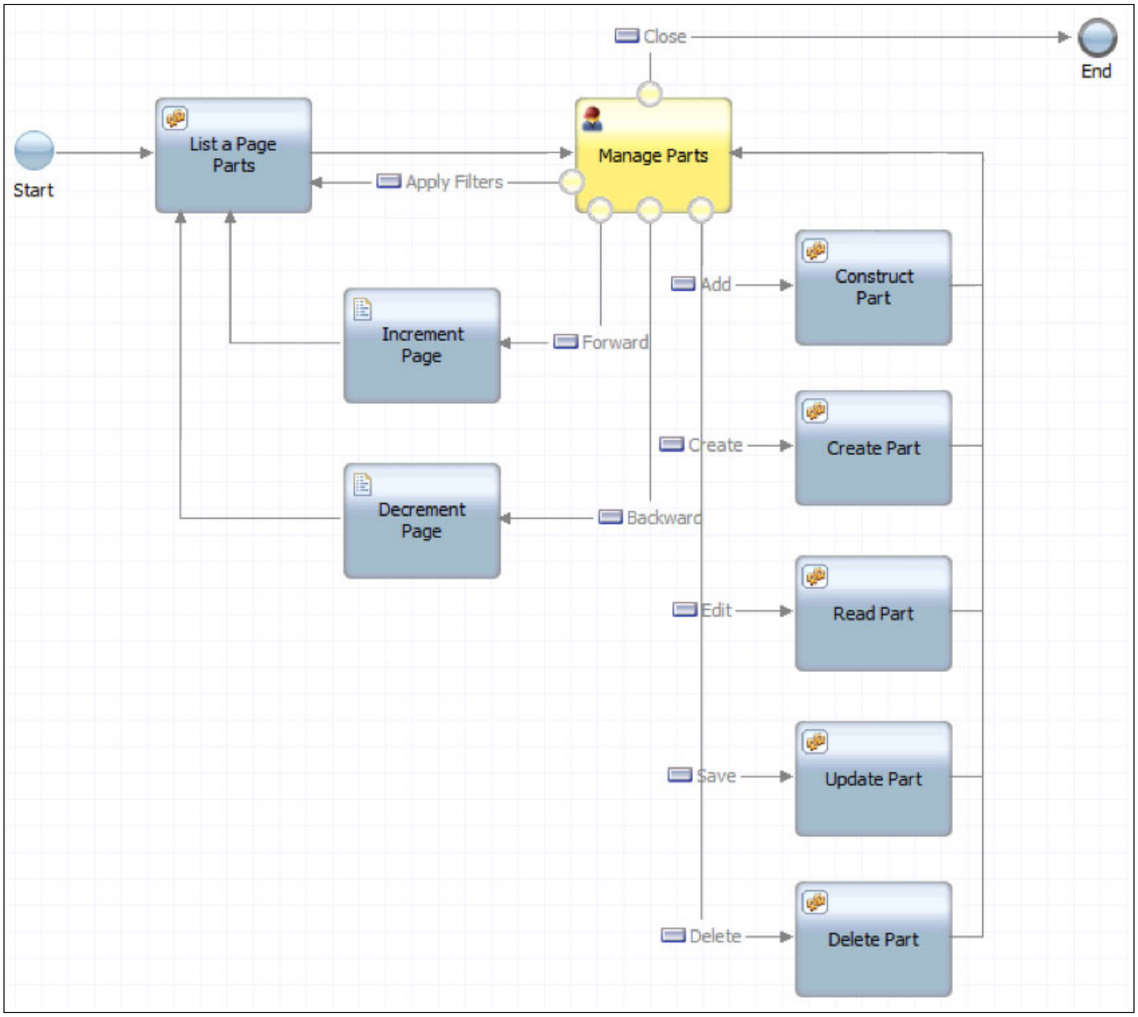


Figure 5-39 Parts grid behavior without Apex Coach Views

Behavior for a simple portal

As you add more functionality to a "one page app" the number of Boundary Events continues to increase and becomes unmanageable. A simple portal with two Business Object tabs (Parts and Products) and a Tasks tab would have 25 Boundary Events as shown in Figure 5-40 on page 186.

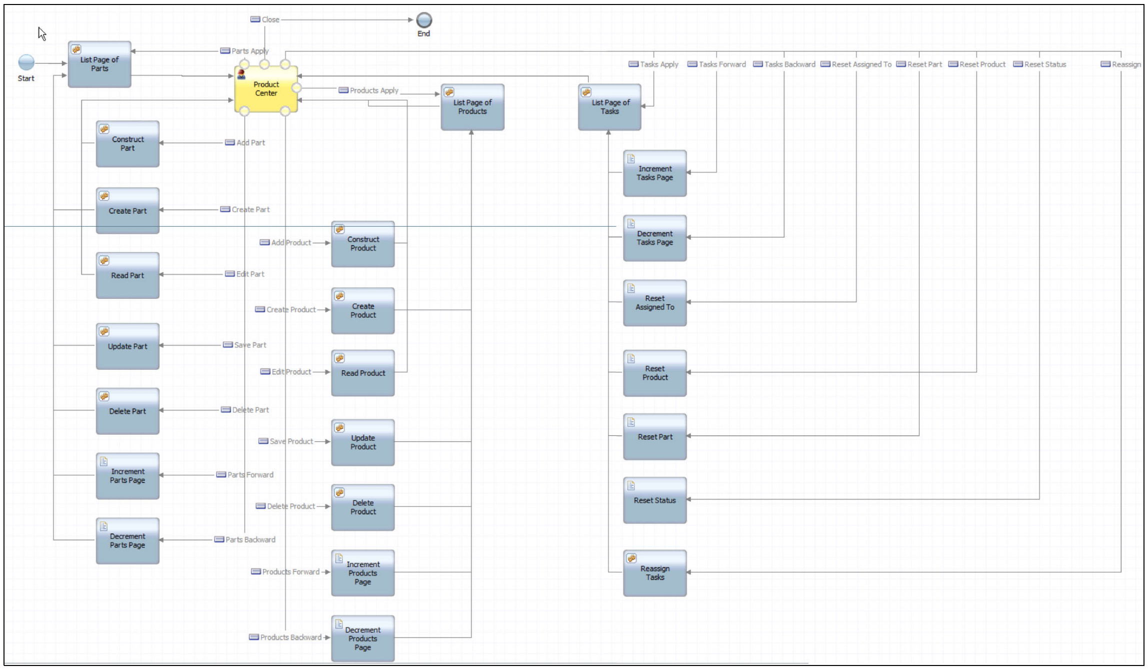


Figure 5-40 Behavior for a simple portal without Apex Coach Views

Behavior duplication makes reuse impractical

Not only does the Human Service become unmanageable, the more fundamental problem is that all of the Boundary Event logic must be duplicated on every Coach that uses the composite view. The resulting increase in modeling effort and opportunity for errors makes heavy reuse impractical.

5.4.2 Packaging behavior without Apex Coach Views

Using the Coach Views that come with IBM BPM, the only way to package behavior in the composite Coach Views is to write code. After a business technical person has defined the content in a composite view by assembling atomic Coach Views on the Layout tab, the composite view is turned over to a developer with level 3 skills to write JavaScript methods in the Behavior tab. Refinements of the composite view layout by the business technical user may break the code in the Behavior tab requiring the developer to get involved again. This back and forth makes iterative refinement much more time-consuming and difficult to achieve.

5.4.3 Packaging behavior with Apex Coach Views

Apex Coach Views include a set of non-visual controls to help a business technical user package a behavior in a composite Coach View. This results in dramatically simplified Human Service as shown in Figure 5-41.

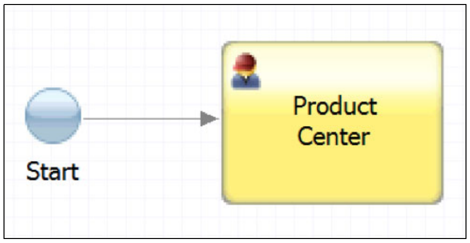


Figure 5-41 Behavior for a simple portal with Apex Coach Views

The packaging views include:

- Dataspace
- Method
- Event

To illustrate how these work, package the Add Part dialog as a reusable component.

Model the basic composite view

Start by creating a new Coach View called "Add Part Dialog". Drag Coach Views onto the Layout tab as follows:

- Add Part Dialog (composite view)
  - Construct_Part (Service Call)
  - Add Part (Dialog)
    - Part Number (Masked Input)
    - Description (Multiline)
    - Engineer (Lookup)
    - Save (Button)
  - Create_Part (Service Call)

Isolate the composite view's data

The Add Part Dialog could be used in many different Coaches. It could even be used multiple times on the same Coach. To ensure that the client-side data in this composite view does not interfere with the caller's data, it is isolated from the caller by adding a Dataspace Coach View.

Add the Show method

When the Add Product Dialog composite is used on a Coach, the caller needs a method to display the dialog. This is done using the Method Coach View. To add the show method, drag the Method Coach View onto the layout and configure it as follows:

- The id defines the name of the method – set it to "show"
- Parameters is a comma-separated list of input parameters – leave it blank
- Implementation is a JavaScript snippet that is executed when the method is called – set it to "views.Construct_Part.execute();"

Add the onAdd event

When the dialog adds a part, the caller needs to be notified so that it can refresh the part grid. This is done using the Event Coach View. To add an "onAdd" event, drag the Event Coach View onto the layout and configure it as follows:

- The id defines the name of the event – set it to "onAdd"
- Parameters is a comma-separated list of output parameters – leave it blank

To trigger the event when the part is added, add this to the onExecuteComplete configuration option of the Create_Part view:

```
parent.onAdd();
```

Finally, to allow the caller to specify what should happen when the event fires, add a configuration option called "onAdd" of type string. The Event Coach View will automatically execute the JavaScript snippet that the caller enters in that configuration option. The completed design is shown in Figure 5-42 on page 189.

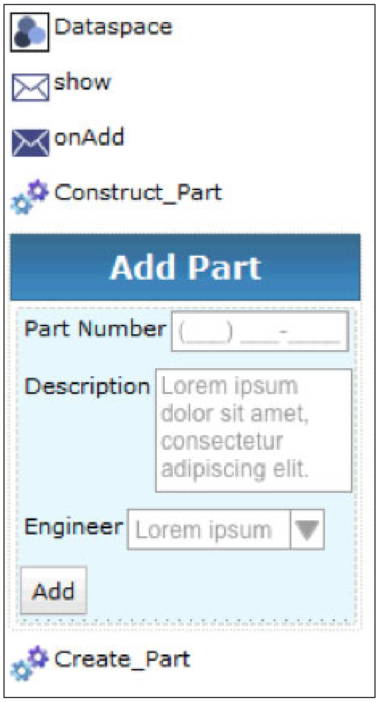


Figure 5-42 Add Part dialog packaged with behavior

Using the composite Coach View

To use the packaged composite view, drag it onto the Coach and note the id – it will be “Add_Part_Dialog1” by default. In order to make the dialog show when the user clicks the Add Part button, set the button’s onClick as follows:

```
views.Add_Part_Dialog1.show();
```

To refresh the grid after a part is added, set the onAdd configuration option of the Add_Part_Dialog1 view as follows:

```
views.Parts_Grid.refresh();
```

5.4.4 Summary

Apex Coach Views’ packaging views make it easy to package reusable user interface components that include both presentation and behavior. This dramatically reduces development time, reduces the need for level 3 skills, and improves the consistency and quality of the end user experience.

5.5 Optimizing performance

This section highlights four approaches that are used to improve performance in Apex Coach Views.

5.5.1 Reduce page load time by deferring processing

When the page loads, the Coach View framework processes each of the views in the page by calling their load functions. As the number of Coach View instances grows, the page load time increases. To mitigate this performance problem, Apex Coach Views give the option to defer the processing of certain types of views.

Defer processing of tab content

The Tab view in Apex Coach Views has a “Load On” configuration option with two values:

- Show
- Page Load

The default is Show, which delays the processing of the views contained inside the tab until the tab is selected by the user. In large “one page apps”, there can be several tabs (and sometimes nested tabs as well). Deferring the processing of the Coach Views until the tab is shown reduces the page load time.

In some cases, it makes sense to process the tab contents immediately after the page loads. This allows Ajax services to retrieve data on the tab before it is actually shown. Setting load on to “Page Load” addresses this scenario.

Defer processing of dialog content

The Dialog in Apex Coach Views is another view that has a “Load On” configuration option. Like Tab, the default is “Show” and is used in most cases.

5.5.2 Reduce page size by packaging Coach View source

As the number of Coach Views used on a page grows, the page size grows as well. This is driven by the fact that the source of each Coach View is in the page source each time. Apex Coach Views reduces the page size by packaging the source for each Coach View in a JavaScript file that will be cached on the client. The three-tab portal that was described in Section 5.4.1, “Modeling behavior without Apex Coach Views” on page 183 includes 114 Coach View instances drawing from 36 different Coach Views. Refactoring the code into a JavaScript file reduced the page size from 297 kb to 182 kb, a 39% reduction.

5.5.3 Reduce page size by modularizing into templates

In order to further reduce the page size, Apex Coach Views introduced templates. A composite Coach View can be wrapped in a Template Wrapper Coach View and stored as a template in the library. To use the template in a Coach or composite Coach View, a Template Loader Coach View is added specifying the name of the template. When the page is sent to the client, the source includes the Coach Views down to and including the Template Loaders. This results in a dramatic reduction in the page size. The portal is reduced from 182 kb to 21 kb, an 88% reduction.

When the page is loaded on the client and a Template Loader Coach View is encountered, it requests the template from the server and inserts it into the page. Templates that are inside tabs or dialogs that the user does not open are not even requested from the server.

5.5.4 Reduce server round trips by caching templates

Although templates reduce the page size dramatically, calling the server each time that a new template is encountered is inefficient. To address this potential issue, the Template Loader view in Apex Coach Views caches the templates on the client. This caching not only improves performance on a second visit to the same page, it also improves performance on any page that uses that template. The templates are stored with a key composed of the template name and the snapshot id. This ensures that the template loader uses the correct version of the template and even supports multiple versions of the same template.

5.6 Conclusion

Apex Coach Views puts powerful user interface capabilities within the reach of business technical users. They improve usability, reduce development cost, and improve performance. Do not waste your time developing your own coach views from scratch. Use Apex Coach Views so that you can focus on delivering business value.

6

Real World Lessons from BP3

BP3 is an IBM OEM partner and reseller exclusively focused on IBM Business Process Manager (BPM) and complementary products such as IBM Blueworks Live™ and ODM. We work with clients across all segments of the economy to deliver sustainable process excellence. BP3 provides several service offerings, specifically designed and managed to help organizations with their BPM programs from the moment they know what the BPM acronym stands for, all the way through ongoing support and maintenance of production applications. BP3's heritage on the IBM BPM platform dates back to 2003 Lombardi when BP3's founders were senior staff at Lombardi, and extends through the present-day:

- BPStrategy approaches process problems from a product agnostic standpoint, using a combination of Lean and Six Sigma to improve processes and determine how BPM can best be applied to accelerate your business.
- BPServices is BP3's core service offering, helping clients with everything from building BPM programs and COEs, to defining and refining processes using Blueworks Live, all the way through implementing process applications on top of BPM.
- BP Labs serves a dual role as both the technical innovation lab for BP3, as well as our support organization for customers. BP Labs develops and curates tools and knowledge on topics such as in-flight migration, solution analysis, and UI customizations. BP Labs employs a team of former

consultants and Lombardi and IBM alumni who work from Austin, TX, and the BP3 European office to fulfill various environment, application, and tool support agreements offered by BP3.

- BPMobility provides expertise on BPM solutions that have a significant user presence from outside the traditional desktop and notebook environment. This includes everything from unique user experience requirements where a custom BPM web portal might need to work on various devices from a mobile phone to a large screen TV all the way through building hybrid and native applications for iOS and Android.

With the introduction of CoachViews in BPM 8.0 and the increasing demand for mobile enablement from our clients, BP3 saw an opportunity to create a UI framework that would yield both desktop and mobile interfaces from a single drag and drop implementation effort using CoachViews. This approach would eliminate the additional cost of mobile-enabling a BPM process by providing the developer with CoachViews that look great and work naturally on either desktop or mobile devices, instead of UIs that were only designed for the desktop.

The response from our clients have been overwhelmingly positive. Not only are they mobile-enabling their processes from day one, but also they are simply able to produce great looking desktop interfaces for BPM, on par with major commercial and social media web sites.

In addition to BP3's services and product offerings, BP3's blog is widely read for its coverage of all topics around BPM, innovation, and process. You can find it at the following location:

http://www.bp-3.com/blogs

6.1 Responsive design

With every new BPM implementation, it becomes clearer to us that the enterprise has transcended the desktop and notebook interface as the only means to get work done. Popular social, e-commerce, and entertainment platforms, have conditioned their users to expect a rich experience. Furthermore, these giants have built native applications for various device types, setting the expectation that those platforms can be accessed from anywhere, without compromising the user experience. Enterprise applications have fallen a bit behind, but they are starting to catch up.

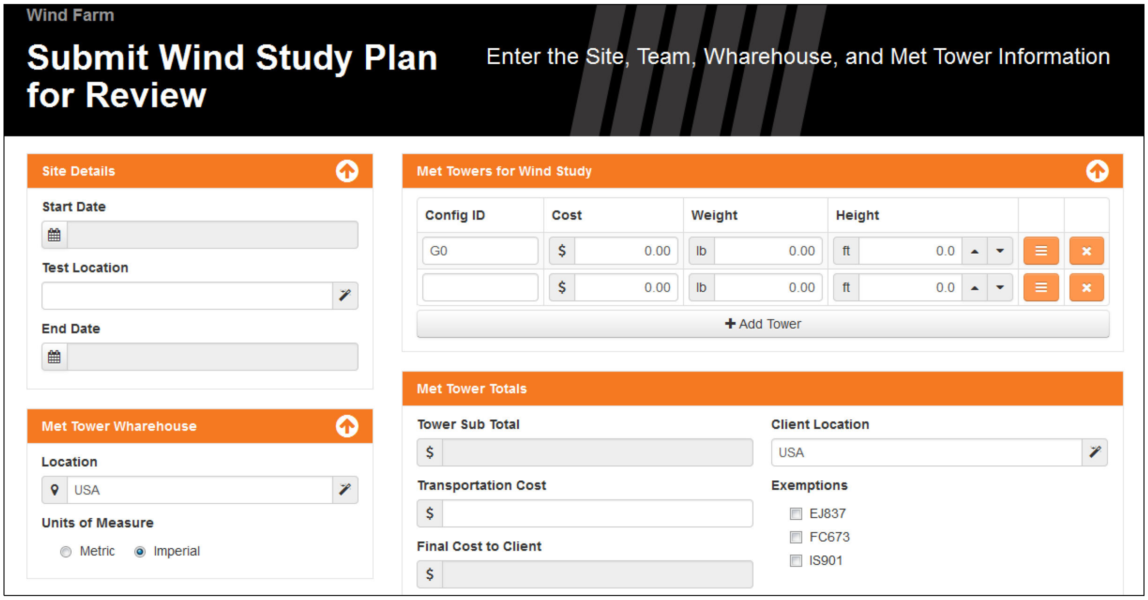
When building a BPM solution that requires mobile participants, creating native applications is conceivably the way to get the absolute best user experience, but it is far from the most cost effective. Even though iOS and Android make up more than 80% of all mobile devices, version fragmentation complicates things for developers. In order to capture most of Android users, you would need to create Android apps for six very different versions. Version fragmentation is much less of an issue with iOS, but it does still exist. Even if the BPM solution needs to support a single device type and version, the cost is still significant due to the additional development effort, and a completely different skill set from traditional BPM Developer.

An alternative to native interfaces is web apps. With standardization in the HTML5 spec, ever more powerful mobile devices, and the significant improvements in browser capabilities in the last few years, many web applications can be built to be nearly indistinguishable from their native counterparts. Furthermore, if implemented properly, the same web app will function uniformly on all mobile web browsers, regardless of whether it is Apple, Android, or another device. Still, there are use cases (for example, games, 3D modeling, and video editing) where native apps yield a much better user experience than even the most skilled HTML5 developer can craft, but those are seldom found in BPM projects. BPM solutions are heavy on complex forms, tables, reports, and charts. Major players in modern web development have proven that interfaces like this can be created with HTML5 to feel completely native (for example, Senchas HTML5 remake of Facebook, as a response to Facebook going native). Exceptional platforms like IBM Worklight® have taken this a step further, and made creating mobile web apps for BPM almost as easy as creating the desktop UIs in the BPM Coach Designer.

Going with a mobile web app, instead of a native one, lowers the cost for making BPM interfaces accessible from mobile devices, but it still requires an extra development effort in addition to the screens built in the coach designer. Development time needs to be allocated for building a different set of interfaces in a different tool. Sometimes this is the best solution because the functions available to mobile devices, within a given process app, are drastically different from the desktop interfaces. For example, if one BPM activity always needs to be completed on a Tablet (iPad or Android) and never on the desktop, building that mobile interface in a separate tool makes sense, because it would not be implemented in the Coach Designer. Still, most of mobile use cases that we see in BPM solutions today are interfaces that need to be available to both desktop and mobile users. Whether it is a type of request form, approval screen, or another activity in the process, users want to be able to complete those tasks on both desktop and mobile devices.

Responsive design was first introduced in 2010 in an '*A List Apart'* magazine article. Since then it has quickly caught on with some of the largest Internet companies today. It is an approach to web interface development where a single web page is implemented to display properly, regardless of screen size. Instead of building one web interface for phones, one for tablets, and a third for desktops, modern web design and CSS3 media-queries are used to create a single implementation that adapts to the device it is being viewed on.

Figure 6-1 is an IBM BPM Coach being viewed on a desktop web browser. It has been implemented using Brazos, the BPM UI toolkit from BP3, with just the BPM Process Designer.


Wind Farm

Submit Wind Study Plan for Review

Enter the Site, Team, Wharehouse, and Met Tower Information

Site Details

Start Date

Test Location

End Date

Met Tower Wharehouse

Location: USA

Units of Measure: Metric [✓ Imperial]

Met Towers for Wind Study

Config ID	Cost	Weight	Height
G0	$ 0.00	lb 0.00	ft 0.0
	$ 0.00	lb 0.00	ft 0.0

Met Tower Totals

Tower Sub Total: $

Transportation Cost: $

Final Cost to Client: $

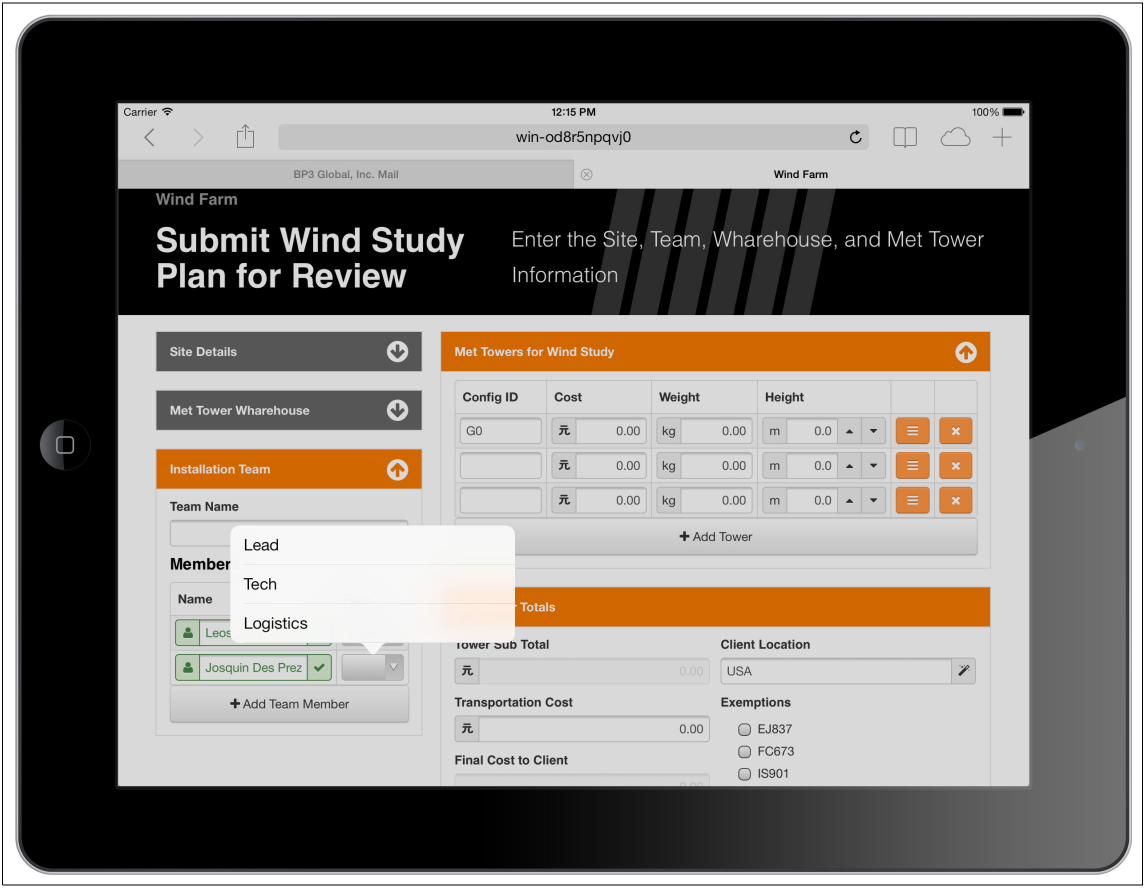
Client Location: USA

Exemptions

- ☐ EJ837
- ☐ FC673
- ☐ IS901

Figure 6-1 Wind Study Coach on a Desktop

If we load this same exact coach from Safari, on an iPad or iPhone, it will look like Figure 6-2 and Figure 6-3 on page 198.



Config ID	Cost	Weight	Height
G0	0.00	kg 0.00	m 0.0
	0.00	kg 0.00	m 0.0
	0.00	kg 0.00	m 0.0

Figure 6-2 Same Wind Study Coach on an iPad

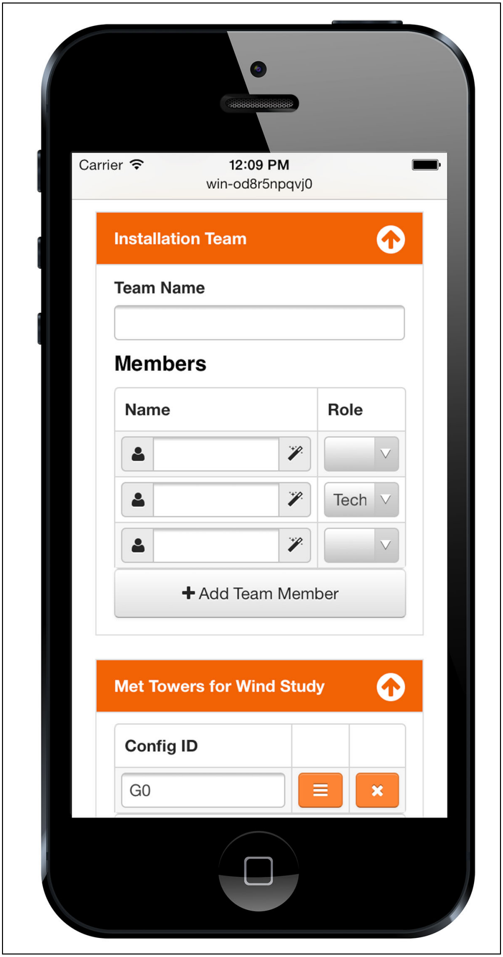


Figure 6-3 Same Wind Study Coach on an iPhone

Note that the interface adjusts to the screen size. Individual CoachView elements are built to resize, rearrange, and completely shift from one part of the interface to another based on the device. There is no need to waste time on pinch and zoom, the form is presented in the optimal format for the screen it is being viewed on. This is Responsive Design at work. With the addition of the CoachView framework to BPM in version 8.0, we can now incorporate modern HTML5 concepts and CSS3 media queries into the coach designer controls, making this possible.

6.2 The Coach View UI Toolkit

The majority of BP3's contribution to this Redbooks publication focuses on Brazos, our CoachView UI toolkit. The approach and features of this toolkit are a reflection of the demand we have seen from our clients, and our experiences using it on almost every IBM BPM 8+ project we work on.

The purpose of this section, however, is to address how to approach building your own CoachView UI Toolkit. Brazos is a great solution for most IBM BPM deployment needs, and if you can, we recommend using it rather than starting from scratch to build a new one.

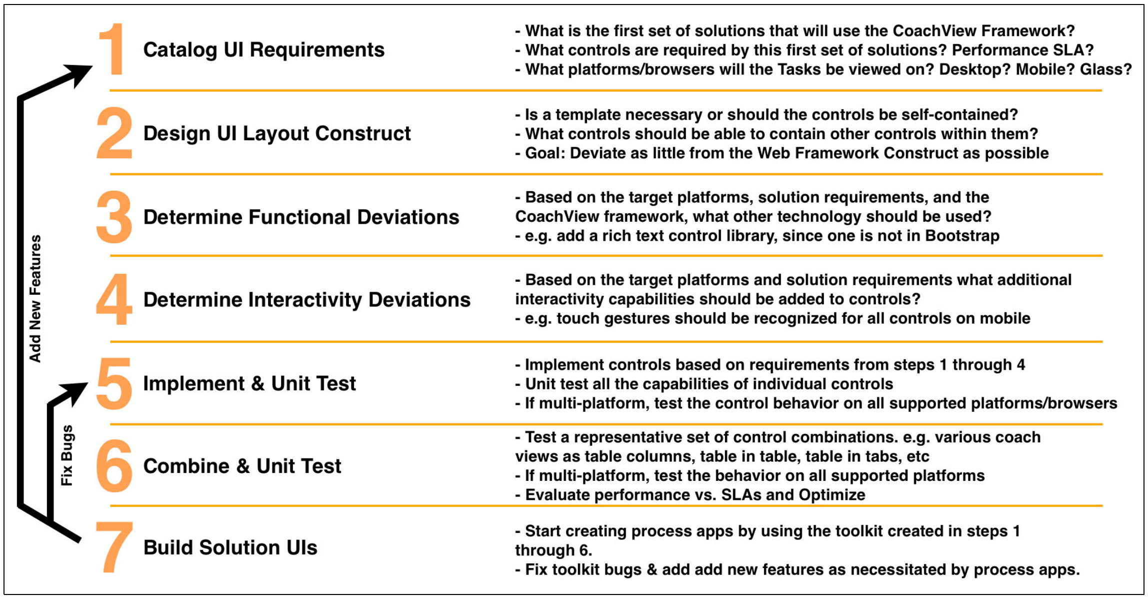
However, there are reasons to consider making such an investment. Since CoachViews were first introduced in BPM, we encountered multiple organizations that had already invested in a certain web UI framework in other parts of their organization before being introduced to IBM BPM. There are many exceptional web UI frameworks out there, thus organizations frequently pick one that is not Dojo (out-of-the-box BPM controls are written with Dojo) or Bootstrap (like Brazos).

For example, one company may already be using Kendo UI to build enterprise applications because of its great .NET connectors or another might have adopted Sencha due to the nice integrated development environment. In each case, the organization has altered the look and feel of the framework to fit their corporate identity, built up internal expertise, and accustomed its end users to the framework. With the power of CoachViews, IBM BPM has the unique capability to incorporate any modern web UI framework into Coach Views. By implementing UIs that an organization is already accustomed to, organizations can reduce friction for BPM adoption. Moreover, since process applications can be built much faster on BPM than many other development platforms, using a familiar UI framework will help the BPM footprint grow within the organization.

We found that an iterative approach, much like the one used to build process applications in BPM, works best when adapting a web UI framework to CoachViews and the Process Designer. A comprehensive web UI framework will contain dozens of controls with hundreds of configurable features; therefore, the implementation needs to be broken down in to well-planned iterations in order to keep the scope and budget manageable.

Although the specific requirements of the first few process applications to be built with the new UI framework are important, the first iteration of the framework implementation should focus on translating the layout paradigm of the web UI framework to CoachViews. The goal is to define a set of CoachViews that will allow a BPM developer to take advantage of the layout features offered by the web UI framework. For example, if it is a responsive framework, and sections rearrange based on screen size, the CoachViews should do the same, as BP3 did with the Brazos Toolkit. In addition to layout, the first iteration should include some basic input controls, with minimal configuration options.

Subsequent iterations are defined based on the web UI framework at hand as well as the intent for the CoachView framework being built. Each iteration will add another layer of capability, whether it is advanced input features, modular CoachViews, or mobile enablement. Within every iteration, a complete design, build, and QA cycle should be taken to ensure alignment with the overall designation for the framework and to retain consistent quality.



1 **Catalog UI Requirements**
- What is the first set of solutions that will use the CoachView Framework?
- What controls are required by this first set of solutions? Performance SLA?
- What platforms/browsers will the Tasks be viewed on? Desktop? Mobile? Glass?

2 **Design UI Layout Construct**
- Is a template necessary or should the controls be self-contained?
- What controls should be able to contain other controls within them?
- Goal: Deviate as little from the Web Framework Construct as possible

3 **Determine Functional Deviations**
- Based on the target platforms, solution requirements, and the CoachView framework, what other technology should be used?
- e.g. add a rich text control library, since one is not in Bootstrap

4 **Determine Interactivity Deviations**
- Based on the target platforms and solution requirements what additional interactivity capabilities should be added to controls?
- e.g. touch gestures should be recognized for all controls on mobile

5 **Implement & Unit Test**
- Implement controls based on requirements from steps 1 through 4
- Unit test all the capabilities of individual controls
- If multi-platform, test the control behavior on all supported platforms/browsers

6 **Combine & Unit Test**
- Test a representative set of control combinations. e.g. various coach views as table columns, table in table, table in tabs, etc
- If multi-platform, test the behavior on all supported platforms
- Evaluate performance vs. SLAs and Optimize

7 **Build Solution UIs**
- Start creating process apps by using the toolkit created in steps 1 through 6.
- Fix toolkit bugs & add add new features as necessitated by process apps.

Figure 6-4 Stages of an iteration

If done correctly, process applications can start using the UI toolkit as early as iteration 2 of the implementation. As new snapshots of the toolkit are created, process applications update their dependencies and begin using the newly available features without breaking anything that has already been built. Continuous automated unit testing of the UI framework ensures that the toolkit remains upgradeable from snapshot to snapshot as shown in Figure 6-4.

6.3 BP3's Brazos UI Toolkit

Brazos is a responsive IBM BPM UI toolkit created by BP3. The toolkit implementation adapts Bootstrap, one of the most popular web UI frameworks, to the IBM BPM CoachViews and serves as a complete replacement for the out-of-the-box controls as shown in Figure 6-5.

Figure 6-5 Different target platforms

BP3 has made the toolkit available for no charge to everyone through our website. Anyone can get the toolkit and use it for whatever purpose they want, from development to production. We found Brazos indispensable on our projects, and we think that the whole IBM BPM community benefits when processes have great user interfaces. We do offer several varieties of comprehensive support agreements. They range from being able to file Brazos bugs all the way through having BP3 build custom Brazos controls, on demand. Contact info@bp-3.com for more details.

As detailed in the previous section, we do not claim that Brazos is the one and only answer to UI in IBM BPM. The purpose of this section is not to convince you that you should use Brazos, but rather to present the reasoning behind why the toolkit exists as it does today and to convey the design choices we made along the way.

6.3.1 User experience matters

Even though business users have been conditioned to expect a less stimulating experience from enterprise software, it is by no means indicative of their desire. Better tools lead to a more productive workforce. Aesthetically pleasing web interfaces, which look like they were designed for the hardware that they are being consumed on and seemingly predict the intentions of their users, will always win supporters over a web form that looks like it was built in the mid-2000s. The goal for UIs in enterprise software today should not only be to meet business and functional requirements but also to exceed the user experience expectations of a generation that is accustomed to high-quality web interfaces like Google, Facebook, and Twitter. In a competitive marketplace for top talent, exceptional enterprise solutions will go a long way in improving productivity, morale, and retention of employees. Starting with the adaptation of Bootstrap, the intent behind Brazos is to bring the best of modern web design, which users are accustomed to in their private lives, to enterprise BPM solutions.

Still, taking a note from iOS application development, end user experience is only one piece of the puzzle. To ensure that a development community thrived around their platform, Apple put an overwhelming amount of time and money in to Xcode, the development environment used to build iOS applications. Even the minutest features have been tweaked extensively to optimize developer productivity and ease of use.

Even though the target audience for the BPM Process Designer is significantly smaller than Xcode, it in many ways follows the same pattern. BPM Developers are able to build all the artifacts that make up a process application within a single tool, toolkits organize reusable components, and snapshots make version management a breeze. Brazos builds on this foundation and attempts to make BPM UI Development as pleasant as putting together an iOS App. Brazos implements little things such as ensuring that all controls display correctly without having any data bound to them, supporting both instantiated and non-instantiated business data and property variables, and making sure that all controls are aligned aesthetically, regardless of the interface a developer puts together, to name a few.

These subtleties may seem inconsequential or “a matter of training”, but they add up. The less time a developer has to spend figuring out (or reading about) how to use a UI framework, the more time they have to actually build interfaces. Moreover, the development experience in Brazos has been thought out with the business analyst in mind. A non-technical user can prototype complex UIs, incorporate advanced features like typehead and modals, and of course craft both a mobile and desktop experience without needing to write any JavaScript, all within the Process Designer.

6.3.2 Business case

The primary driver behind the creation of Brazos was the need for IBM BPM process participants to complete their tasks from a mobile device. Before Brazos, this was accomplished by either having end users use desktop coaches on their mobile devices or by implementing a set of interfaces specifically for mobile devices in a different technology. Using desktop screens on a mobile device is an unpleasant user experience with an adverse impact on productivity. Not to mention, developing a set of mobile interfaces in addition to the desktop ones is a costly affair. Brazos delivers both a rich desktop experience, and a rich mobile experience, with a single implementation effort that is no more difficult than building a set of screens (just for the desktop) the old way.

6.3.3 The implementation

Let us examine a few implementation concepts.

- "Frameworks utilized"
- "Vector-only approach"
- "Native controls" on page 203
- "Touch versus click" on page 204

Frameworks utilized

Brazos incorporates Bootstrap, jQuery, and additional open source libraries like FontAwesome, Spinner.js, and so on. Most of BP3's time building Brazos has been spent on adapting the open source frameworks to the BPM use case, CoachViews, and touch-enabling all the components.

Vector-only approach

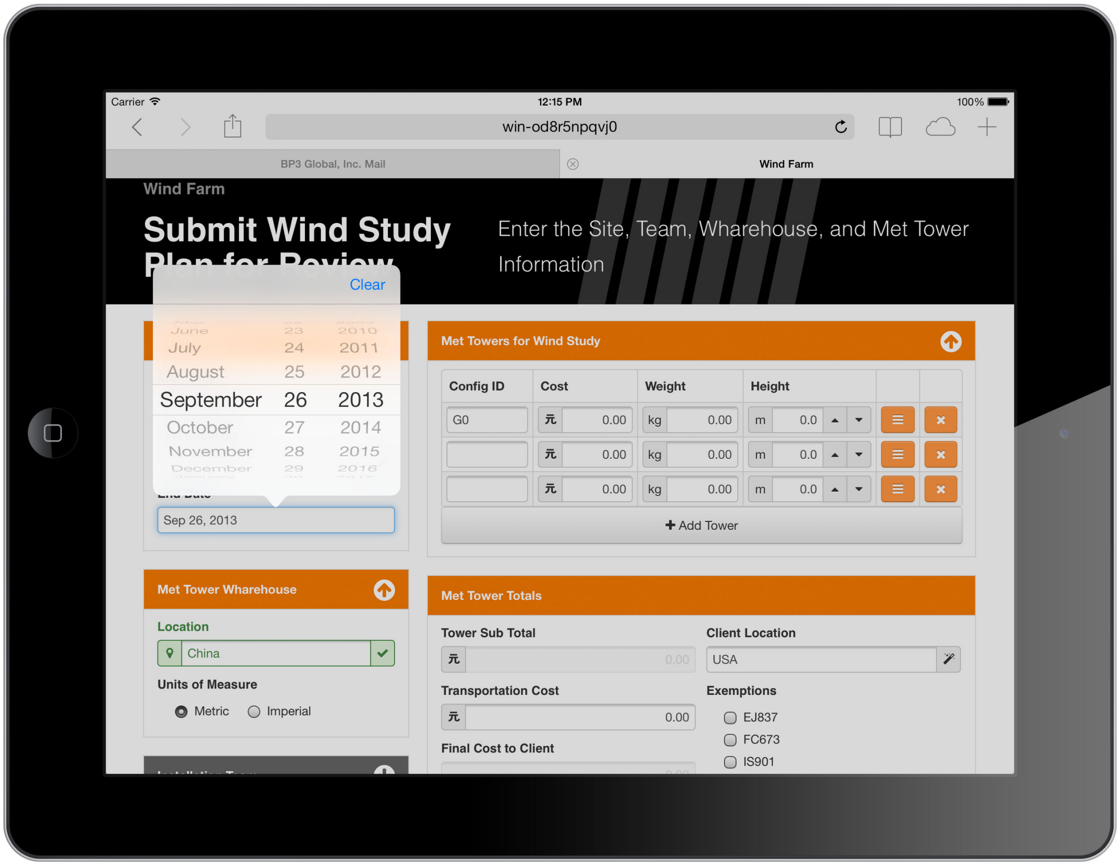
High pixel density interfaces were first introduced to consumers with the retina display on the iPhone 4. Since then, Apple has expanded the retina display footprint to iPads and MacBook Pros. Furthermore, other vendors have started building high pixel density displays for both mobile devices and desktop screens.

Web interfaces that use standard resolution, scalar, graphics look fine on traditional pixel density displays. When these are viewed on high pixel density displays, they seem washed out and blurry. Because of this, Brazos only uses vector graphics and fonts across all controls. Vector graphics look sharp, regardless of pixel density.

Native controls

Modern HTML5 browsers, like the ones that are found on iOS and Android, provide the ability for web pages to take advantage of certain native controls.

Actions like date and time selection, combination box selection, picture upload, and others can be performed much more effectively with these native controls since the device manufacturers have thought through the best way to present the user with say a date selection as shown in Figure 6-6.


Carrier 12:15 PM 100%

win-od8r5npqvj0

BP3 Global, Inc. Mail | Wind Farm

Wind Farm

Submit Wind Study Plan for Review

Enter the Site, Team, Wharehouse, and Met Tower Information

Clear

June	23	2010
July	24	2011
August	25	2012
September	26	2013
October	27	2014
November	28	2015
December	29	2016

End Date: Sep 26, 2013

Met Towers for Wind Study

Config ID	Cost	Weight	Height		
G0	元 0.00	kg 0.00	m 0.0		
	元 0.00	kg 0.00	m 0.0		
	元 0.00	kg 0.00	m 0.0		

+ Add Tower

Met Tower Wharehouse

Location: China

Units of Measure: [● Metric] ○ Imperial

Met Tower Totals

Tower Sub Total: 元 0.00

Client Location: USA

Transportation Cost: 元 0.00

Exemptions: ☐ EJ837 ☐ FC673 ☐ IS901

Final Cost to Client

Figure 6-6 Native iPad date picker from a Brazos control in Safari

In order for this to work correctly, each control must be implemented using the proper HTML5 types. Brazos uses the appropriate standard HTML5 types across all controls.

Touch versus click

On a desktop, interface elements like buttons, drop downs, and other interactive components wait for the *click* event to occur before performing their action. Desktop JavaScript controls like the out-of-the-box IBM BPM table, wait for a user to click the add table button before making an extra row appear. This works great. When a user clicks the button, the row instantly appears.

When a web page is being viewed on a mobile device, a click on the desktop does not translate to a finger tap on the touchscreen. In fact, the user will observe a 300ms delay between when they tap on the add row button (from the previous paragraph) and when the row will actually appear, making the interface feel clunky. This happens because the HTML5 specification includes touchscreen specific events. When an element is just waiting for a click, it waits for the following event sequence to occur: (touchstart, touchmove, touchend, mouseover, mousemove, mousedown, mouseup, CLICK). In order to avoid any such delay on mobile devices, controls need to be implemented to support touch events in addition to click events.

In order for Brazos controls to instantly respond to both click on the desktop and tap on mobile devices, each control has been built to support both event types. This extra work pays off every single time a user touches the interface, across every screen built with a Brazos control on it.

6.3.4 Layout and controls

As mentioned in prior sections, the first step in adapting a web development framework to the CoachView framework is to design the layout. The approach to lay out implementation and assembly within the coach designer may differ significantly based on the web development framework being adapted. Working with responsive web frameworks brings an extra layer of complexity. Not only should the person assembling an interface with the coach designer be able to create desktop screens, but they also need to define how the screen scales to a smaller screen size. As the screen size changes, layout components may rearrange themselves, sections may appear and disappear, and so on.

Brazos includes Bootstrap’s robust layout framework, which uses a row and column metaphor for layout design. A row takes up the full width of the screen. Each row can be broken down in to multiple columns. The relative width of columns within a given row is specified by adding a class value of span1 through span12. See Figure 6-7 on page 206.

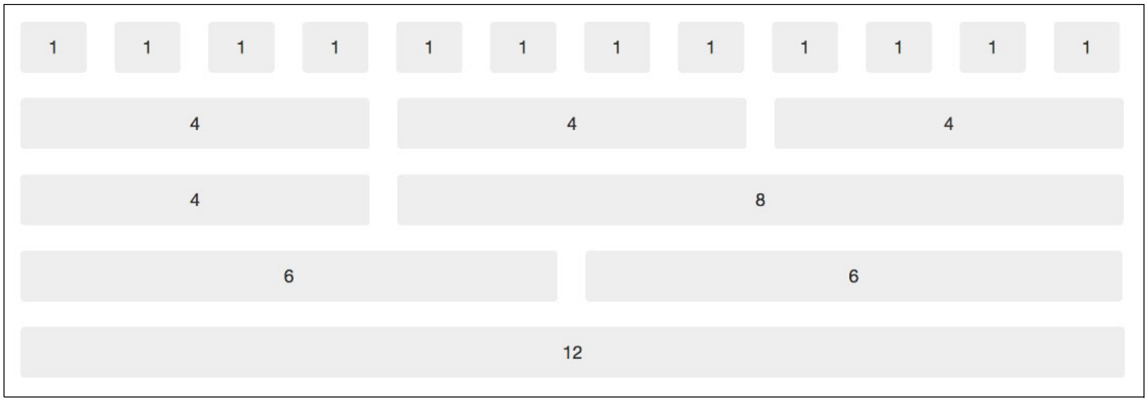
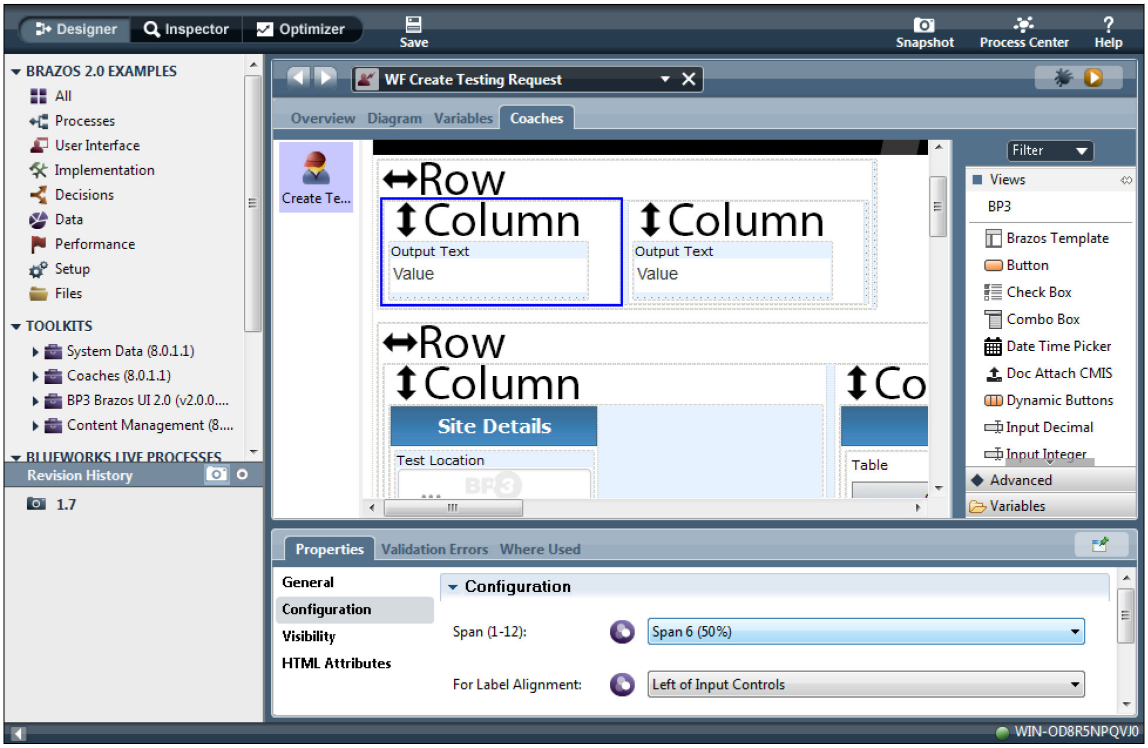


Figure 6-7 Span values and associated column width

As illustrated in Figure 6-7, a span of 12 is the full width of a given row, if two columns are added with a span of 6 into a row, each one will take up half the row, and so on. As the screen size displaying the interface changes, so does the size of the interface. On phones, where there is not enough room to display multiple columns in one row, the interface rearranges itself to display a single column per row, and adds as many rows as necessary to house all columns.

Figure 6-8 on page 207 shows column width configuration for Brazos columns in Process Designer.



Span (1-12): Span 6 (50%)

For Label Alignment: Left of Input Controls

Figure 6-8 Column width configuration(Span) for Brazos columns in Process Designer

In implementing Brazos, the same row and column metaphor was retained. A single “Responsive Row” CoachView represents the full width of the screen and can contain several “Responsive Column” CoachViews. The relative width of each column within a row is defined by using the span property of the Responsive Column as shown in Figure 6-8.

6.3.5 Example 1: Desktop, mobile, and beyond

The business case highlighted in this example is for a chemical manufacturing company. The requirement is to have a BPM interface for defining a custom chemical order. Once the order is defined and submitted, the actual manufacturing process begins. The screen needs to contain: A list of chemical ingredients, machines used throughout the manufacturing cycle, the associated man-hours, and basic order details. Additionally, the business requires the ability to scan SIDs of batches used for each chemical ingredient. The SIDs are found as bar codes on every drum of chemical ingredient.

Dynamic tables will be needed for both the chemical ingredients as well as the *Machine Usage*. Since one container of a given chemical ingredient may not be sufficient to complete one recipe, the user needs to be able to define multiple sources for a given chemical. From an interface perspective, this translates to a table (batches) within a table (chemicals). The batch table should have the ability to add batches and scan the associated SIDs (bar codes) of each batch with a mobile device.

Brazos is the right fit for this business case because it offers lightweight coach views that can be used to create complex interfaces that work great on either desktop or mobile. Even though mobile web browsers do not have the explicit ability to scan bar codes, Brazos can be used in a Hybrid App to accomplish this goal.

Implementation strategy

The first thing to consider is what devices the app will be used on and what functionality will be needed on each of those devices. Brazos uses a responsive UI framework, which supports interfaces from desktop to mobile. Even though a single Coach automatically adapts to various screen sizes, certain design decisions will be driven by the target devices. This makes knowing which devices a given interface is being built for an important consideration.

For this interface, most of the work will be done on a desktop, with a few steps on a mobile device. The business user would launch the task from a desktop, add overview details, add the chemical ingredients that make up the *Recipe*, and add the necessary *Machine Usage*. Once all the chemical ingredient rows are added, the user will pull the screen up on a mobile device and scan serial numbers (bar codes) for batches of the chemical ingredients, entering the associated amount used of each batch. The bar code scanning will be accomplished by having Brazos controls talk to a native shell application that supports that capability. This example will explain how it can be done on both Android and iOS.

Build-out

We can divide the build-out phase into the following sections:

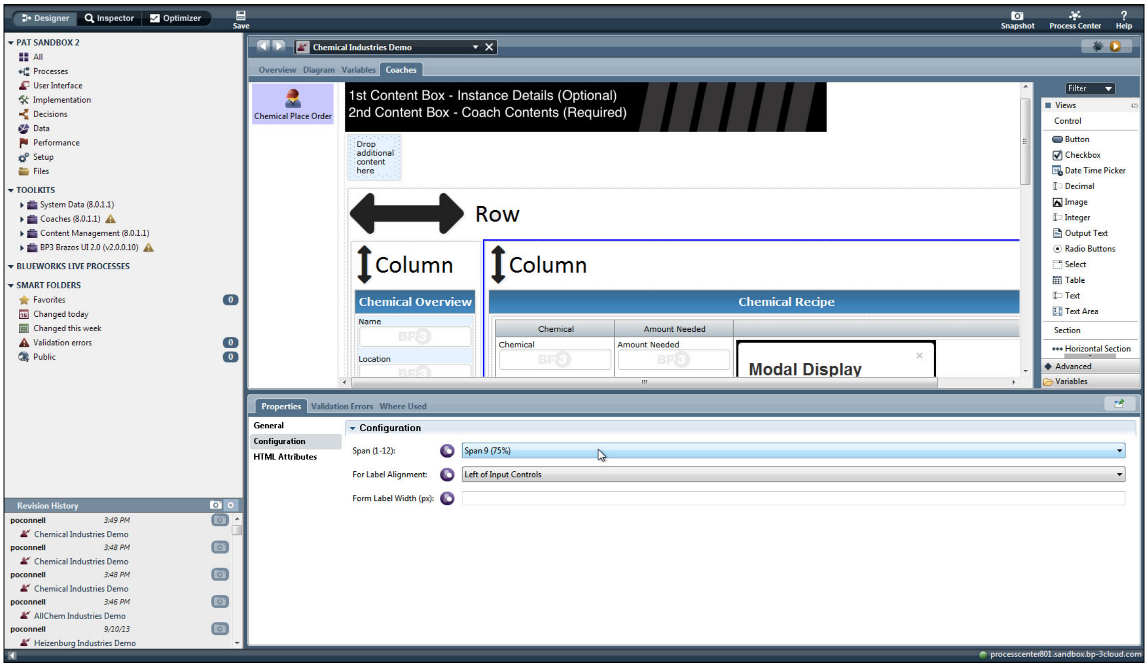
- Layout construct (Responsive Row, Responsive Column, Section, Tabs)
- Simple controls (Input Box, Drop Down, Check Box, Doc Upload, and so on)
- Advanced controls: Table
- Advanced controls: Modal

Layout construct (Responsive Row, Responsive Column, Section, Tabs)

Defining a good looking layout that meets the business requirements can sometimes be a daunting task. To simplify this, Brazos comes with a starter template that serves as the foundation of the responsive construct. The template has configuration options to customize its appearance with minimal effort. In this case, we use the Brazos template, populate the project and activity properties, and set a custom CSS. The CSS file is used to alter the appearance of any UI element, whether it is the background of the header, the size of the input box, or the style of the drop-down.

With the template in place, the layout is defined by using a combination of Responsive Row (Row) and Responsive Column (Column) CoachViews. First, a Row is added to the Brazos Template. The contents of the Row should only consist of Columns; the Columns then contain actual Brazos controls. Relative width of columns within a given row is defined by using the span configuration option on the Column. The Row will display its Columns in order, from left to right until the screen size shrinks enough to stack the Column contents vertically. This behavior is what lets the UI adjust to any screen from phone to desktop.

For this scenario, one row containing two columns is added to the template as shown in Figure 6-9 on page 210. The first column is for *Chemical Overview* details. Its span value is set to 25% of screen width. The second column, containing both the *Chemical Recipe* and *Machine Usage* tables, will have a span of 75%. Since the rows and columns are not visible screen elements, the Section coach view is used to logically group controls on a given interface. One section (labeled *Chemical Overview*) will be added to the left column and two sections (labeled *Chemical Recipe* and *Machine Usage)* will be placed in to the right column. Here is what it looks like in the Process Designer.


Span (1-12): Span 9 (75%)

For Label Alignment: Left of Input Controls

Form Label Width (px):

Figure 6-9 Setting the right column to take up 75% of screen width

The default behavior for mobile devices will be to stack all three sections one on top of the other, allowing the user to scroll through the content vertically. This is accomplished without any additional configuration.

Simple controls (Input Box, Drop Down, Check Box, Doc Upload, and so on)

After defining the layout, we can begin adding controls. The Input String coach view is a simple input box, with a variety of configuration options to cater to more complex business requirements. From this example, we take a look at how this control was used for *Location* under the *Chemical Overview* section. To implement the type-ahead feature, we first create an AJAX service that has a String List output. Once this service is created, within the configuration options on the Input String control, check the Typeahead Enabled option and bind your Ajax service to the Selection Service options. To allow for automatic validation of the control, check the Strict configuration option as shown in Figure 6-10 on page 211.

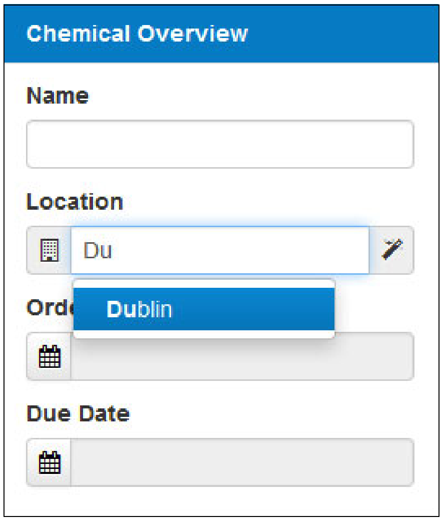
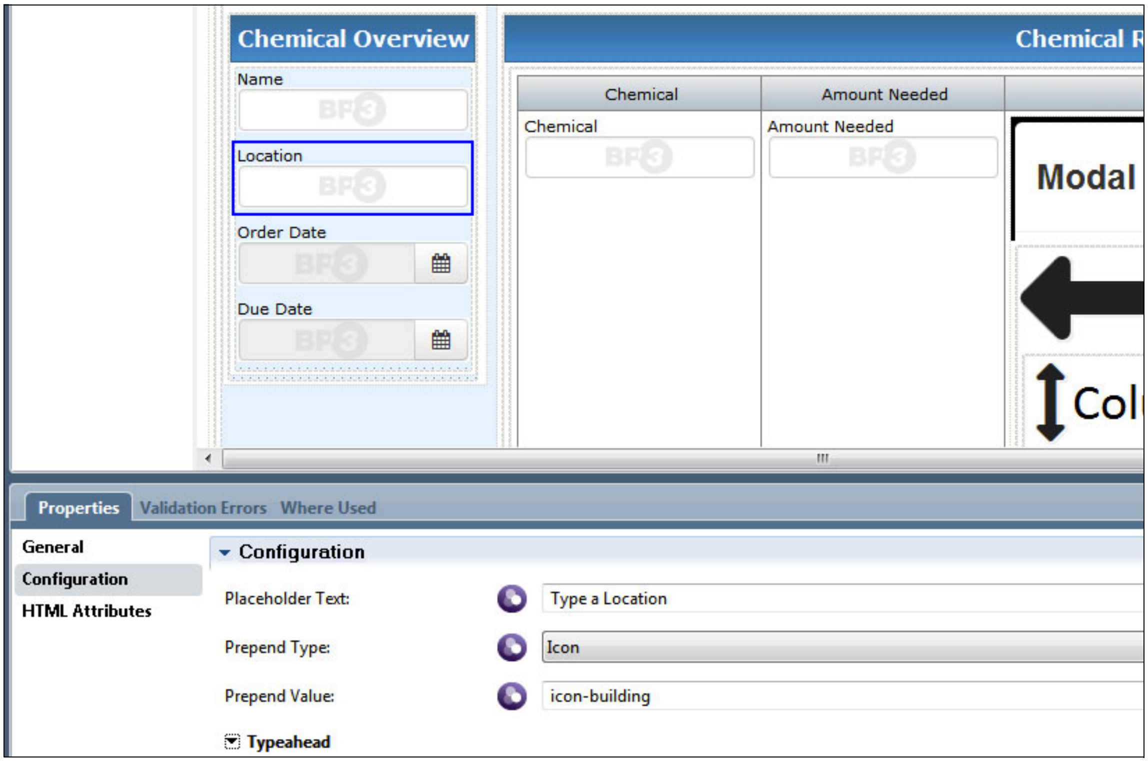


Figure 6-10   Type-ahead feature of the Input String coach view

Another configuration option that is used in this example is the prepend. In Figure 6-11 on page 212, you will notice that we have chosen the *Icon* Prepend Type, and have set the Prepend Value to icon-building. This will make a building icon appear on the left side of the input field.



Placeholder Text: Type a Location

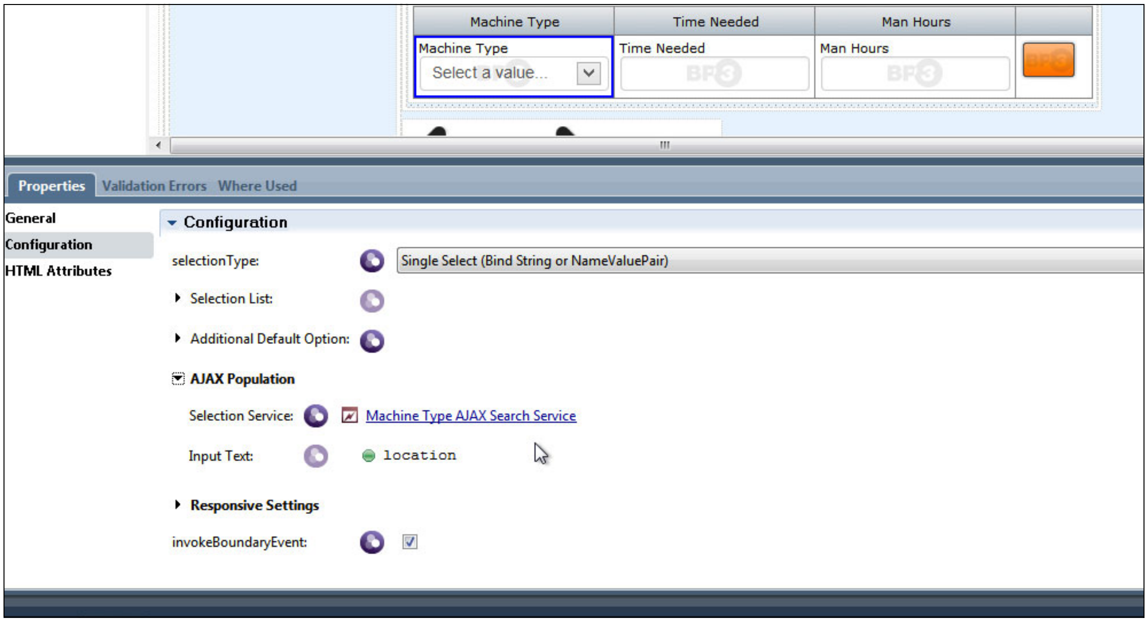
Prepend Type: Icon

Prepend Value: icon-building

Typeahead

Figure 6-11 Input field prepend option

The Combination Box coach view was used in the *Machine Usage* section of this example for selecting machine type as shown in Figure 6-12 on page 213. This control can bind a list of name-value pairs or query an Ajax lookup service to populate the selection options. We use the latter option. As with the type-ahead feature on the Input String, you need an Ajax service here. However, the service will have an output type of list of name-value pair and input of type String. When the Ajax service is complete, you bind it to the *Selection Service* option. For our input into the attached service, we will be binding location from the *Chemical Overview* section. Since each location has a different set of machine types to choose from, we use the location to create a dynamic list of name-value pairs that populate our selection options.



Configuration

selectionType: Single Select (Bind String or NameValuePair)

Selection List:

Additional Default Option:

AJAX Population

Selection Service: Machine Type AJAX Search Service

Input Text: location

Responsive Settings

invokeBoundaryEvent: ✓

Figure 6-12 Dropdown selection service

In addition to the more complex configuration options, most Brazos controls have an optional *invokeBoundaryEvent* configuration. When selected, a Boundary Event will fire when the contents of that control are changed by the user. One of the places we use this feature in this example, is to calculate Man Hours. The Invoke Boundary Event is enabled on both the Machine Type combination box Coach View as shown in Figure 6-13, as well as the Time Needed Input String. When either of these values are changed, a Boundary Event will route to a Server Script that contains code to calculate the associated Man Hours.

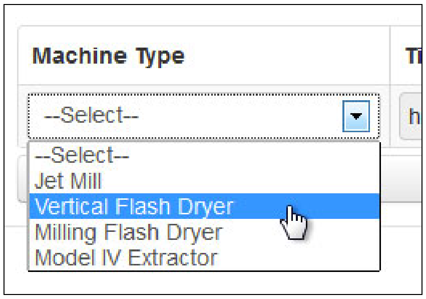


Figure 6-13 Combination box at run time

Advanced controls: Table

The Table Coach View is used to display a list of complex objects. Each table column can consist of any Brazos control-like combination box, input string, output text, modal (discussed in the next section), and even other tables. The Table control is simple to use and has minimal configuration options. It functions much like the out-of-the-box table.

Using the *Machine Usage* table as the example, we drag the Table control into the appropriate section and bind it to the list of machine objects. To add columns, drag other controls from the pallet into the table. The first column is the *Machine Type* combination box, which we covered in the section before this. The second and third columns are simple Input Decimal Coach Views that bind to the *TimeNeeded* and M*anHours* attribute of the machine object list.

The final column needs to be a row delete button. To accomplish this, we add a Button control from the Brazos toolkit and set the table behavior (of the button) to “delete row”. Enabling the *Add Row* button for the Table can be done by checking the *Allow Row Addition* in the configuration options of the table.

Advanced controls: Modal

A unique feature available in Brazos is the Modal Coach View. A modal is a great way to initially hide a set of controls from the user. It is essentially a miniature coach that pops up over the rest of the content when a button is clicked. Although a modal can be used outside of a table, it is most commonly found in tables as shown in Figure 6-14.

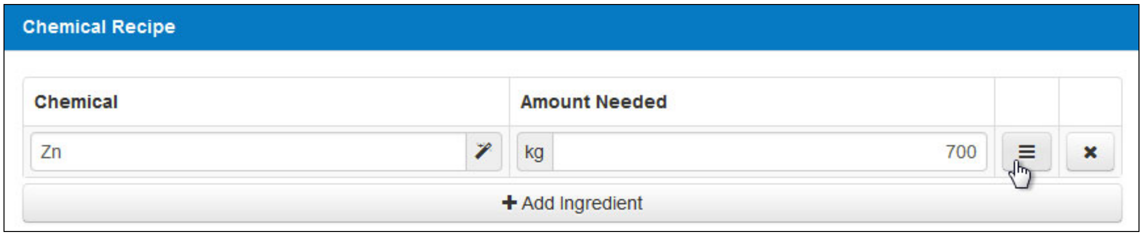
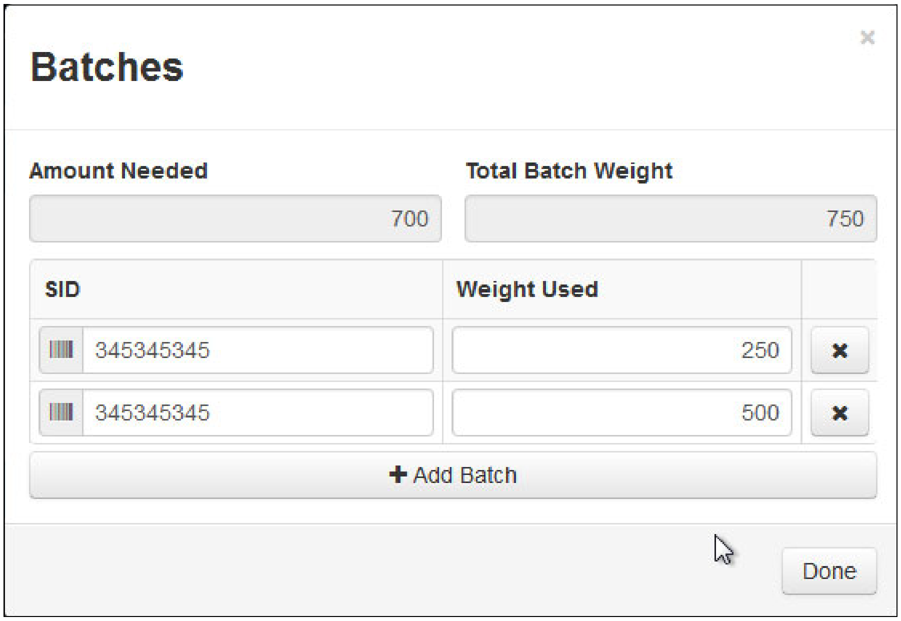


Figure 6-14 Modal on-screen load (as a table column button)

A Modal is a great fit for this scenario because we want to initially hide a few columns from the user as well as a nested table. To implement, the Modal control is first dragged into the *Chemical Ingredients* table as a new column; when the coach first loads, it will look like a button. The Button contents and the modal title can be modified in the configuration options. Once the modal is in place, you can add any control necessary inside of it. In addition to a few fields, this Modal contains a table of batches as shown in Figure 6-15 on page 215.

The *Weight Used* column has an associated Boundary Event that causes *Total Batch Weight to* update *any time a change is made to the table*. The SID column of the batch table is populated by using a bar code module in a Hybrid shell described in the next section.


Batches

Amount Needed	Total Batch Weight
700	750

SID	Weight Used	
345345345	250	×
345345345	500	×

Figure 6-15 Modal when the button on a given table row is clicked

Hybrid application

If a mobile device capability needs to be used as part of a BPM UI, but it is unavailable to the web browser, going with a hybrid approach may be the best path forward. In a hybrid application, coaches built with Brazos run in a native mobile device shell. This allows you to build all interfaces with Brazos in the coach designer, needing only to implement the specific missing feature in native code.

For this example, we need to be able to scan a bar code and have the value populated in to a Brazos control. While the mobile web browser can access the device camera, it does not have a built-in API to interpret bar codes. Implementing the image processing capability in client-side JavaScript would be tedious and most likely not perform well, but there are numerous bar code scanning libraries for both Android and iOS. The simplest solution, in this case, is to build the interface with Brazos, and create a CoachView control that can communicate with a native app shell. That native app shell would then open a bar code scanner, scan the bar code, and populate the Brazos control with the bar code value as shown in Figure 6-16 on page 216.

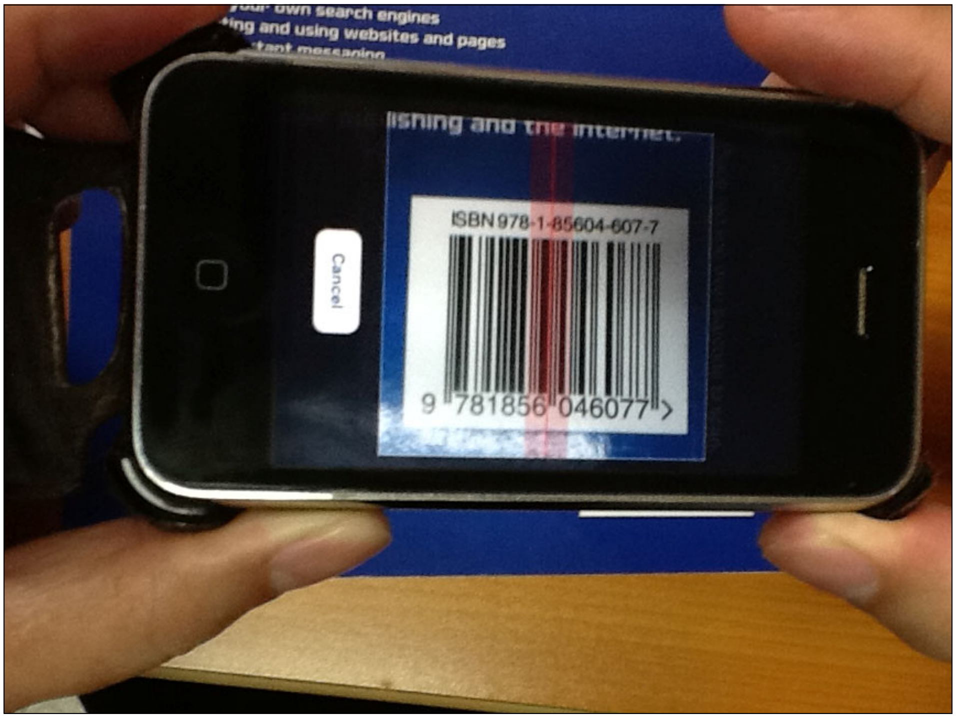


Figure 6-16 Bar code scanner module from a hybrid shell

Bar code Scanner CoachView

As a baseline for the control, you can duplicate the String Input from Brazos, since the bar code scanner looks the same, except instead of letting a user type in a value, it invokes the bar code scanner. You need to modify the onClick event on the control to communicate with the native shell and open the bar code scanner.

For iOS support: An invisible iframe needs to be created somewhere in the Document Object Model (DOM). The target for that iframe will be set to something similar to: ‘bp3brazos://myFunc/divID/attr1/attr2’. The native shell listens for that bp3brazos prefix to start the scanner.

For Android support: All you need to do is simply call: Android.myFunc(divID,attr1,attr2). Android web views create a javascript class on loaded web pages. This makes communicating between CoachViews and the native shell a breeze.

iOS Shell App (Objective C)

The object type that is used to display the coach is a UIWebView. Its delegate needs to implement the shouldStartLoadWithRequest method. This is invoked when an iframe attempts to load the fake URL set up in the previous section. We can capture the attempt based on the bp3brazos prefix and treat it as the command to start the bar code scanner.

Once the bar code is scanned, stringByEvaluatingJavaScriptFromString can be called on the UIWebView, setting the input field of the bar code scanner coach view to the bar code value.

Android Shell App (Java)

The Android WebView class is used in this case. To make the Android JavaScript class available to the CoachViews, we first need to add:

```
webView.addJavascriptInterface(new WebAppInterface(this), "Android");
```

From there, you simply implement myFunc(divID,attr1,attr2) in your class. That function will get called when the user clicks the bar code scanner control. The logic for starting the bar code scanner resides within said function.

After the bar code is scanned, webView.loadUrl("javascript:..."); is used to set the bar code scanner Coach View to the bar code value.

Validation

With the release of BPM version 8.0.1, IBM has provided an out-of-the-box validation framework. The framework gives the option to test a set of variables for validity with a server-side script, when a user triggers a Boundary Event (for example, clicks a button). If the validation script adds errors using the out-of-the-box API, the token will route back to the coach and display the validation errors. If the validation script completes without errors, the original Boundary Event will proceed forward as shown in Figure 6-17 on page 218.

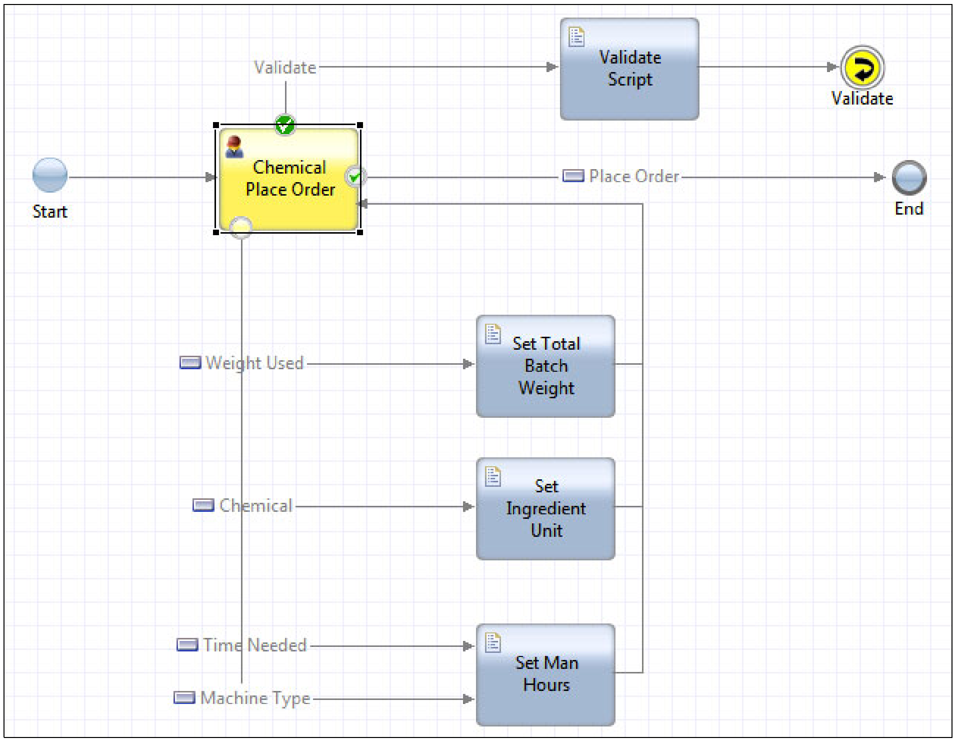
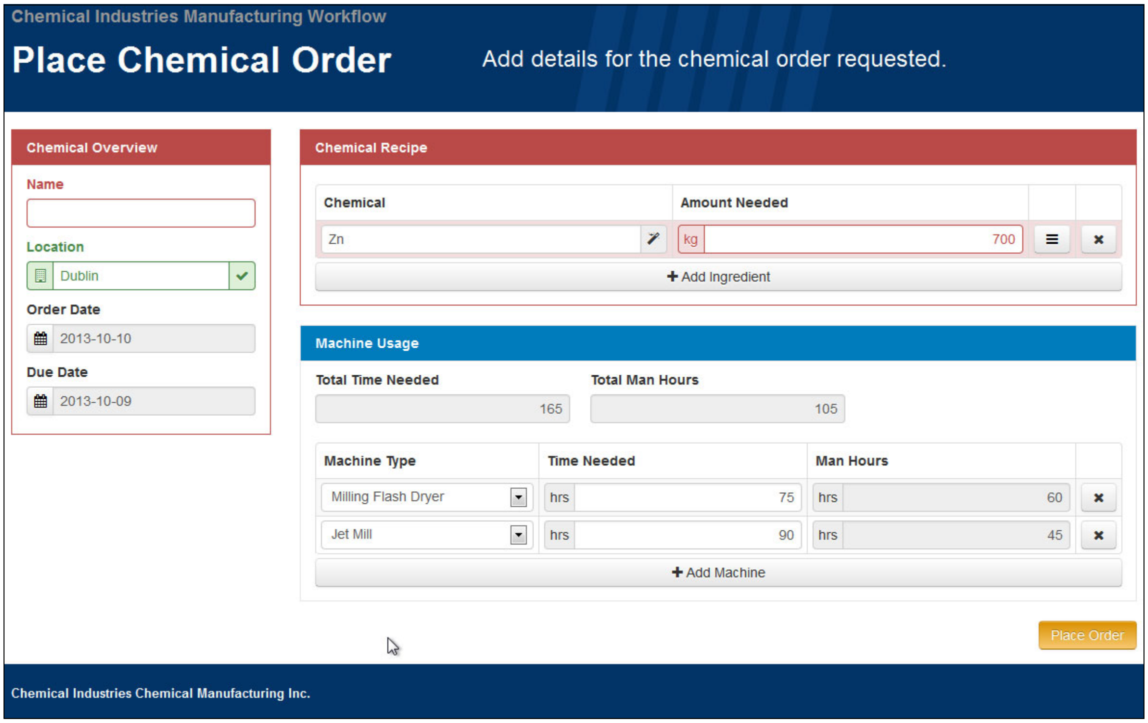


Figure 6-17 OOB Coach Validation setup

Brazos supports the out-of-the-box validation framework by identifying erroneous fields and displaying their associated errors. Additionally, Brazos adds a unique touch from a UI perspective. Any collapsed sections slide open and get highlighted in red. The validation errors for a given field can be read by hovering over the field that is bound to the erroneous variable as shown in Figure 6-18 on page 219.


Chemical Industries Manufacturing Workflow

Place Chemical Order

Add details for the chemical order requested.

Chemical Overview

Name:

Location: Dublin

Order Date: 2013-10-10

Due Date: 2013-10-09

Chemical Recipe

Chemical	Amount Needed
Zn	kg 700

Machine Usage

Total Time Needed: 165

Total Man Hours: 105

Machine Type	Time Needed	Man Hours
Milling Flash Dryer	hrs 75	hrs 60
Jet Mill	hrs 90	hrs 45

Chemical Industries Chemical Manufacturing Inc.

Figure 6-18 Result of OOB validation framework errors

6.3.6 Example 2: Scalability and program success

The example in this scenario is a complex user interface used for underwriting a loan for the purchase of a home or property. Underwriting a loan is a complicated, and sometimes time consuming process. It involves diving into personal documents for research, performing calculations to ensure financial security, reviewing properties and appraisals, and even performing background checks. This process varies from person to person. Some underwriters have been doing the job for more than 15 years and have a unique approach that works for them and helps them perform their analysis efficiently. The user interface in this business case was designed to be an aid in that process: a fluid, easy to use design that any underwriter could adapt their unique approach to.

The complexity of this interface led to the need to control the layout to a very precise degree. The controls provided by Brazos are the perfect fit because:

1. Defining custom form layouts with variable control sizes is simple and requires no technical expertise. For example, the form in Figure 6-19 on page 220 can be created in under a minute, using eight Input String controls and one combination box.

2. The controls offered by Brazos are lightweight: minimal code to provide the needed functionality.
3. Nesting tables, responsive sections, and modal windows are simple and consistent.

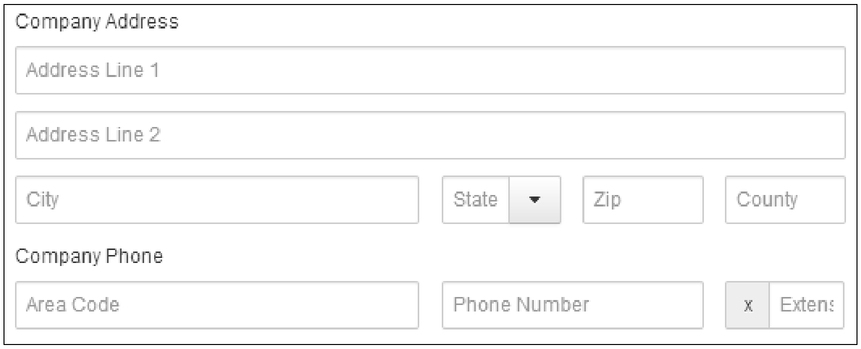


Figure 6-19 Form layout with variable control widths

Toolkit management

Even when starting with Brazos, BP3 consultants usually arrive at a need to develop other controls while on a client project. When this need arises, there might be the temptation to go into Brazos and start modifying it to add additional controls. However, this should be avoided. BP3 releases periodic updates and not modifying the actual Brazos toolkit will ensure it can be upgraded to future versions.

BP3's suggested approach is to create a separate toolkit with client-specific controls based on Brazos. This toolkit does not need to include co-dependencies with the Brazos toolkit, because all of your Brazos core resources are loaded when you use the Brazos Template. Considering that the dependency loading is done by the template, each control should only contain the incremental code to make it functional.

In your main application, you will then have two dependencies: One to the Brazos toolkit, and one to the client-specific toolkit. When a Brazos update comes out or updates to the client toolkit are made, upgrading of the dependencies only needs to occur in one place, the Process Application. You do not need to worry about cross-dependencies between toolkits.

When designing separate artifacts that rely on Brazos resources, be sure to pay attention to the release documentation with new versions of Brazos. In the rare case that a piece of code that you are relying on is deprecated, you might be required to update your own controls accordingly.

Designing a Complex UI

Starting from the bottom

Usually requirements for an entire user interface are not available at once. During the design and development phase of a complicated UI, more granularity is uncovered with time, which often leads to rework. Fortunately, if designed correctly, individual components of a Brazos UI can be designed from the bottom up, scaling as needed.

Responsive Rows and Responsive Columns work at any level of nesting. A Responsive Column that is set to be "Span 6 (50%)" will always be half the width of its parent (a Row) regardless of where it is in the coach. Combining this feature with the reusability of Coach Views, we can begin designing pieces of UIs even before there is a sense of what the overall structure of the coach will be. Consider the following example. We know that on an underwriting worksheet we are going to need an address field in a few different places. We do not know where it will be on the layout, but we know the data model and the required fields, and if we adhere to the Brazos layout paradigm, we can design a UI that will fit in to virtually any future scenario. See Figure 6-20 for an address Coach View.

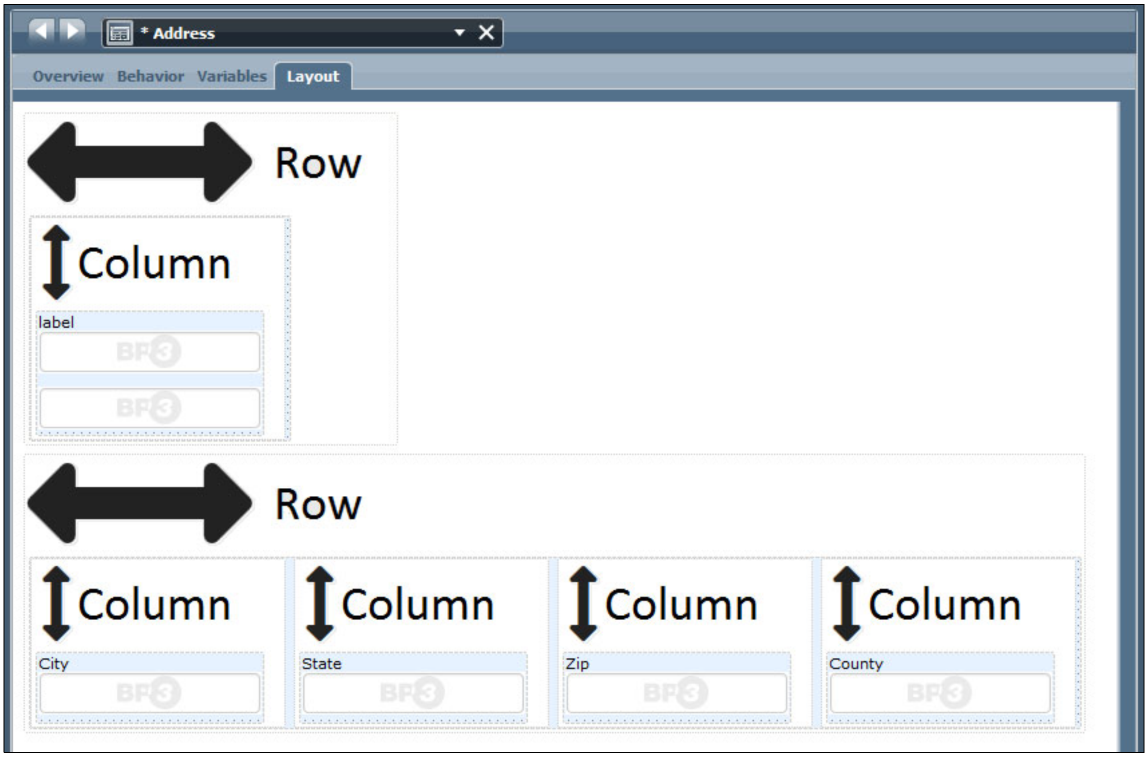


Figure 6-20 An address Coach View

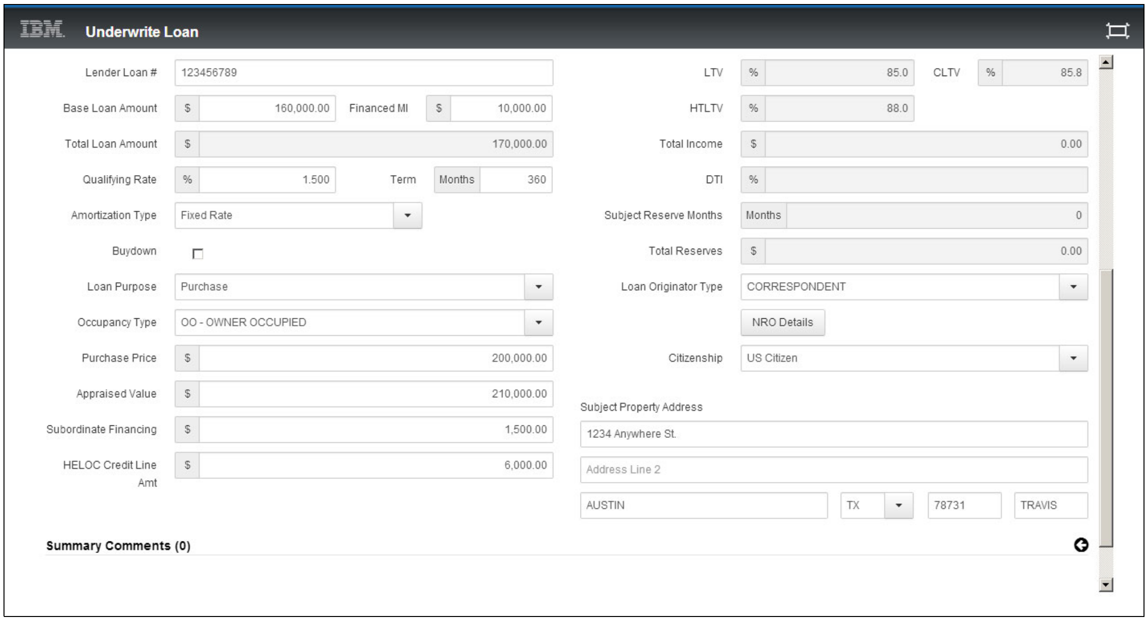
In Figure 6-20 on page 221, we see the implementation for the address from Figure 6-18 on page 219. The design is straightforward. No code is needed in any of the event handlers or CSS to accomplish this. To build the UI, we drag and drop the components in the pictured format, and adjust the configuration options as wanted. In this case, we need to configure the relative widths of the columns. The top column with the two address lines is Span 12 (100% width), while the bottom columns are Span 6, Span 2, Span 2, Span 2 from left to right (remember that a row's columns should add up to 12).

In just a few minutes, we have created a view that can be used in a multitude of scenarios.

Working our way up

Now, when we go to implement the larger UI, we can use the Address Coach View that we just created by simply dragging it in to its parent. When designing complex UIs, it is extremely important to break down components of the UI into smaller more manageable Coach Views. It is easier to work with 10 Coach Views with 10 controls that logically go together rather than a coach with 100 controls on it. With the process outlined above, your individual components do not have to rely on their parent coaches.

In Figure 6-21 and Figure 6-22 on page 223, you can see the composite Coach View from the above example being used for two different purposes.


Underwrite Loan

Lender Loan #	123456789
Base Loan Amount	$ 160,000.00
Financed MI	$ 10,000.00
Total Loan Amount	$ 170,000.00
Qualifying Rate	% 1.500
Term	Months 360
Amortization Type	Fixed Rate
Buydown	☐
Loan Purpose	Purchase
Occupancy Type	OO - OWNER OCCUPIED
Purchase Price	$ 200,000.00
Appraised Value	$ 210,000.00
Subordinate Financing	$ 1,500.00
HELOC Credit Line Amt	$ 6,000.00
LTV	% 85.0
CLTV	% 85.8
HTLTV	% 88.0
Total Income	$ 0.00
DTI	%
Subject Reserve Months	Months 0
Total Reserves	$ 0.00
Loan Originator Type	CORRESPONDENT
Citizenship	US Citizen

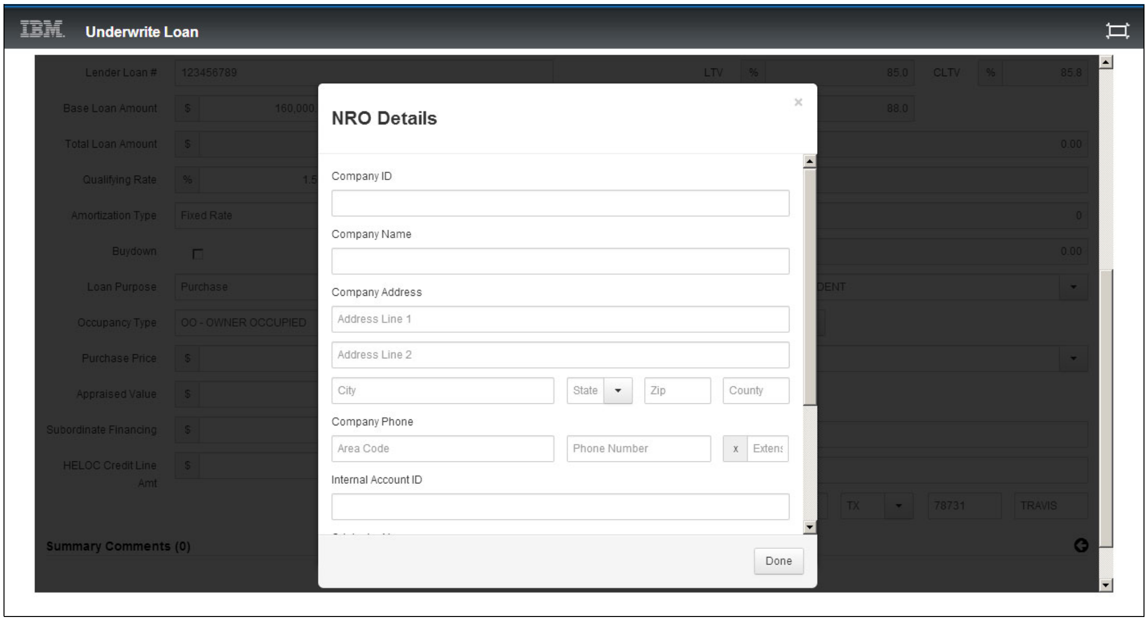
NRO Details

Subject Property Address
1234 Anywhere St.
Address Line 2
AUSTIN | TX | 78731 | TRAVIS

Summary Comments (0)

Figure 6-21 Address coach view usage: Example 1

In Figure 6-21 on page 222 the address Coach View is being used in the lower right corner of the screen. It is mapped to a business object called “subject property address” and is placed inside a responsive column that happens to be Span 6, or one half the width of the entire coach screen. Note that the child controls adhere to the relative column widths within the space available to the parent Coach View. Our city field was designed to be Span 6 (50%) of the Address Coach View. So when we place the entire Address inside a Span 6 column, the city naturally becomes 25% the width of the entire coach without any extra work on our part.


Underwrite Loan

Lender Loan # 123456789 LTV % 85.0 CLTV % 85.8
Base Loan Amount $ 160,000 88.0
Total Loan Amount $ 0.00
Qualifying Rate % 1.5
Amortization Type Fixed Rate 0
Buydown 0.00
Loan Purpose Purchase DENT
Occupancy Type OO - OWNER OCCUPIED
Purchase Price $
Appraised Value $
Subordinate Financing $
HELOC Credit Line Amt $
TX 78731 TRAVIS
Summary Comments (0)

NRO Details

Company ID
Company Name
Company Address
Address Line 1
Address Line 2
City State Zip County
Company Phone
Area Code Phone Number x Extens
Internal Account ID
Done

Figure 6-22   Address coach view usage: Example 2

In Figure 6-22, we can see the same view being used for company contact details as for the loan originator before it. This time, it is in a modal view. Again, with no extra work, we can drop in the control and it will naturally adjust to the size of its parent.

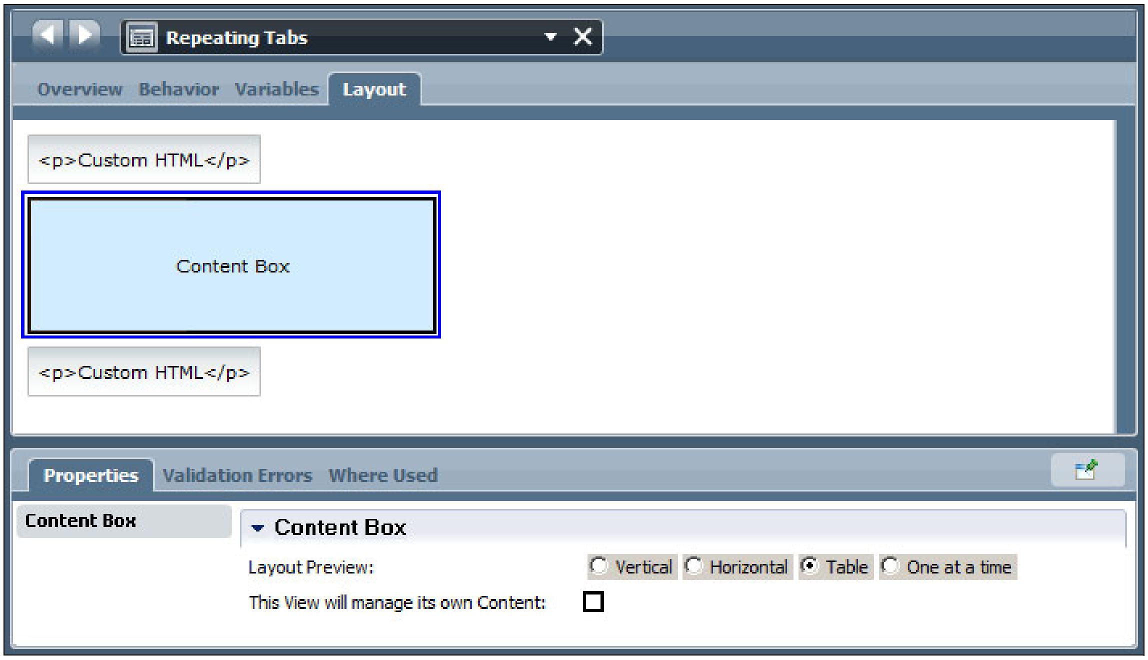
Implementing complex UI elements

Developing client-specific controls

Even though Brazos offers various features, we frequently encounter the need for business-specific controls at our clients. This section covers the general approach to developing controls that are compatible with Brazos. This approach is being used by BP3 in situations where additional controls need to be developed.

Consider the following example: While underwriting a loan, we need to capture information about one or more borrowers whose income is being considered for eligibility. At first glance, this seems like a good opportunity to use a table since we have a list of elements. However, since each borrower is associated with a large amount of information (name, address, SSN, monthly income, credit information, and so on) that needs to be captured, a table would not be ideal. A good alternative would be a “dynamic tabs” control: a control that behaves very similar to the table control, but each row is represented by a tab instead of a table row.

First, consider the HTML markup to construct the Tabs interface. Figure 6-23 shows the layout for the control. Notice that the content box is selected to be a “Table”. It will show up like a table control in our designer because we need the repeating functionality, but it will render like tabs.



<p>Custom HTML</p>

Content Box

<p>Custom HTML</p>

Content Box

Content Box

Layout Preview: ( ) Vertical ( ) Horizontal (•) Table ( ) One at a time

This View will manage its own Content: ☐

Figure 6-23 Repeating (Dynamic)Tabs template

The first Custom HTML box will contain:

```
<div>
      <ul class="nav nav-tabs"></ul>
      <div class="tab-content">
```

The <ul> will contain individual tabs that the user can click, and the <div> following the <ul></ul> will contain the actual content of the tabs. Notice the classes on each of the elements. These are Bootstrap classes. When creating custom controls to be used with Brazos, it is encouraged to only use either Bootstrap components or components that were specifically implemented to work with Bootstrap. This will speed up the Coach View build process and help make the final interface look and behave consistently.

At this point, we move on to the event handlers of the Coach View. Consider the following code shown in Example 6-1, which sets up the clickable tab buttons and our tab panes. This code belongs in the load() event handler for this control.

Example 6-1 Set up clickable tab buttons and tab panes

```
var _el = this.context.element
var sections = $("> div > .tab-content > .ContentBox", _el);

sections.addClass("tab-pane");
     each(function(index, section) {
      var href = "#"+$(section).attr("id");
      var title = "Tab Title";

      var tabButton = $("<li><a
href='"+href+"'><span>"+title+"</span></a></li>");

      $("a", tabButton).click(function (e) {
         e.preventDefault();
         $(this).tab('show');
      });

      $("> div > ul", _el).append(tabButton);
});
```

Start with line 2. This is a CSS query that returns the list of all sections that are children of the content box we outlined in Figure 6-23 on page 224. Coach Views that are children of other Coach Views are loaded before the parent, and constructed with various helper classes and attributes to aid in referencing them. The query in line 2 returns all “rows” in our table. The remainder of the code turns these table rows into tabs. This query is a jQuery function, and since jQuery is included with Brazos, we can use it in any controls that are children of the Brazos Template.

Next, we highlight some of the Brazos utilities that we are using to construct the rest of the control. The fourth line, `sections.addClass("tab-pane")`, adds a utility class to every “row” in our table. This is a Bootstrap class that helps with the hiding and showing of tab panes when we click the tab buttons.

The remainder of the code sets up the HTML for the clickable tab buttons. The use of Coach View attributes (such as `$(section).attr("id")`) coupled with Bootstrap styles and functions (`$("a").tab('show')` is a Bootstrap function to activate a tab pane that is based on a tab button).

There are many opportunities to expand this control beyond its basic functionality (such as adding and deleting tabs dynamically), but it is remarkable that after writing under 10 lines of HTML and a dozen lines of JavaScript, we have a fully functioning control that leverages the power of the Brazos libraries, and the out-of-the-box functionality of IBM BPM Coach Views. Taking advantage of Bootstrap, Brazos, and CoachViews, as illustrated in this example, simplifies creating new BPM controls.

Performance testing

CoachView UIs rely heavily on modern web technologies, including CSS3 and HTML5. As such, modern browsers are recommended when using Brazos. Any recent version of Firefox, Chrome, and Safari are all good options. Although Brazos will work with Internet Explorer 8, IE9+ is strongly suggested because the JavaScript engine in IE8 is dramatically slower and HTM5 tags are not supported. In fact, any Coach View-based IBM BPM user interfaces will perform considerably slower on IE8 because of the poor JavaScript engine.

With any complex UIs (Brazos or otherwise), it is important to make sure they perform well in a given client's setup. Since UI rendering, usage, and event handling are all mainly done in-browser, performance testing must be done in the context of an organization's supported browser, work locations, and ideally on various systems.

Even though Brazos was specifically designed to contain less code than the out-of-the-box BPM controls in order to improve the performance of complex UIs, the simplicity of UI development provided by the CoachView framework comes at a cost. BPM UIs will always require more HTML/JavaScript/CSS and thus load slower than custom built web applications. Still, performance problems can easily be avoided, even for the most complex UIs, by defining detailed performance SLAs early, and continuously testing the application from actual end-user work locations throughout the build process.

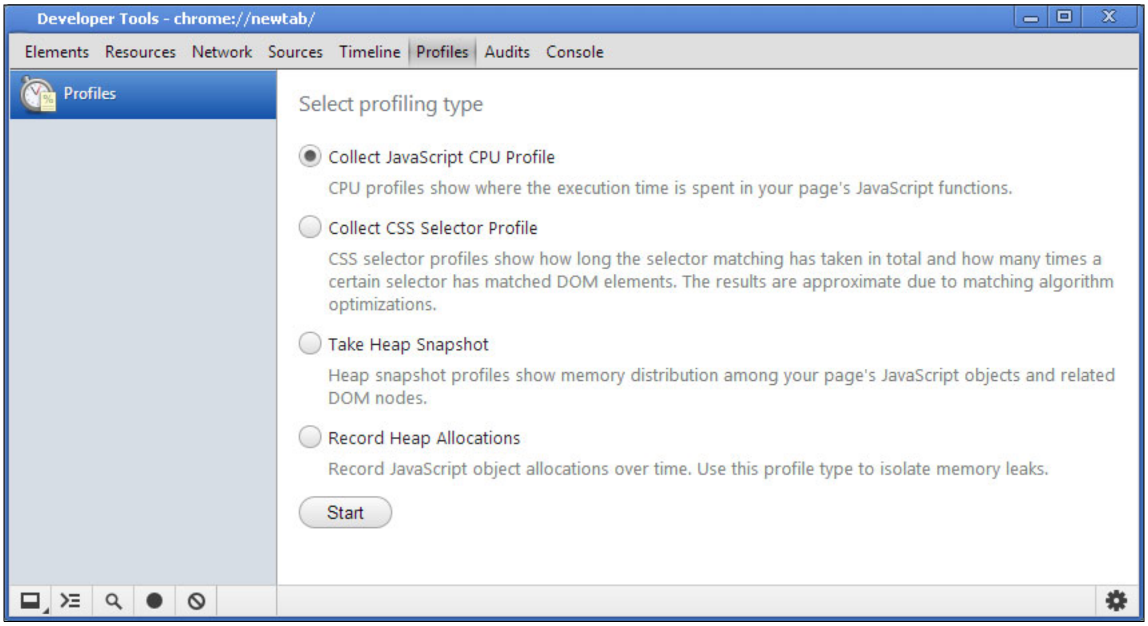
In order to ensure that CoachView interfaces perform well, consider the following two main concepts:

1. During Coach Load: Coach rendering has changed extensively with the introduction of Coach Views. In previous versions of IBM BPM, the server fed an HTML page to the browser and the browser merely had to render the markup. In IBM BPM 8.0 and greater, the browser is now responsible for more. The server defines the business objects and the base Coach View templates, and the browser is responsible for instantiating each Coach View

individually. This means that every Coach View on the screen has its load() event called before the entire page is rendered. This can lead to long load times for complex UIs. If there are 100 controls on a page, there are *at least* 100 load() events executed before the entire coach is rendered to the end user. That being said, Brazos controls were designed to be lightweight, and any modern browser should be able to easily render 100 controls on a page.

2. After Coach Load: After your coach has been rendered, there are quite a few things that happen behind the scenes as an end user is filling out forms and performing actions. It is important to keep in mind that whenever data is changed, a sequence of events is fired that updates the data model. If you have high levels of interactivity between different pieces of data, coaches can slow down as event handlers are triggered. Luckily, because the local data model is updated every time a control value changes, when it comes time to submit a coach or save the data, all the data can be posted to the server immediately because the browser has the most up-to-date copy of all variables.

In the event of performance issues, there are various analysis tools that can help you. Even if a coach does not have performance issues, it is important to run a few coach profiles to identify potential pain points or bottlenecks. Firefox, Chrome, and Safari all have tools that can analyze performance over the typical usage period of a coach. Figure 6-24 shows some of the performance tools offered with Chrome.



Select profiling type

[✓ Collect JavaScript CPU Profile]
CPU profiles show where the execution time is spent in your page's JavaScript functions.

Collect CSS Selector Profile
CSS selector profiles show how long the selector matching has taken in total and how many times a certain selector has matched DOM elements. The results are approximate due to matching algorithm optimizations.

Take Heap Snapshot
Heap snapshot profiles show memory distribution among your page's JavaScript objects and related DOM nodes.

Record Heap Allocations
Record JavaScript object allocations over time. Use this profile type to isolate memory leaks.

Figure 6-24 Google Chrome profiling tools

Knowledge transfer and support

When choosing to develop a user interface with Brazos, it is important to educate an organization on the tools and underlying open source frameworks that make Brazos possible. There are hundreds of available functions, styles, and utilities that are built into the Brazos toolkit and can make developing supplemental controls very simple.

An in-depth knowledge of the code behind each Brazos control is not necessary since the toolkit is fully supported by BP3. BP3 is constantly working on fixing defects, adding new features, and updating the toolkit. If an organization or individual is interested in their feature requests taking priority over the rest of the items in the Brazos back log or would like custom Brazos controls built for them, several levels of Brazos support agreements are offered by BP3. Contact info@bp-3.com for more details.

6.4 Conclusion

This chapter highlighted the work that BP3 has done with CoachViews. In order to meet complex mobile and desktop requirements while only using the Process Designer, they have developed Brazos, a CoachView toolkit. This toolkit takes advantage of responsive design to optimally display BPM Coaches on any device, regardless of screen size. The toolkit is available at no cost to use for everyone and can be downloaded at the following location:

`http://bp-3.com/brazos`

7

Real world lessons from EmeriCon

EmeriConVIEWS is a generalist toolkit that can improve both the authoring experience and the user experience for IBM Business Process Manager (BPM) Coach NG User Interface Services.

Improved experience means a streamlined and more agile experience for a BPM UI author. For a customer team, it means a more iterative experience and a faster way to implement business requirements. For a user, it means a richer interaction with BPM Coach Pages and visual elements that behave with good performance characteristics.

EmeriConVIEWS is based on the IBM BPM Coach NG framework. It builds on Coach NG by providing a large set of new and improved controls (45+ compared to 15+ Coach NG stock controls), including:

- High-performance tables populated through bound data and Ajax-services
- Drill-down-enabled charts, simple and multi-data series charts
- Masked input fields
- Electronic signature
- Dialogs
- Timers
- On-page Ajax
- Special sections for enhanced layout

All of which are styled in a manner consistent with the Coach NG look and feel.

EmeriConVIEWS also extends the Coach NG framework by simplifying and enhancing key aspects of a BPM User Interface Service, such as:

- Control addressing and exploitation: The ability to refer to controls and to manipulate them meaningfully on a page or in subviews
- Cross-control interactions: The ability for controls to react to each other's actions and state changes
- Control configuration: More configuration-driven control options for common UI requirements in BPM solutions

EmeriConVIEWS aims to provide an authoring experience and a final BPM solution that is only minimally – if at all – affected by the technical complexities inherent to the base Coach NG framework.

7.1 Rationale for EmeriConVIEWS

The Coach NG framework is a powerful and flexible framework compared to its *Heritage Coach* predecessor. But the added power and flexibility provided with IBM BPM 8.x Coach UIs introduces a new set of technical expectations that is not necessarily typical of technical BPM consultant skills. A priority of EmeriConVIEWS is to bridge the gap between the reality that good BPM consultants often do not make good UI developers, and UI developers do not necessarily make good BPM consultants.

The following goals are primary drivers for EmeriConVIEWS:

- Accelerate, streamline, and simplify BPM UI development activities by increasing the consumability of the Coach NG framework – resulting in less moving parts, and providing a more intuitive authoring experience that is more attractive, presentable, and agile in the context of an iterative approach.
- Focus on solving business problems instead of technical problems. Effectively lower the bar for HTML, CSS, Dojo, Ajax expertise and provide a programming model that is simpler to exploit, more consumable to UI authors/developers, and creates less technical moving parts.
- Provide users with a richer, more interactive, and responsive experience, which behaves consistently on both computer Web browsers and mobile device Web browsers.

7.2 Authoring experience

A primary aim of EmeriConVIEWS is to streamline and simplify the UI authoring experience. Because EmeriConVIEWS is built on top of the IBM BPM Coach NG Framework, all conventional BPM UI development approaches work unaltered with the EmeriConVIEWS toolkit and controls. The value of EmeriConVIEWS is both in its additional controls and in its added capabilities (alluded-to in the introductory paragraphs and described in this section).

7.2.1 Expectations

Authoring BPM UIs with EmeriConVIEWS typically entails the following activities:

- Place and style controls from a rich palette of controls, section types, and charts on a Coach Page and in Reusable Coach Views.

- Name controls using their control IDs (see below) to make them easier to interact with at authoring time: The control ID, which is unique across view siblings is used to retrieve a reference to and manipulate the control.
- Bind controls to business data as appropriate: Whereas Coach NG views usually require binding to backing data, EmeriConVIEWS controls are only bound to data if the control represents business data. Otherwise, data binding is not necessary to work with the control.
- Model client-side interactions between controls when needed through simple-to-use event-handling, control properties, methods, and formulas.
- Model Boundary Events for page navigation, including simple client-side validation to inhibit/control page navigation.

In some ways, the authoring experience remains similar to using the base Coach NG framework. In many other ways, the experience is significantly streamlined and accelerated. Some key differences include (in no particular order):

- A simplified control/field addressing scheme that works the same way everywhere (on coach views or coach pages).
- Enhanced field interactions through easily consumable event-handling capabilities, further enhanced by functionality, on controls, sections, charts, and so on, to react to those events (including intuitive control methods).
- Formulas for automatic form-like computations between fields (including support for tabular data).
- Additional controls (approximately 45 versus 15): Charts, Masked Text, Versatile Sections, Electronic Signature, Dialogs, Event Management, Timers, On-Page Ajax Services, and more.
- Configurable charts with built-in drill-down capabilities and a simple-to-use drill-down tree pattern for backing Ajax services, which supports both SQL and JavaScript.
- Pub-sub event management capabilities to manage loosely coupled interactions between UI elements.
- Styling across all controls, sections, and dialogs to provide a consistent Coach NG look and feel.
- Flexible client-side validation and Boundary Event constraint capabilities to control when a page can navigate.
- High-performance tables (displaying both NG Views and lightweight “DOM-based” content rendering) backed by both conventional data bindings and Ajax services.

- General patterns and capabilities that address certain usage in the current iteration of Coach NG framework (see 7.4, “Technical considerations specifically addressed” on page 332).
- Additional configurable options for most controls providing additional flexibility and simplified configuration for authors

7.2.2 Creating interactions between controls

Simplified interactions between controls on a UI require that controls be addressable and that logic be attached to events of those controls in a straightforward manner. EmeriConVIEWS provides core enhancements to facilitate control interactions.

View addressing

A key prerequisite for creating interactions between controls on a Coach NG-based UI is the ability to address/refer to controls and sections on the page or the views/sub-views it might contain. For example, if a UI author needs the title of a section to change when a choice from a Select control is made, for example:

```
horizSection.setTitle("Product: " + select1.getSelectedItem())
```

…how are the references to the Section and the Select control obtained in the first place?

EmeriConVIEWS provides a straightforward addressing scheme to represent the *view tree* in the UI. In the view tree, the coach page is the root of the tree. The leaves of the tree are EmeriConVIEWS controls (or other controls that implement the EmeriConVIEWS addressing scheme). Subviews are intermediate nodes in the tree.

For example, consider the following sample Coach Page that is shown in Figure 7-1 on page 234.

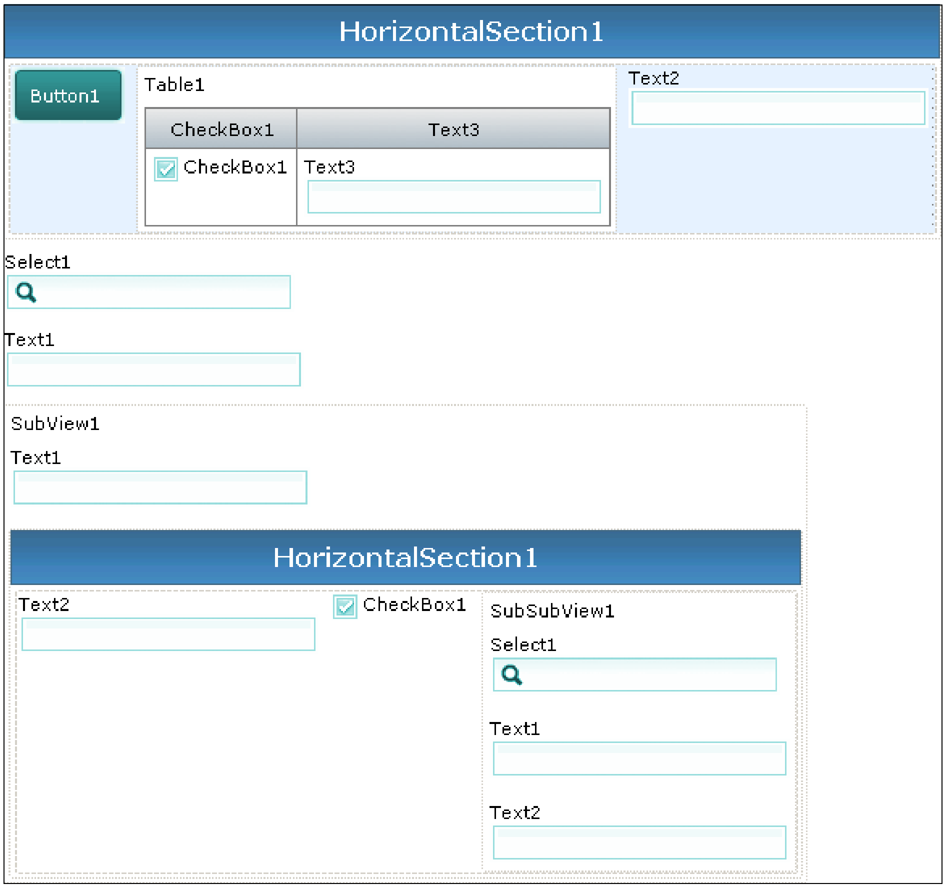


Figure 7-1 Sample Coach Page to illustrate control addressing

This sample Coach Page can be represented by the following Coach View tree shown in Figure 7-2 on page 235.

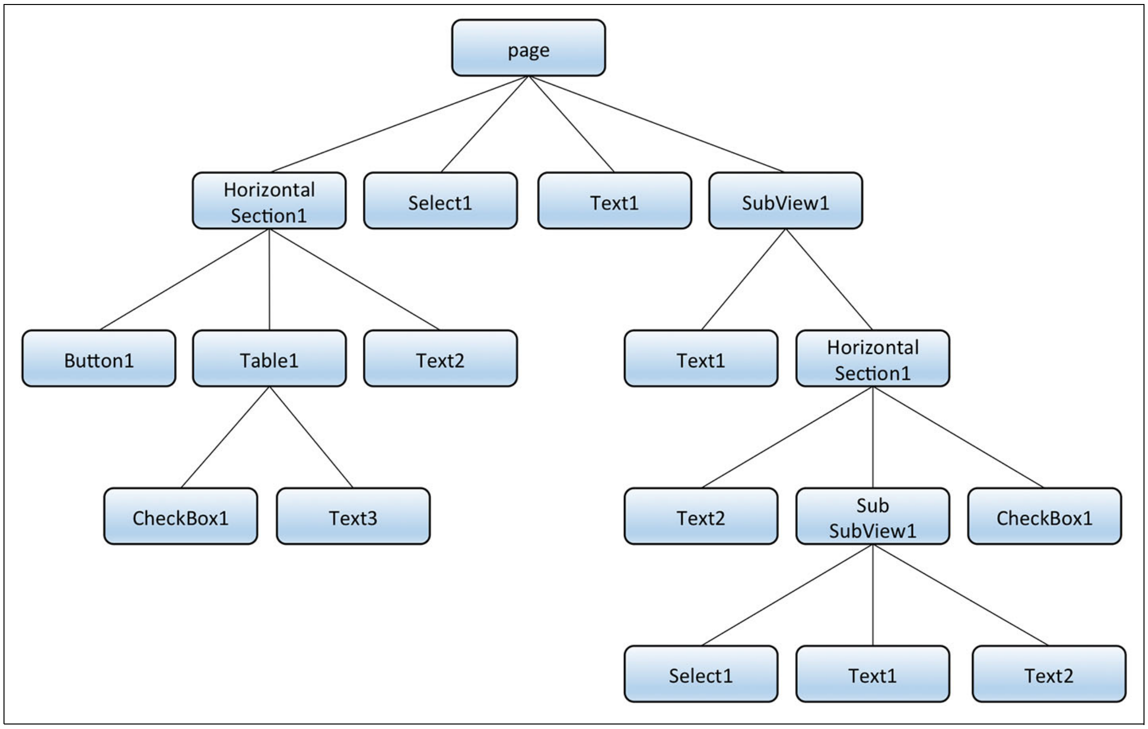


Figure 7-2 Visual Coach View tree diagram

Figure 7-1 on page 234 shows that the tree is consistent with the Coach NG View tree accessible through the Coach NG JavaScript API (for example, using `context.getSubview()`), however such a hierarchy creates authoring and maintenance problems (for example, code breakage) as soon as controls are moved in or out of sections, even if the change is only made for aesthetic reasons.

EmeriConVIEWS addresses the control referencing problem in two ways:

- By creating an intuitive addressing scheme. For example, the Text1 control in SubView1 above can be referenced using the path /SubView1/Text1.
- By not taking sections (except if they are repeatable, such as Tables) into account in the reference of a control. For example, accessing the third check box in Table1 (assuming the table contains at least three rows) is done through the following reference: /Table1/CheckBox1[2] – notice how HorizontalSection1 is not part of the path.

With EmeriConVIEWS, the view tree previously shown in Figure 7-2 on page 235 changes as shown in Figure 7-3 (from a control referencing standpoint).

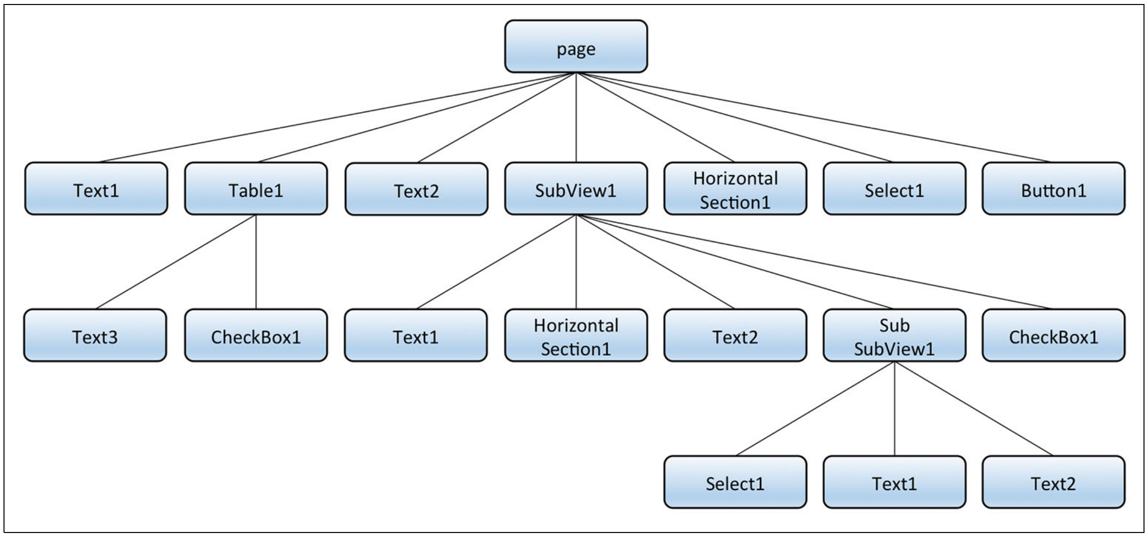


Figure 7-3 Addressing Coach View tree diagram

This tree allows authors to make visual adjustments on the UI without risking UI logic breakage.

The example Coach shown previously is shown again with reference paths for each control, section, and subview as shown in Figure 7-4 on page 237.

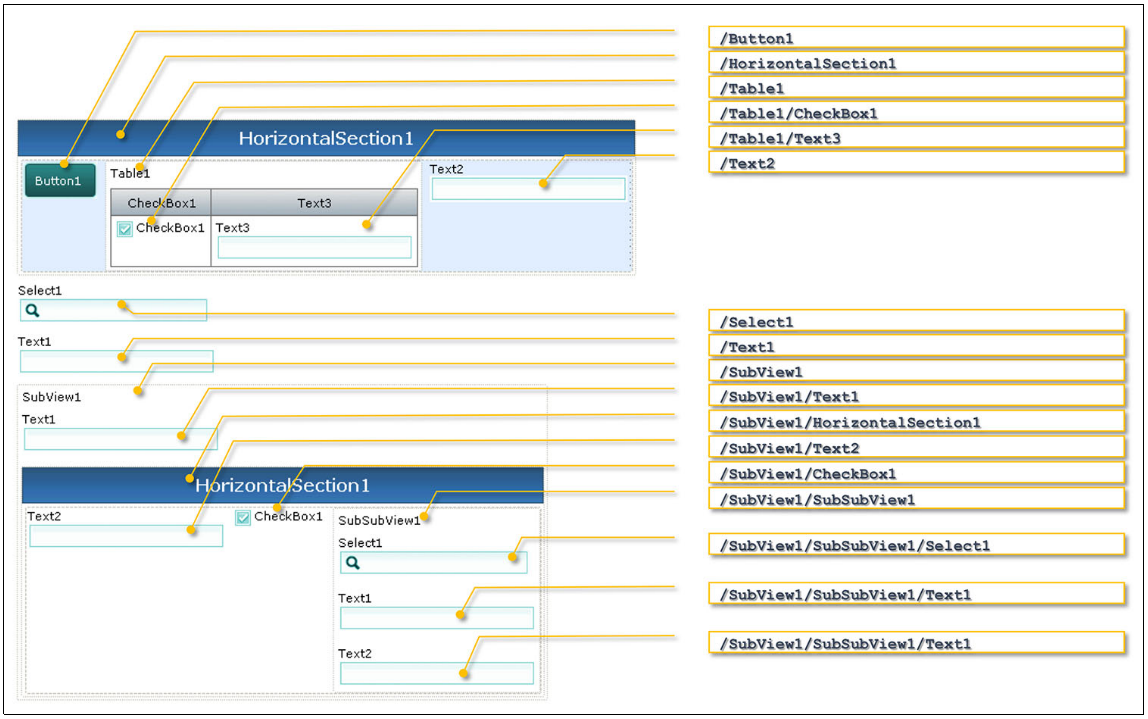


Figure 7-4 Sample Coach Page with called-out control addresses/paths

Important: Paths of controls contain their control IDs (not their labels) and the control IDs of their relevant ancestors in the view tree.

Throughout this chapter, for simplicity and to be able to better visually relate the control on the page or view to the control reference used, the labels of the controls will be the same as, or extremely similar to, the IDs of the controls. But again, the control IDs are used for addressing, not the labels.

The ID of a control can be set from the Coach Page or View editor as shown in Figure 7-5 on page 238.

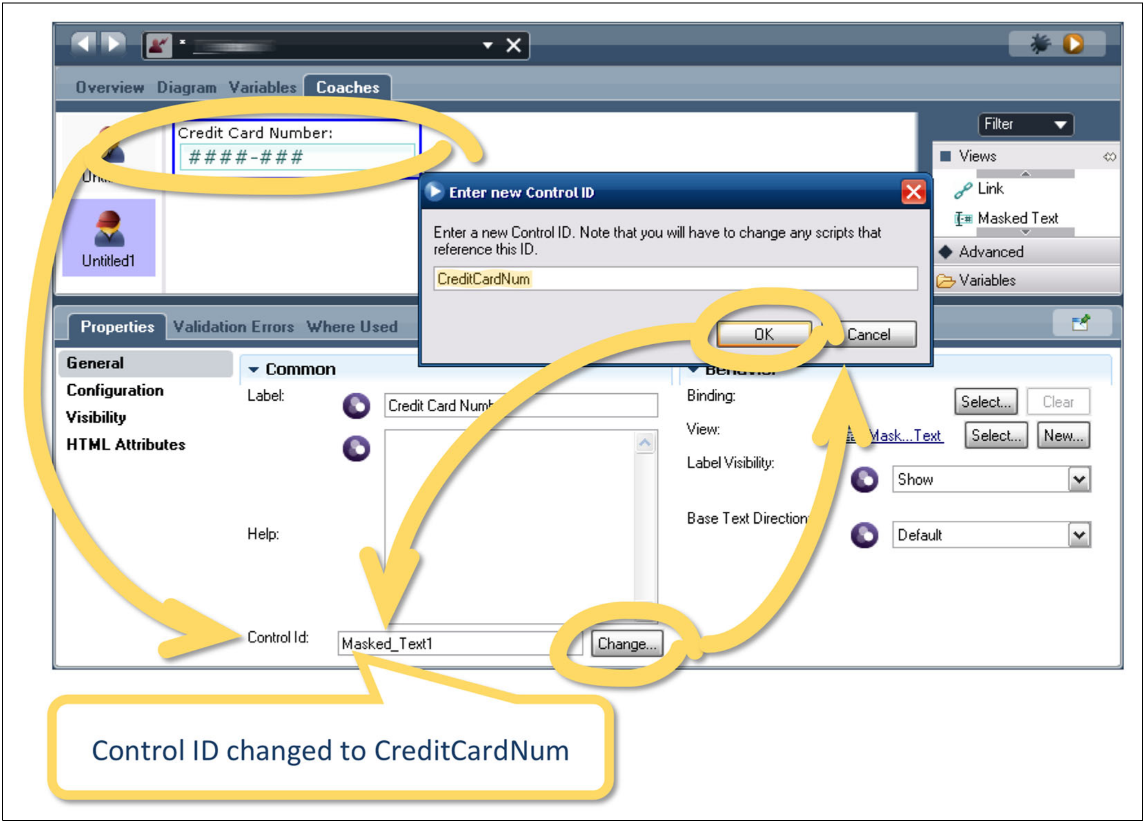


Figure 7-5   Setting the ID of a control in the Coach Page or Coach View editor

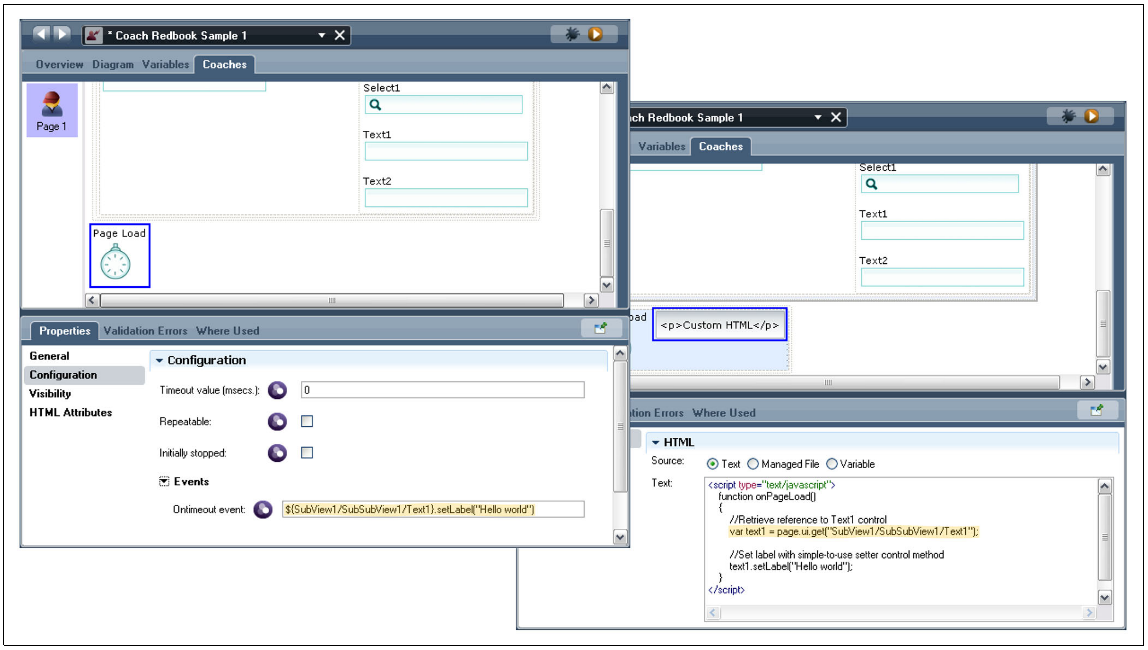
Note: The Process Designer editor enforces the uniqueness of Control IDs at a given level in the View tree. This ensures that the addressing scheme is never ambiguous, even between sibling views

EmeriConVIEWS provides several ways to use the addressing scheme.

Using a JavaScript block at the top level of a Coach Page, a user might programmatically set the label for Text1 in SubSubView1 as shown in the left part of Figure 7-6 on page 239.

Control addressing is available in many situations throughout EmeriConVIEWS, in business logic, and for control events and formulas (more on these later).

The same behavior can be accomplished with the more compact approach that is shown in the right part of Figure 7-6.



Configuration

Timeout value (msecs.): 0

Repeatable: ☐

Initially stopped: ☐

Events

Ontimeout event: ${SubView1/SubSubView1/Text1}.setLabel("Hello world")

Source: [✓ Text] Managed File Variable

Text:

```
<script type="text/javascript">
  function onPageLoad()
  {
    //Retrieve reference to Text1 control
    var text1 = page.ui.get("SubView1/SubSubView1/Text1");

    //Set label with simple-to-use setter control method
    text1.setLabel("Hello world");
  }
</script>
```

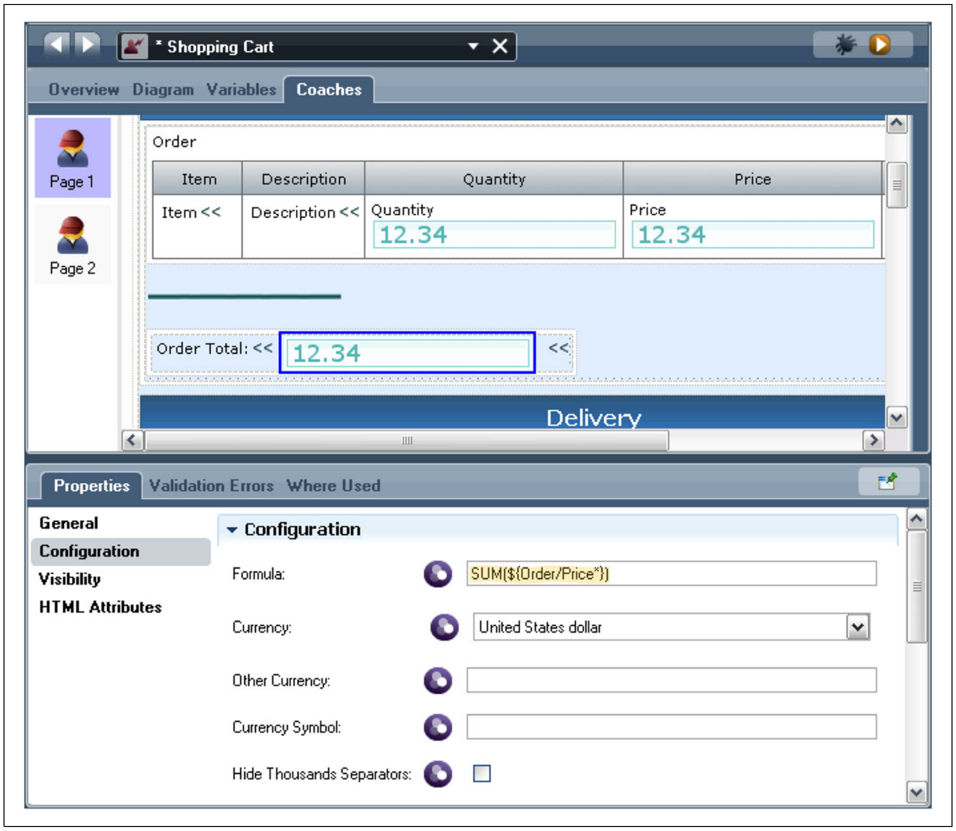
Figure 7-6 Example of control addressing in non-inline event-handling logic and example of control addressing in inline event-handling logic

The ${*control-reference*} notation is an optional addressing shorthand that can be used in inline event-handling logic. This shorthand, which helps with compactness in inline event logic, is not allowed in regular JavaScript code. A less compact expression of the above logic might be:

```
var text1 = page.ui.get("SubView1/SubSubView1/Text1");
text1.setLabel("Hello world");
```

Additional details about event handing are provided in the section below entitled “Control event-handling” on page 241.

Lastly, Figure 7-7 shows control addressing used in formula-based computations (in this case the formula sums up all Prices in the Order table).



Item	Description	Quantity	Price
Item <<	Description <<	Quantity 12.34	Price 12.34

Order Total: << 12.34 <<

Configuration

Formula: SUM(${Order/Price*})

Currency: United States dollar

Other Currency:

Currency Symbol:

Hide Thousands Separators: ☐

Figure 7-7 Example of control addressing in formula-based computations

Additional details about using formulas are provided below in “Formulas” on page 247.

In summary, EmeriConVIEWS addressing provides a simple and consistent way of referring to controls and sections in regular JavaScript code (on the Coach page or in the Inline JavaScript section of event handlers of subviews), on configuration-based event handlers of controls/sections, or in formulas. Both absolute and relative addressing are supported.

Good practices

Take a look at some good practices:

- There are appropriate places and practical uses for both absolute and relative addressing. However, relative addressing provides a more self-contained approach, which does not assume that child views know about the structure and naming of their containing parent views. This helps create a more robust and flexible solution.

- The publish/subscribe feature of EmeriConVIEWS (discussed in "Event subscription" on page 317) can also help create cleaner, more flexible, and more reusable solution components:
  - For UI requirements that create highly interconnected interactions between UI components that do not have a clear self-contained *line of sight* between each other.
  - For *loose coupling* implementation needs between one to one, or one and many, UI controls.

Control event-handling

Coach NG provides event handlers (load, view, change, and so on) for Coach View developers. Accessing or using these event handlers usually entails creating a Coach View, which is not practical in a scenario where the author simply needs to create *surface* relationships between controls on a page or in a reusable view.

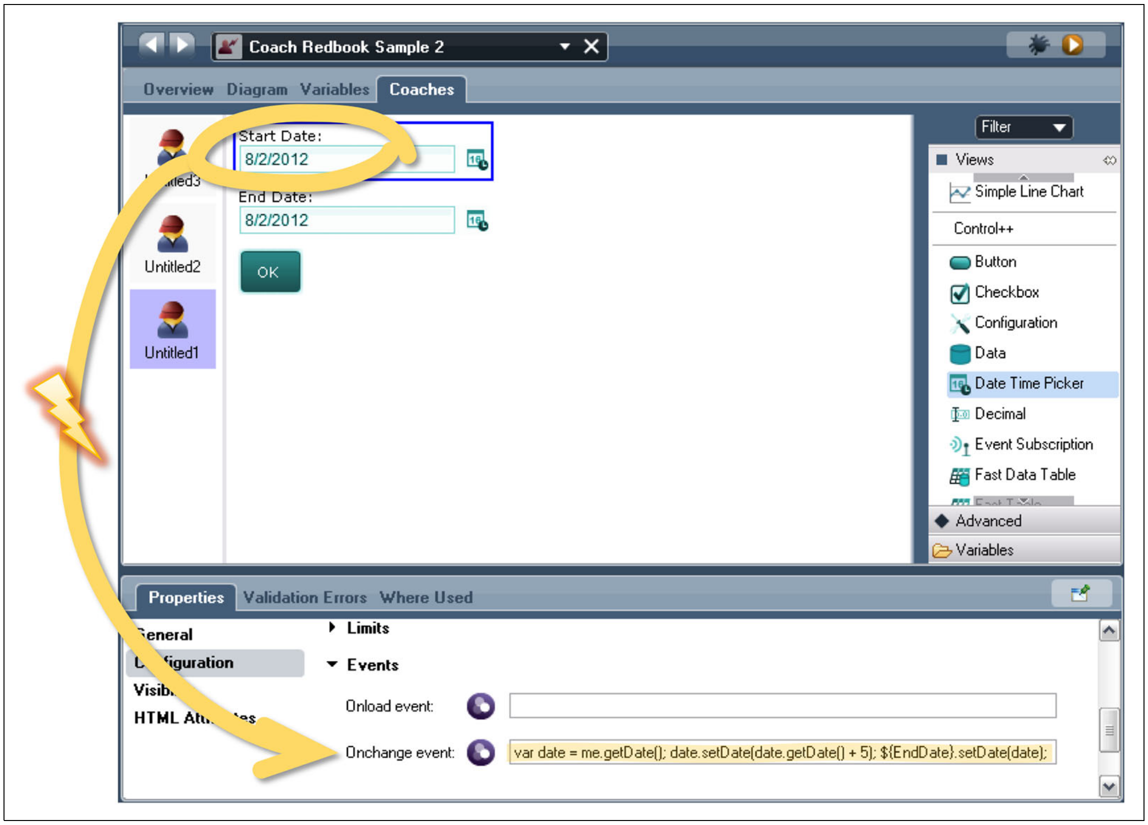
The following kinds of interaction:

- Showing a dialog when clicking a button
- Changing the title of a section when an item is selected in a Select control
- Immediately displaying an error message when invalid data is entered
- Challenging a user before deleting a row in a table

…should be simple to implement by associating logic with events on the emitting controls (for example: "on click" for a Button, "on change" for a Text or Select control, "on row delete" for a table control), and without the need to create extra logic or constructs to support the interaction.

EmeriConVIEWS provides an efficient mechanism to connect control events with business logic.

Figure 7-8 on page 242 automatically sets an end-date to be five days after the selected start date.



```
var date = me.getDate(); date.setDate(date.getDate() + 5); ${EndDate}.setDate(date);
```

Figure 7-8 Inline event-handling logic for Date Time Picker control "onchange" event

In addition to the ${…} shorthand reference syntax, inline event handlers provide a `me` variable (as shown in the prior illustration) for convenience. The `me` variable refers to the control that originated the event.

Different controls support different events. Most controls support the "on load" event. Many additional events are supported depending on control/section types. Other examples of events include:

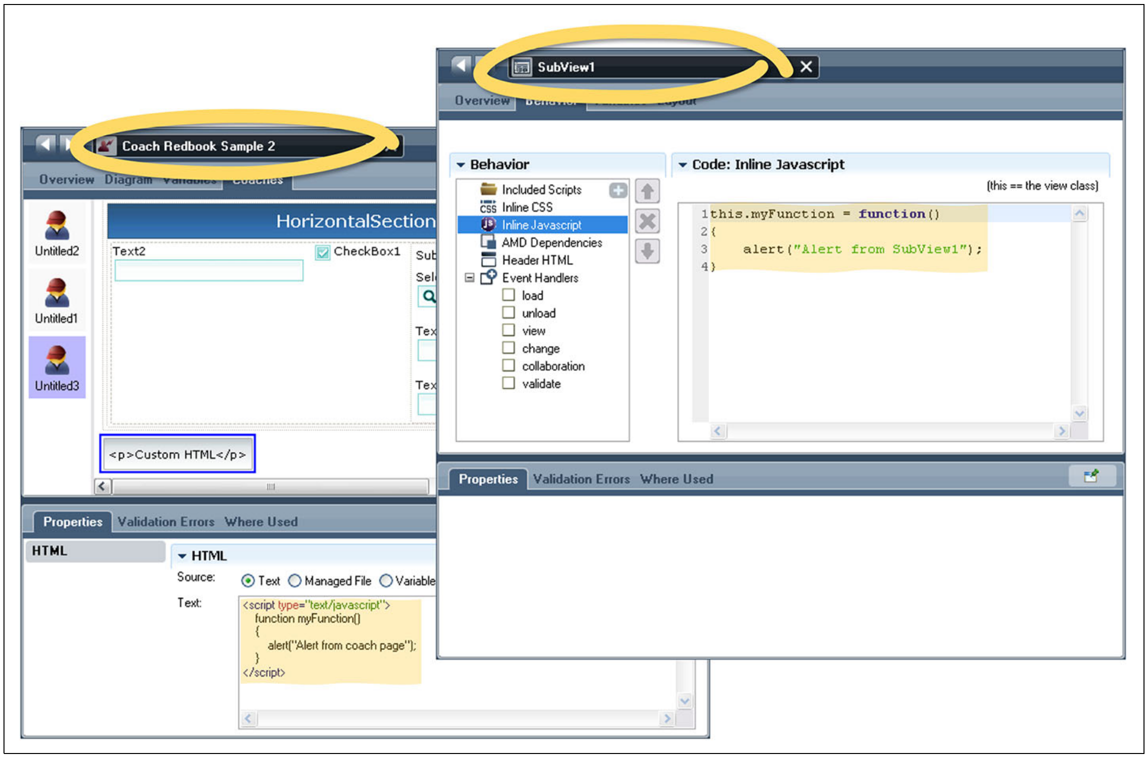
- On click (for button, link)
- On change (for text, data time picker, select, check box, masked text, decimal, integer, and so on)
- On tab/section changed (for tab/accordion/display sections)
- On row deleted (for tables)
- On cancel (for dialog sections)
- On timeout (for timer)
- On result (for on-page Ajax service control)
- On event (for event subscription control)

The ability to attach event logic at the configuration level provides a simple mechanism to create *surface* interactions between controls without the need for additional constructs or complexity. The event handling/attachment mechanism can be used to create simple to very sophisticated interactions.

Resolving ambiguity when calling external functions

Suppose that a Coach View is made of several subviews and each subview and the Coach Page itself contain a `<script>` JavaScript block with a function named myFunction() defined in each. With JavaScript, the latest global definition of myFunction() will overwrite all the other global definitions.

To eliminate this situation, EmeriConVIEWS allows the author to define and unambiguously call the wanted version of myFunction. Figure 7-9 shows how myFunction could be defined both at the SubView1 and the Coach Page level.

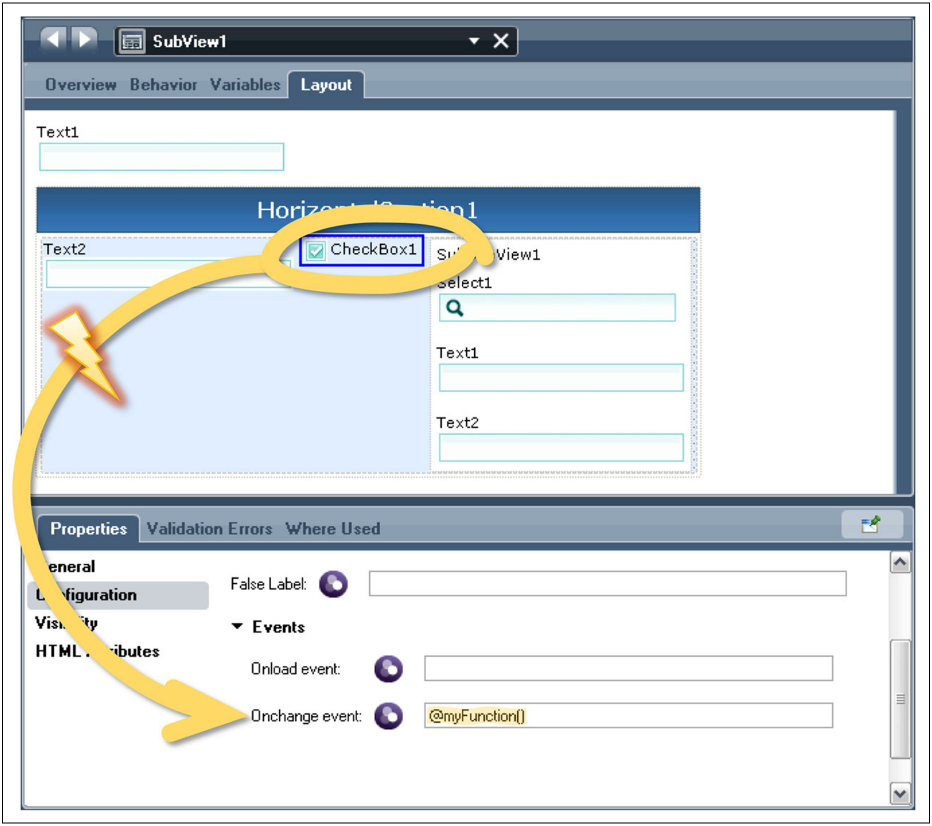


```
this.myFunction = function()
{
    alert("Alert from SubView1");
}
```

```
<script type="text/javascript">
  function myFunction()
  {
    alert("Alert from coach page");
  }
</script>
```

Figure 7-9   Defining functions to be called by event handlers

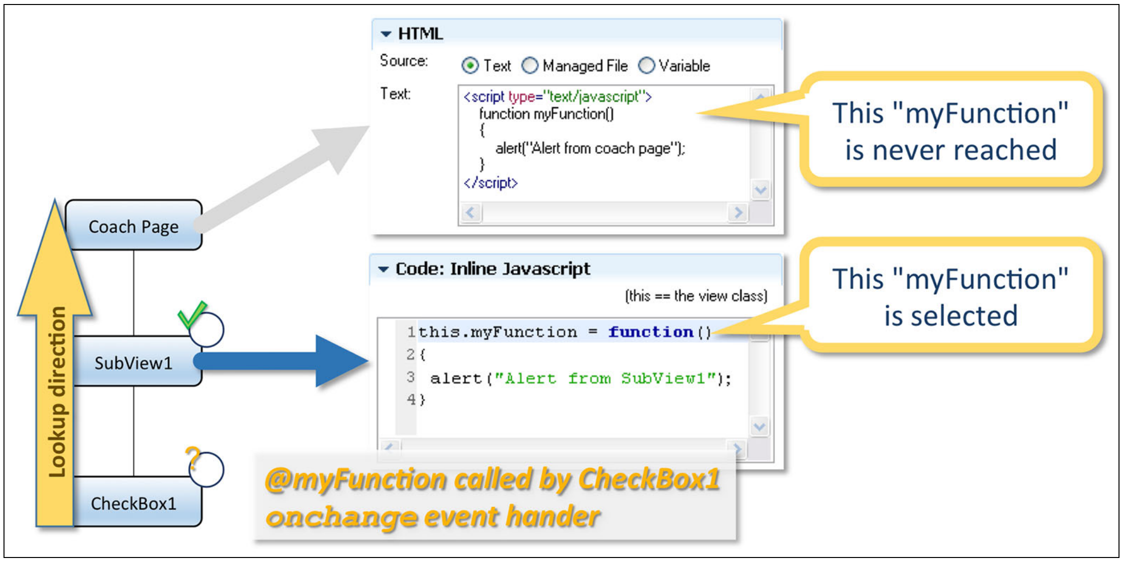
And Figure 7-10 shows how to unambiguously call myFunction() when CheckBox1 in Subview1 is changed (notice the @function-name syntax).



Onchange event: @myFunction()

Figure 7-10 Invoking non-inline logic in event handlers

When calling functions with the `@function-name` syntax, the function *nearest* to the calling control is invoked. Figure 7-11 on page 245 shows how a named function (*myFunction* in this case) is searched up the view chain from the control calling the function, up through the parent chain until the function is found.


HTML
Source: [✓ Text] Managed File Variable
Text:
Code: Inline Javascript
(this == the view class)
This "myFunction" is never reached
This "myFunction" is selected
Coach Page
SubView1
CheckBox1
Lookup direction
?
@myFunction called by CheckBox1 onchange event hander

```
<script type="text/javascript">
  function myFunction()
  {
    alert("Alert from coach page");
  }
</script>
```

```
this.myFunction = function()
{
 alert("Alert from SubView1");
}
```

Figure 7-11 Event handlers calling external functions unambiguously

In the previous example, myFunction in the Coach Page script block is never used because myFunction in SubView1 is reached first.

Good practices:

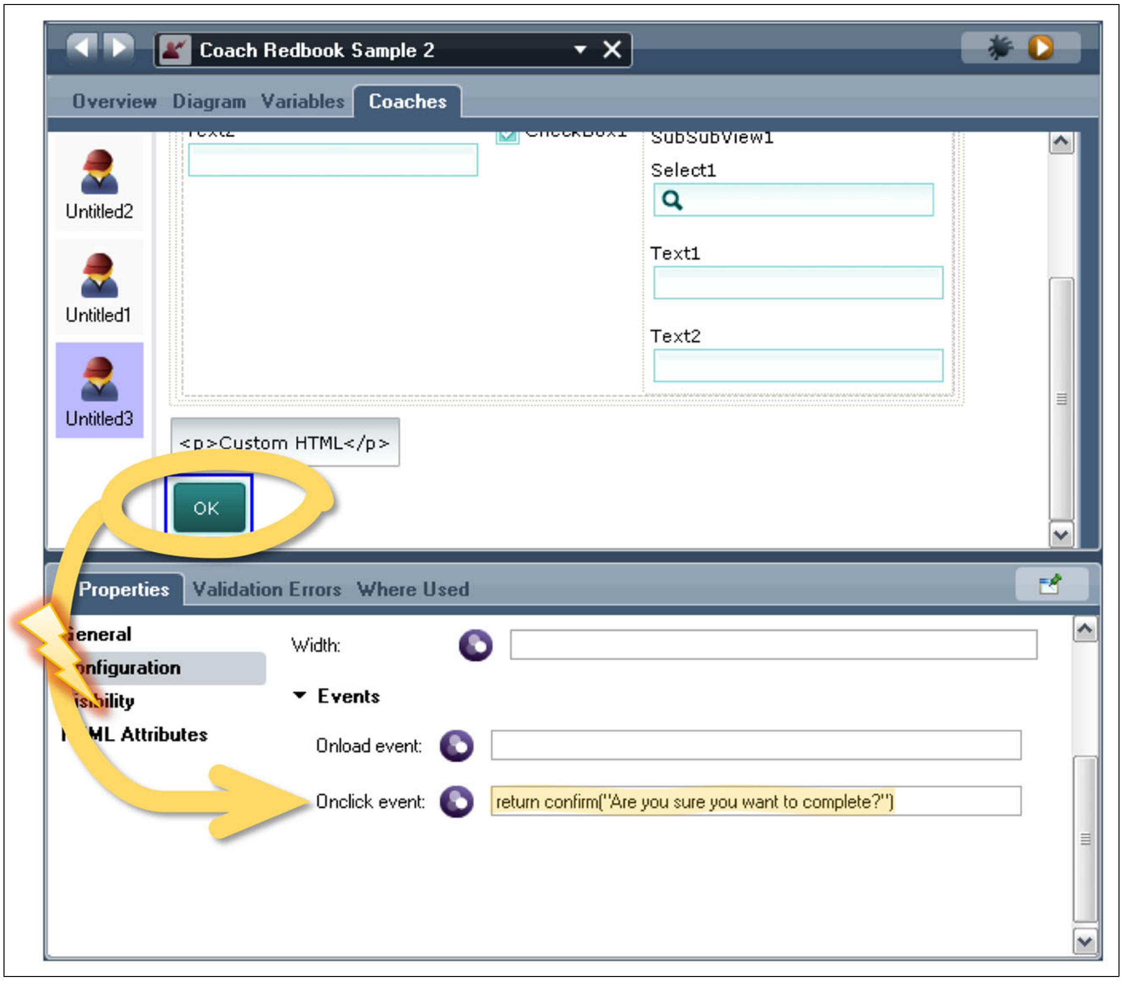
- Using inline event-handling is good for simple logic that only requires a few statements. For more involved business logic, call a function using the `@function-name` syntax either defined in a <script> block or in the "Inline JavaScript" section of a Coach View.
- When specific actions (whether simple or complex) must be executed by several controls/views as a result of one or more control events. It is sometimes advantageous to either call business logic in a centrally defined function, use the publish/subscribe feature of EmeriConVIEWS (discussed in "Event subscription" on page 317) to create cleaner and more maintainable UI interactions, and avoid duplicating logic across several event handlers.
- When defining functions to be called by event handlers for controls defined in subviews, it is usually safer to define the functions in the "Inline JavaScript" section of the subview (see Figure 7-9 on page 243).
- Any function defined in a script block as `function myFunction(){...}` always ends up in the global namespace. To reduce the potential of unintentional function definition overwrite, script blocks in Custom HTML should *not* be used in reusable Coach Views. Functions are more safely defined in the Inline JavaScript section of the reusable Coach View.

- Using functions defined in script blocks at the Coach Page level is appropriate. Such functions are called as last resort as long as functions with the same names are not first found in the parent chain of calling controls.

Controlling Boundary Events with Event handling

Certain controls such as Buttons, Timers, Simple Dialogs, can emit Boundary Events, which cause Coach Page navigation. Sometimes it is very useful to be able to inhibit the firing of the Boundary Events based on business logic.

Inhibiting navigation can be accomplished by returning false in those event handlers. Figure 7-12 shows how a simple confirmation challenge can be issued to an end user when the **OK** button is pressed.



return confirm("Are you sure you want to complete?")

Figure 7-12 Controlling Boundary Event emission in event handlers

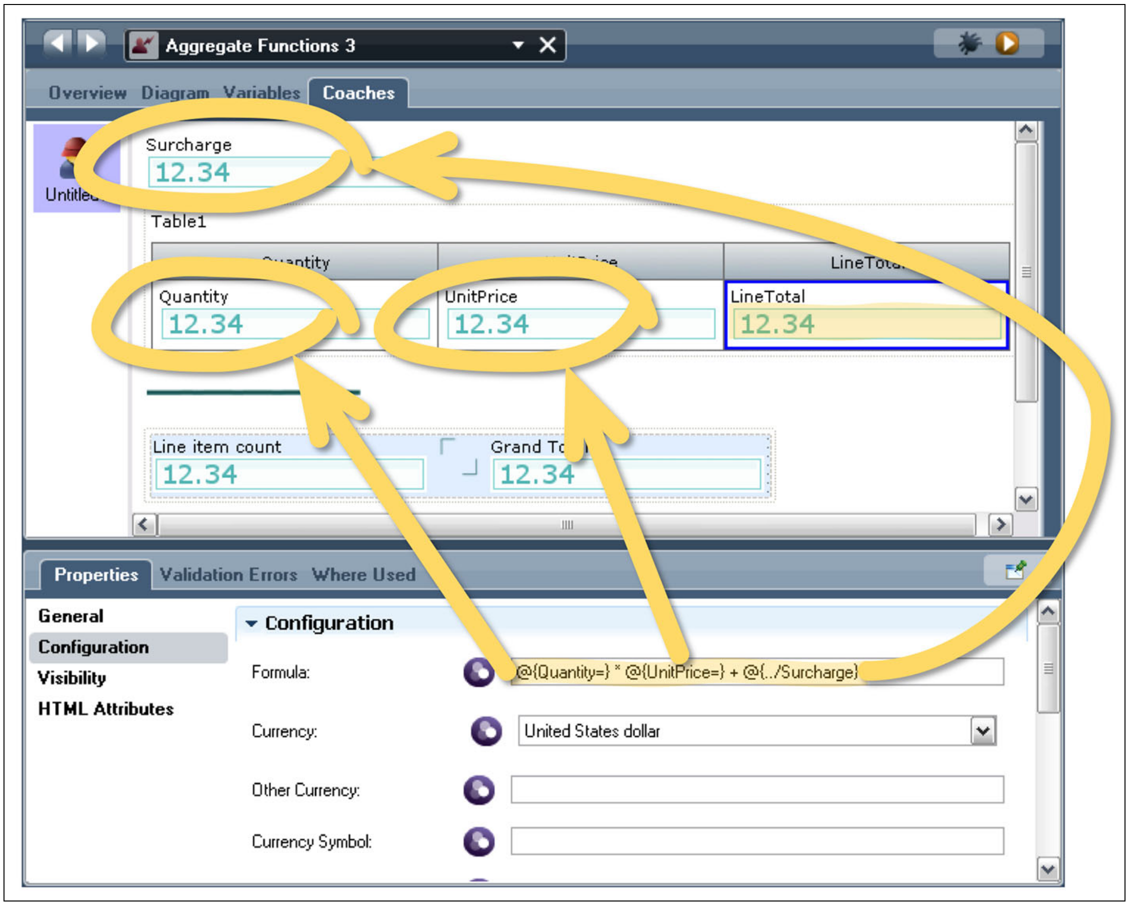
Formulas

Formulas are most useful when the value of a control is computed from the value of one or more other controls. Formulas allow UI authors to create interactions between controls in a way similar to how cells behave in a spreadsheet. The following examples can benefit from formula support in EmeriConVIEWS:

- To compute the value of the total cost of an order based on a total plus sales taxes.
- To compute the cost of a line item based on unit price multiplied by the number of units in a table.

Formulas can also be used for non-numeric computations where the text content of a control depends on one or more other controls.

Figure 7-13 on page 248 shows formulas in the context of tabular (that is, array) data where the line item price is calculated based on quantity x item price (on the same row as the line item price) + a surcharge amount.



Formula: @{Quantity=} * @{UnitPrice=} + @{../Surcharge}
Currency: United States dollar
Other Currency:
Currency Symbol:

Figure 7-13 Formula-based control value computations

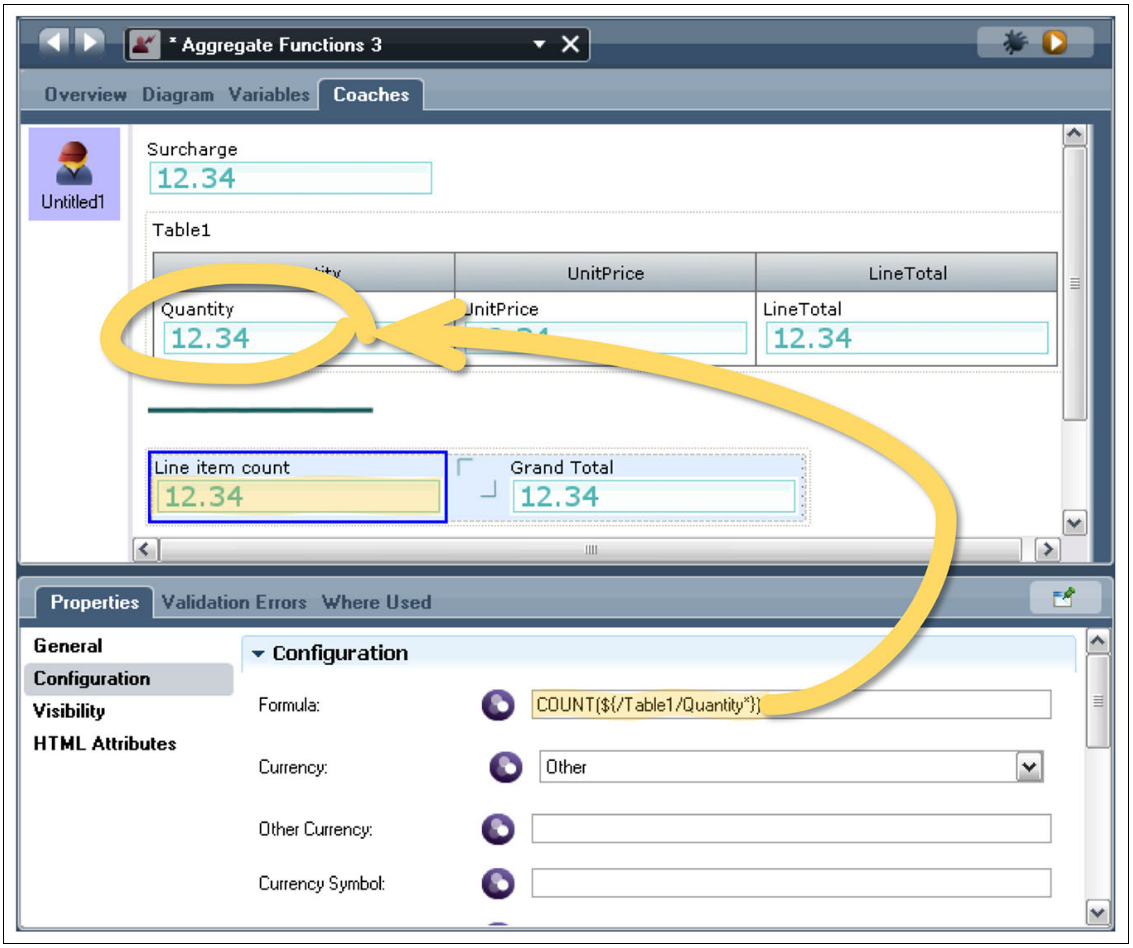
Formulas use shorthand notations for compactness. For example, the `@{control-ref}` notation is equivalent to `${control-ref}.getValue()` in the case above, but is simpler to input and read. To refer to a field in the same row as the field containing the formula, the control reference uses the `{control-ref=}` `notation`. = "means same row as me" and is only useful in the context of tabular data.

The language for formulas is JavaScript so that the sophistication of formula expressions is only limited by the JavaScript language. For example, the formula above could include additional logic such as the following if business requirements so prescribed:

Math.ceil(@{Quantity=})*Math.round(@{UnitPrice=})+@{../Surcharge}

Lastly, conditional calculations could be implemented by using ternary operators to provide both value and logic-based calculation capabilities (Figure 7-16 on page 251).

Figure 7-14 shows the use of *aggregate* functions with tabular data.



Configuration

Formula: COUNT(${/Table1/Quantity*})

Currency: Other

Other Currency:

Currency Symbol:

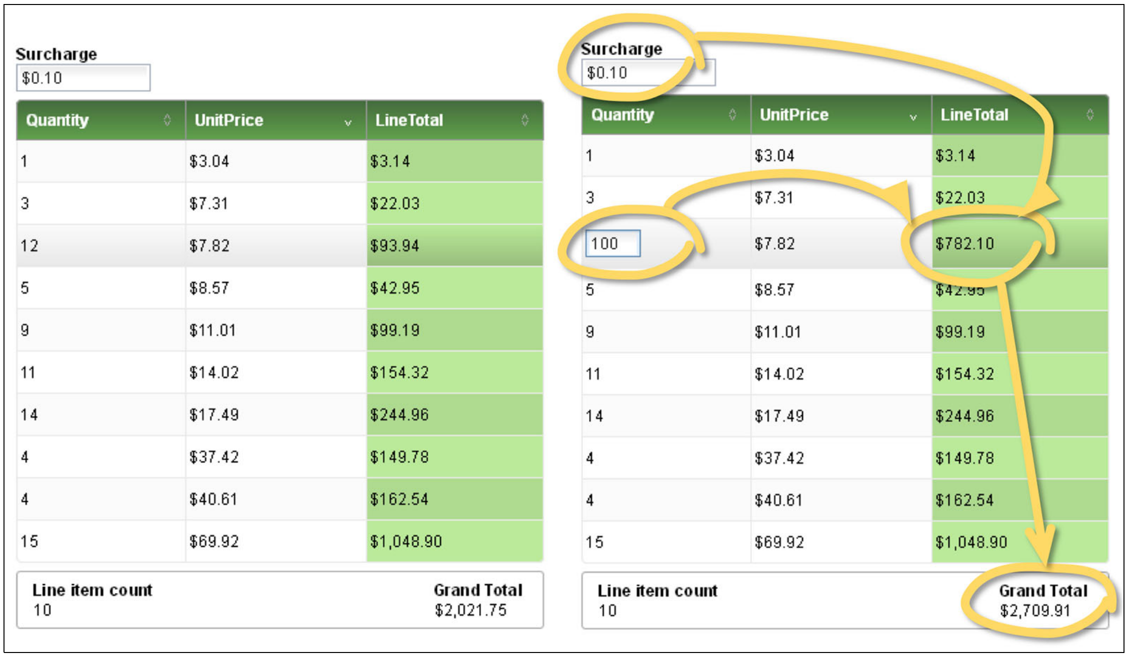
Figure 7-14 Using aggregate functions in formulas

Note the use of * in the reference ${/Table1/Quantity*}, which signifies *all* instances of the /Table1/Quantity field in the table.

Aggregate functions such as COUNT, SUM, AVG, MIN, MAX are available to use in functions. The formula model is extensible and straightforwardly accommodates the creation of other JavaScript based *user-defined* functions.

Formula-based calculations are triggered and propagated automatically and efficiently to “interested” controls. A change in one control instantly updates any other control that directly or indirectly references it.

Figure 7-15 shows the Coach Page previously designed in its rendered state, and with the effect of a quantity change in one row.



Surcharge
$0.10

Quantity	UnitPrice	LineTotal
1	$3.04	$3.14
3	$7.31	$22.03
12	$7.82	$93.94
5	$8.57	$42.95
9	$11.01	$99.19
11	$14.02	$154.32
14	$17.49	$244.96
4	$37.42	$149.78
4	$40.61	$162.54
15	$69.92	$1,048.90

Line item count
10

Grand Total
$2,021.75

Surcharge
$0.10

Quantity	UnitPrice	LineTotal
1	$3.04	$3.14
3	$7.31	$22.03
100	$7.82	$782.10
5	$8.57	$42.95
9	$11.01	$99.19
11	$14.02	$154.32
14	$17.49	$244.96
4	$37.42	$149.78
4	$40.61	$162.54
15	$69.92	$1,048.90

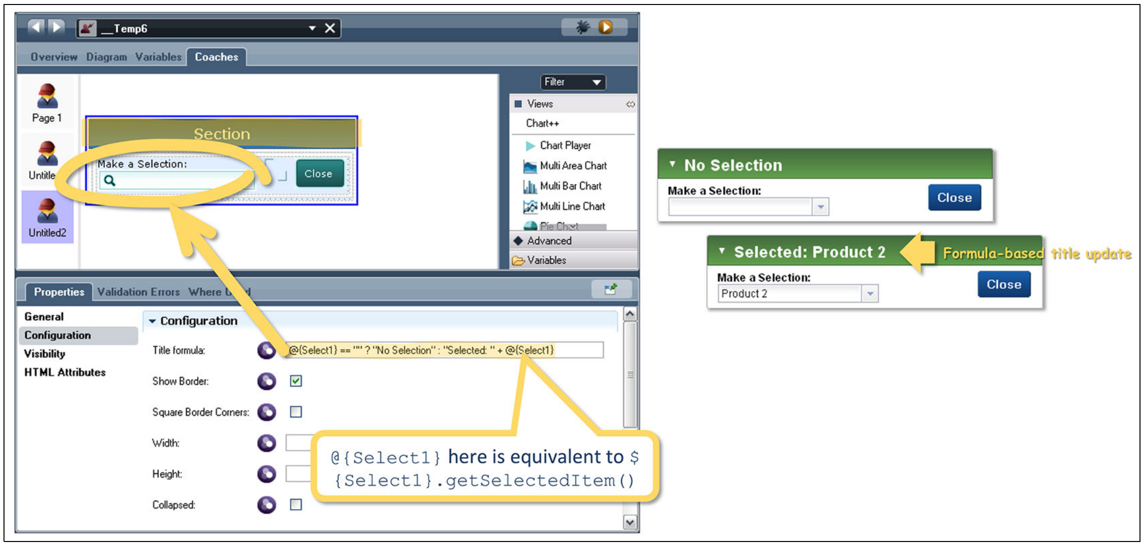
Line item count
10

Grand Total
$2,709.91

Figure 7-15 Example of automatic formula result propagation

Formulas can also be used with alphanumeric values to create automatic relationships between controls. With such an approach, a section title could be updated with a customer name, or an Output Text control could automatically provide summary information.

Figure 7-16 on page 251 shows a section title updated based on a selected item. This behavior cannot be achieved using normal bindings because of the “inline” logic that processes the title.


Title formula: @{Select1} == "" ? "No Selection" : "Selected: " + @{Select1}

Show Border: ✓

Square Border Corners: ☐

Collapsed: ☐

`@{Select1}` here is equivalent to `${Select1}.getSelectedItem()`

No Selection

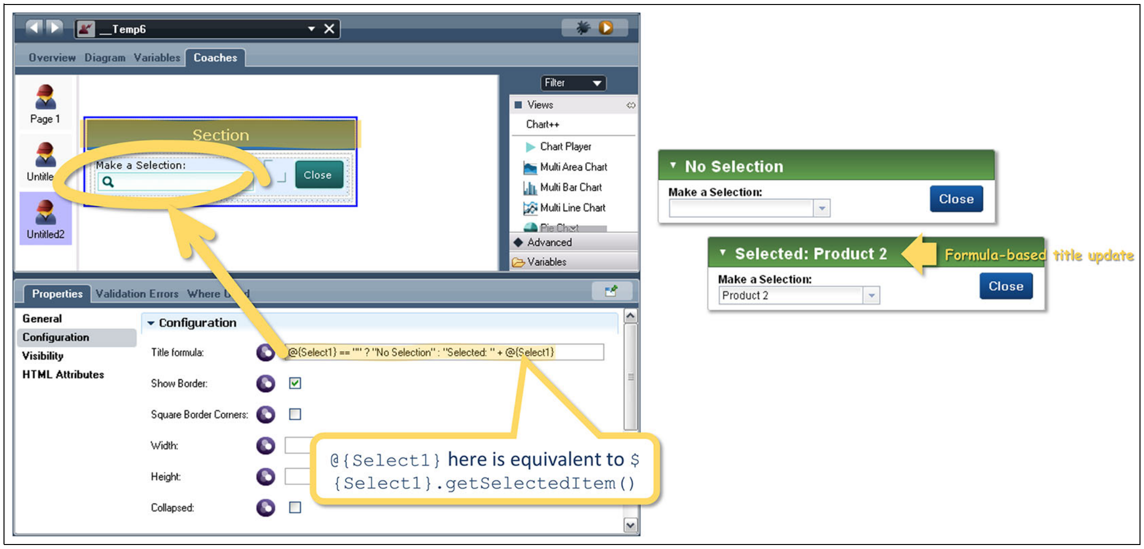
Make a Selection:

Selected: Product 2

Make a Selection: Product 2

Figure 7-16 Using formulas for non-numeric computations

Note the use of the ternary operator to inject additional logic into the formula that is based on whether an item was selected from the Select control or not. At run time, the following effect is created as shown in Figure 7-17.


Title formula: @{Select1} == "" ? "No Selection" : "Selected: " + @{Select1}

Show Border: ✓

Square Border Corners: ☐

Collapsed: ☐

`@{Select1}` here is equivalent to `${Select1}.getSelectedItem()`

No Selection

Make a Selection:

Selected: Product 2

Make a Selection: Product 2

Figure 7-17 Runtime behavior example for formulas with non-numeric computations

Good practices:

- Use formulas when a control *potentially* needs to be automatically updated several times while the UI is displayed. If the formula is used to merely initialize a control's value and further automatic updates are not needed, the control's `onload` event should likely be used instead
- Control references in formulas should be expressed using relative addressing when appropriate to make the formula more portable and self-contained

  Numeric calculations with JavaScript are vulnerable to errors due to inherent language limitations with floating point precision. To ensure consistent and deterministic behavior in calculations, use the `ROUND(number, precision)` formula function for calculations involving decimals in formulas. For example, the line item price formula shown previously could be rewritten as:

  `ROUND(@{Quantity=}, 2) * ROUND(@{UnitPrice}, 2) + ROUND(@{../Surcharge}, 2)`...to ensure maximum precision for 2 decimals.

7.2.3 Creating on-page and off-page business logic

EmeriConVIEWS does not conflict with existing Coach NG functionality to create business logic in BPM UIs. Instead, it fills the common need to express logic and create greater interactivity on the client side, extending the Coach NG programming model with simple to use constructs and control methods.

All existing Coach NG functionality such as using JavaScript activities with the "Stay On Page" event type, page flows between Coach Pages with logic, or service calls in between is still as usable as before without any adaptation.

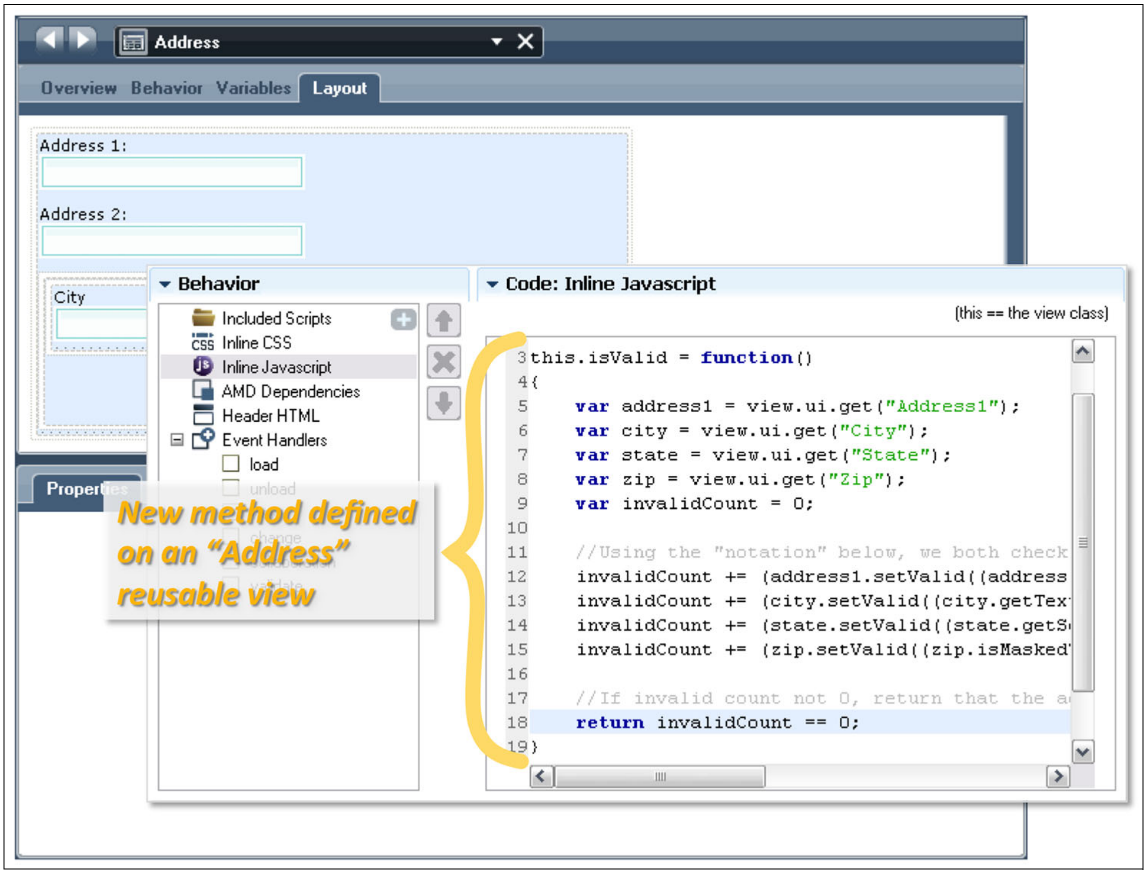
EmeriConVIEWS simply facilitates (often significantly) the following types of interactions and requirements in a way that is more readily exploitable than with the base Coach NG framework:

- Creating logic on control events by using client-side event handlers without the need to create additional views and sub-views and supporting logic (for example, see Figure 7-8 on page 242).
- Creating client-side validation logic and prevent page navigation if validation fails. See 7.2.5, "Handling validation" on page 261 for additional details.
- Creating reusable views that behave like fully exploitable UI components through configuration/properties, methods, and events, similar to well-established authoring practices and UI authoring tools and environments.
- Invoking an Ajax service that is based on a user action (for example, looking up shipping costs that are based on a shipping method that is selected in a Select or Radio Button control).

Adding logic to reusable Coach Views

Reusable Coach Views can be augmented from purely visual elements to full-blown UI components with properties, events, and methods. The advantage of doing so is that it extends the reusability aspects of a Coach View to include behavioral aspects that can be invoked from client-side business logic.

For example, the usefulness and convenience of the following reusable view that is shown in Figure 7-18 is augmented by creating a custom *isValid* method.



```
this.isValid = function()
{
    var address1 = view.ui.get("Address1");
    var city = view.ui.get("City");
    var state = view.ui.get("State");
    var zip = view.ui.get("Zip");
    var invalidCount = 0;

    //Using the "notation" below, we both check
    invalidCount += (address1.setValid((address
    invalidCount += (city.setValid((city.getTex
    invalidCount += (state.setValid((state.getS
    invalidCount += (zip.setValid((zip.isMasked

    //If invalid count not 0, return that the a
    return invalidCount == 0;
}
```

Figure 7-18 Extending reusable view functionality with author-defined methods

This approach allows the creation of any control method, whether getter and setter or action methods, to make the control more programmatically usable.

Good practices:

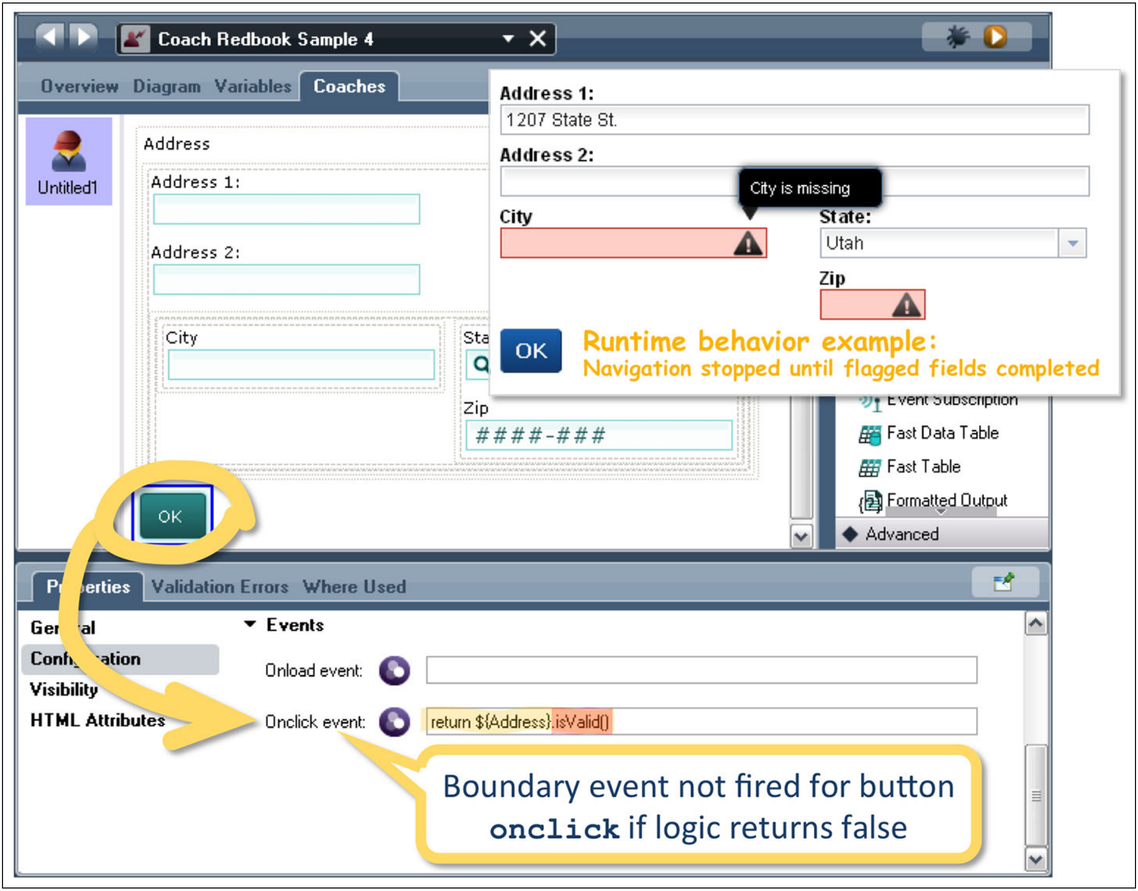
- The `context` object and all other Coach NG objects are fully accessible from inside custom functions, however, `this` can often be ambiguous in JavaScript depending on how the method is called. This is just as true in the context of EmeriConVIEWS as it is for basic Coach NG. Accordingly, a good practice is to save a reference to `this` as early as possible in the Inline JavaScript section of the reusable view. In the previous illustration, a `view` variable is declared on line 1 (which is not visible) as follows:

  ```
  var view = this;
  ```

- Then `view` can reliably be used to refer to the Coach View object in subsequent parts of the Coach View logic and events for such statements as:

  `view.context.binding.get`(“`value`”) and so forth…

Once a view is augmented with methods, those methods can be used in the context of any JavaScript logic. Figure 7-19 on page 255 shows the isValid method is used in the context of a button onclick event to help control whether the page should navigate.



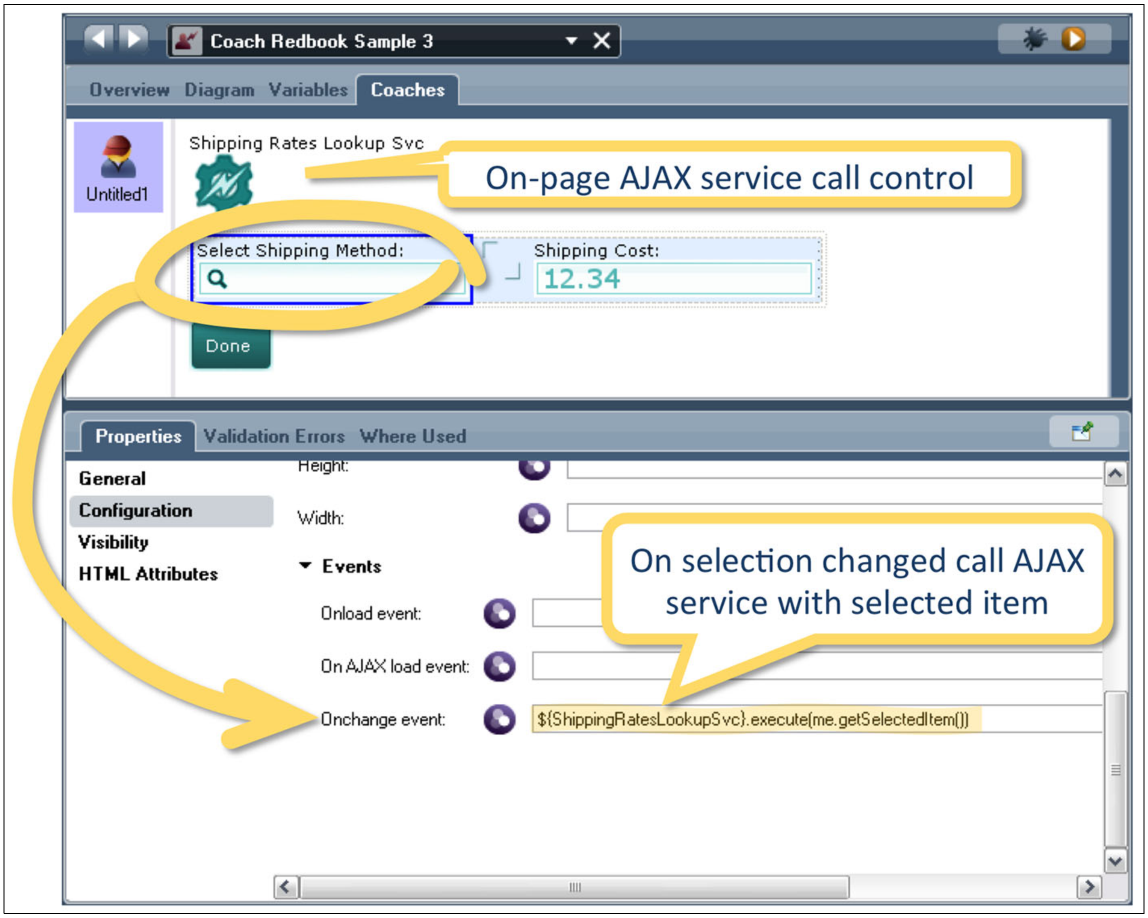
```
return ${Address}.isValid()
```

Figure 7-19 Invoking augmented reusable view methods in business logic

Invoking Ajax services based on user actions

To keep user interactions as seamless and responsive as possible, it is often useful to invoke server-side business logic (as Ajax services) in the background when users perform certain actions on the UI.

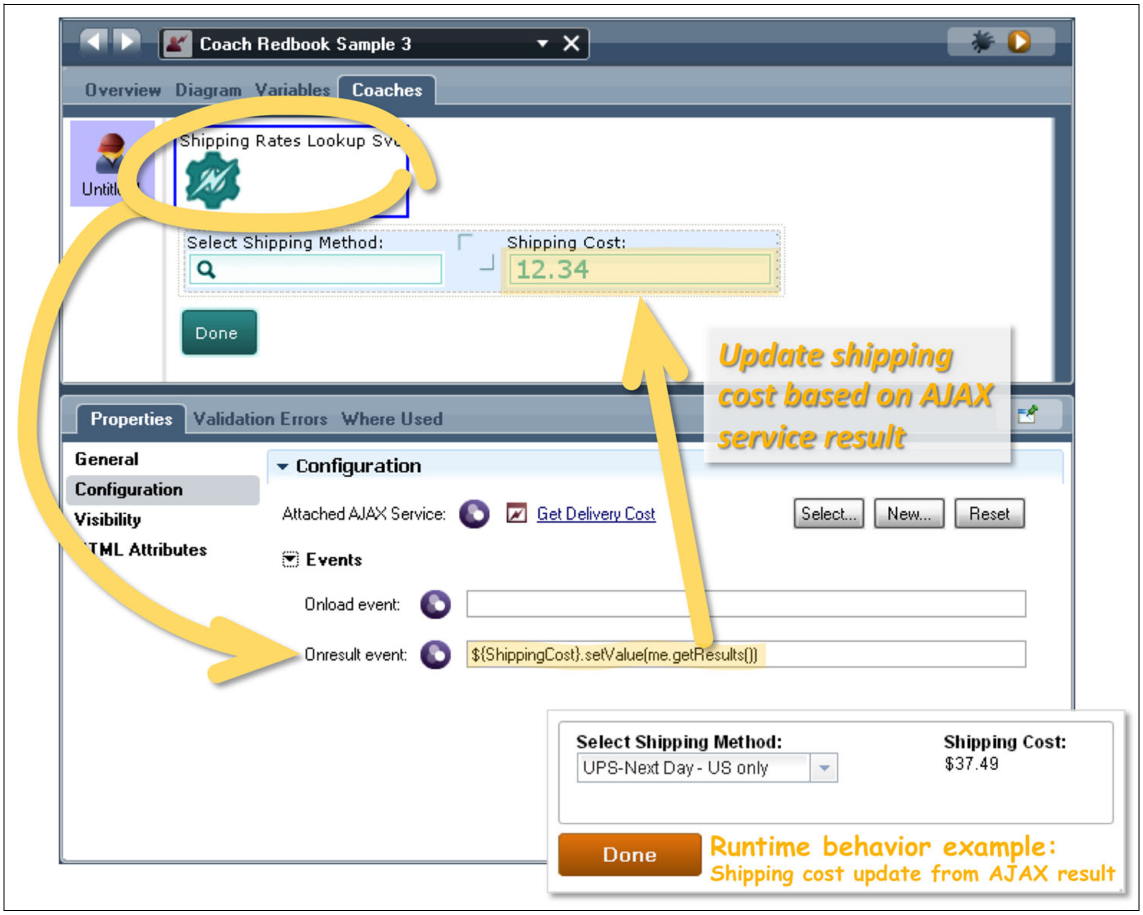
Figure 7-20 shows a simple way to seamlessly retrieve the cost of shipping based on the selected shipping type in a Select control.



```
${ShippingRatesLookupSvc}.execute(me.getSelectedItem())
```

Figure 7-20 Triggering Ajax service from control event

Figure 7-21 shows how the shipping cost value can be updated based on the Ajax service result.



```
${ShippingCost}.setValue(me.getResults())
```

Figure 7-21 Reacting to Ajax service results through control update

7.2.4 UI-only controls (controls with no data bindings)

There are many instances where controls on a UI have no meaningful relationship with business data. For example:

- Check box or radio buttons that are used for UI-only behavior (for example, used to change aspect of other controls on the UI).
- Data entry confirmation (double-entry).
- Drop-downs or Select controls used for further selection in Select control chain that ultimately lead to a control bound to Business Data (preliminary

search or selection pattern). Only the last Select in this chain can have a meaningful data binding.

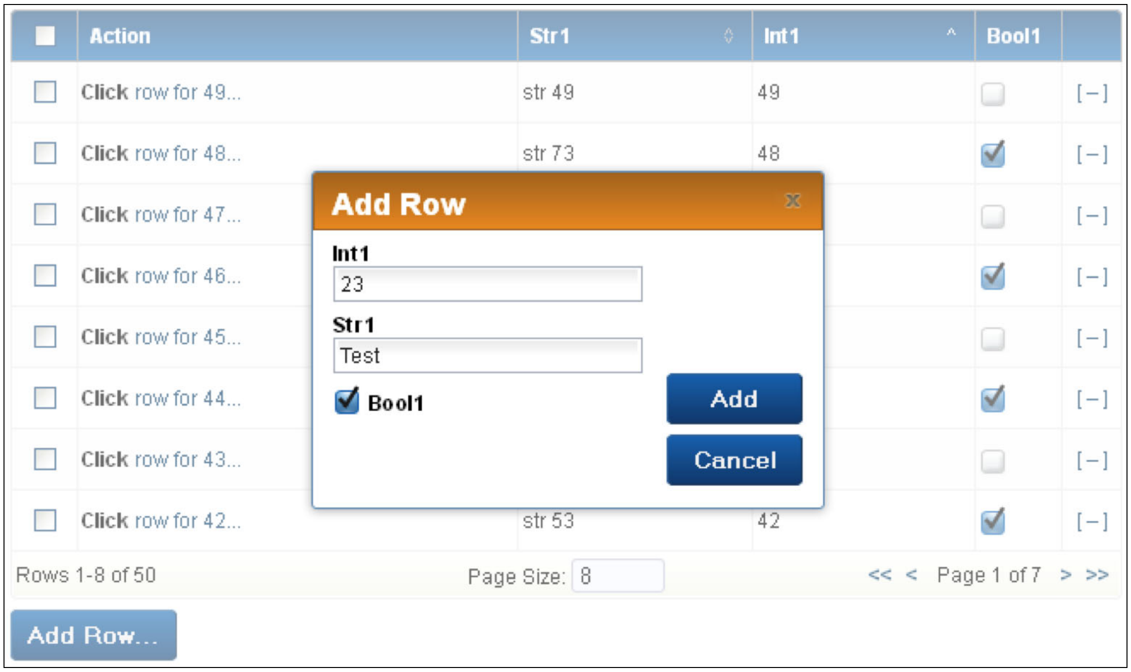
- Dialogs for data entry of line items in a table (the table and its associated fields are bound to data, but the line item entry dialog has no need for bound data).
- Informative fields (for example a line item price) that are presented to the user “on the glass” but are not backed by business data because they are computed at a later stage, or not at all, on the server.

Coach NG controls typically require to be bound to data in order to function properly. This often causes the proliferation of utility Business Data definitions and holder types to accommodate the various “helper” data elements that need to be created.

A specific advantage of EmeriConVIEWS is the ability to use controls without binding them to data. Any EmeriConVIEWS control that does not represent meaningful business data still works as a “first-class” citizen (meaning events, formulas, properties, method remain fully functional) on a Coach Page or a subview.

This feature can significantly reduce the number of “helper” artifacts to be created for UIs in a BPM solution. This leads to a reduced effort, fewer opportunities for errors, and a solution that is ultimately easier to maintain and enhance in the future.

Figure 7-22 shows a simple “table line item add” dialog scenario.



☐	Action	Str1	Int1	Bool1	
☐	Click row for 49...	str 49	49	☐	[−]
☐	Click row for 48...	str 73	48	☑	[−]
☐	Click row for 47...			☐	[−]
☐	Click row for 46...			☑	[−]
☐	Click row for 45...			☐	[−]
☐	Click row for 44...			☑	[−]
☐	Click row for 43...			☐	[−]
☐	Click row for 42...	str 53	42	☑	[−]

Rows 1-8 of 50 Page Size: 8 << < Page 1 of 7 > >>

Add Row...

Add Row

Int1: 23

Str1: Test

☑ Bool1

Add

Cancel

Figure 7-22 Example of Coach Page using “unbound” controls

Figure 7-23 shows the same Coach Page in Process Designer. In this scenario, the "Add Row" dialog only exists to add a row to a table and has no particular business meaning beyond adding a row. Accordingly, the Integer, Text, and Check box controls have no data binding:

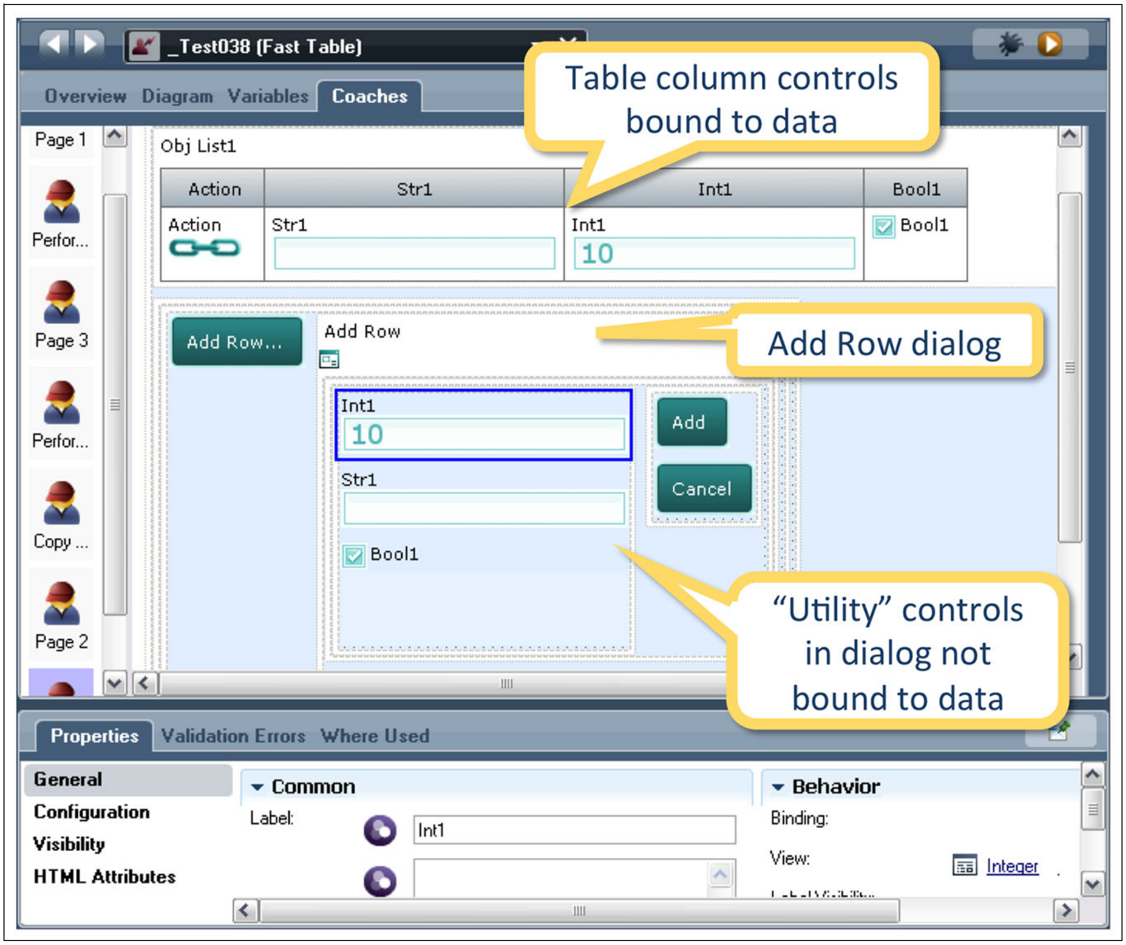
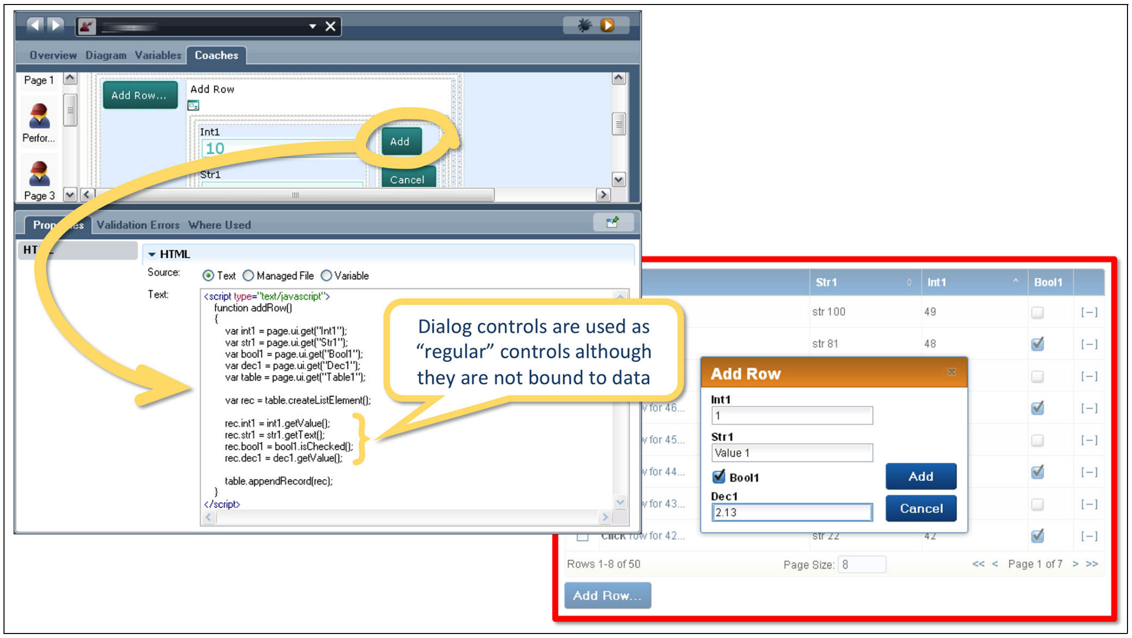


Figure 7-23 Process Designer view of Coach Page using "unbound" controls

Figure 7-24 shows how those “unbound” controls are still fully usable in business logic to add a row to the table.


Overview Diagram Variables Coaches
Page 1
Perfor...
Page 3
Add Row...
Add Row
Int1
10
Str1
Add
Cancel
Properties Validation Errors Where Used
HTML
Dialog controls are used as “regular” controls although they are not bound to data
Str1
Int1
Bool1
Add Row
Int1
Str1
Bool1
Dec1
Add
Cancel
Rows 1-8 of 50
Page Size: 8
Page 1 of 7
Add Row...

Source: (● Text) ○ Managed File ○ Variable

Text:

```
<script type="text/javascript">
   function addRow()
   {
      var int1 = page.ui.get("Int1");
      var str1 = page.ui.get("Str1");
      var bool1 = page.ui.get("Bool1");
      var dec1 = page.ui.get("Dec1");
      var table = page.ui.get("Table1");

      var rec = table.createListElement();

      rec.int1 = int1.getValue();
      rec.str1 = str1.getText();
      rec.bool1 = bool1.isChecked();
      rec.dec1 = dec1.getValue();

      table.appendRecord(rec);
   }
</script>
```

Figure 7-24 Example of exploiting “unbound” controls in business logic

7.2.5 Handling validation

Validation of entered data or user selections is a very common UI requirement. Sometimes it is appropriate to perform validation:

1. On the server after a Coach Page has been submitted.
2. On the server without submitting the Coach Page – through Ajax (meaning without performing a perceptible navigation from the page).
3. On the client (meaning on the web browser), without any interaction with the server.

Note: The delineation between which validation *should* be performed by server versus client logic is beyond the scope of this discussion.

The Coach NG framework provides good options “out-of-the-box” for the first two previously mentioned types of validation. Server-side validation is typically useful for situations where “back-end” resources, such as data, rules, or other services are needed to evaluate data quality, validity, and business-semantic completeness.

The case for client-side validation

The third (client-side) validation type is usually for simpler situations where, for example:

- An order-shipping form should not be submitted if the shipping address is missing
- An end date is earlier than a start date on a form and needs to be immediately detected and flagged to the user (including an explanation of the problem)

Other considerations might include:

- Validation request volumes that are significant enough to justify server load mitigation through client-side logic whenever possible
- Unreliable or slower connections with mobile devices that justify limiting server requests to minimize usability and responsiveness impacts

Client-side validation allows more immediate visual feedback and provides more interactive and responsive safeguards to a user. Client-side validation is a common requirement. Aside from proactive capabilities that constrain user input in some way, Coach NG does not readily provide client-side validation capability (at least not without effort).

Coach NG out-of-the-box includes default rendering capability for many controls (Text, Select, Tabs, and so on). For example, a Text control flagged as invalid through out-of-the-box Coach NG renders accordingly. See Figure 7-25 on page 263.

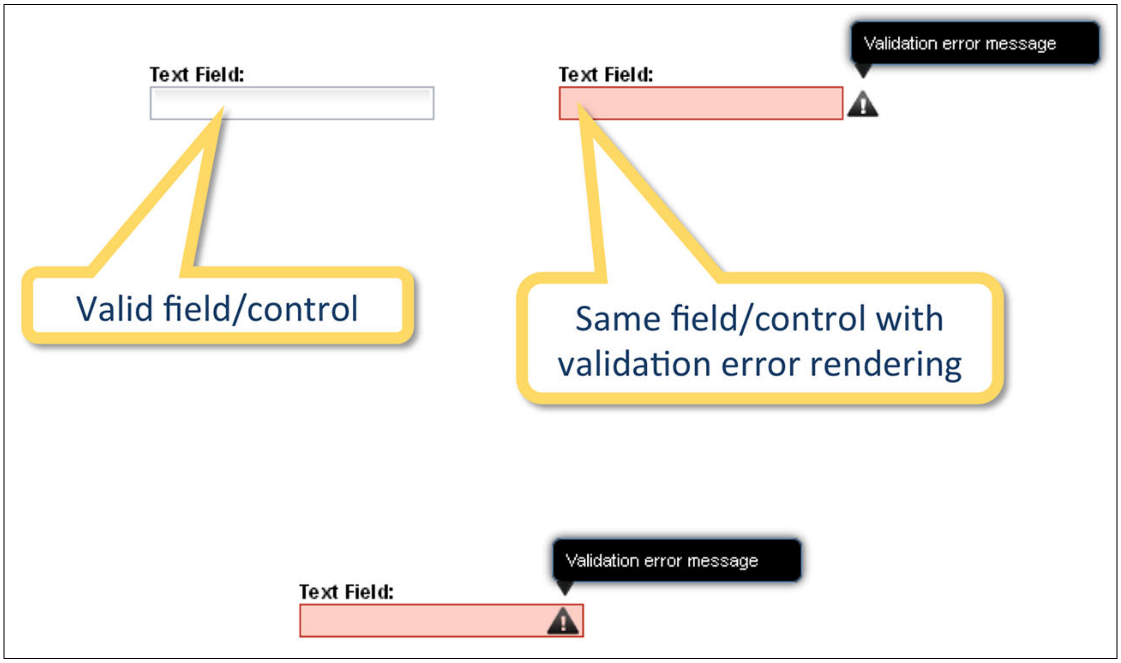


Figure 7-25 Out-of-the-box Coach NG validation error rendering example

EmeriConVIEWS uses the Coach NG validation rendering mechanism to manifest validation errors to the user as shown in Figure 7-25. The key difference is how readily the validation mechanism is exploitable for client-side validation purposes. The next few examples illustrate how EmeriConVIEWS facilitates client-side validation.

Button click-triggered examples

These next few examples show three different ways of implementing basic client-side validation combined with navigation control on a button click event.

- Example 1:

  In this first example, validation is handled in a custom manner by displaying an error message on the Coach Page upon validation failure (in this particular scenario, validation errors are manifested in a nonstandard way compared to normal Coach NG behavior).

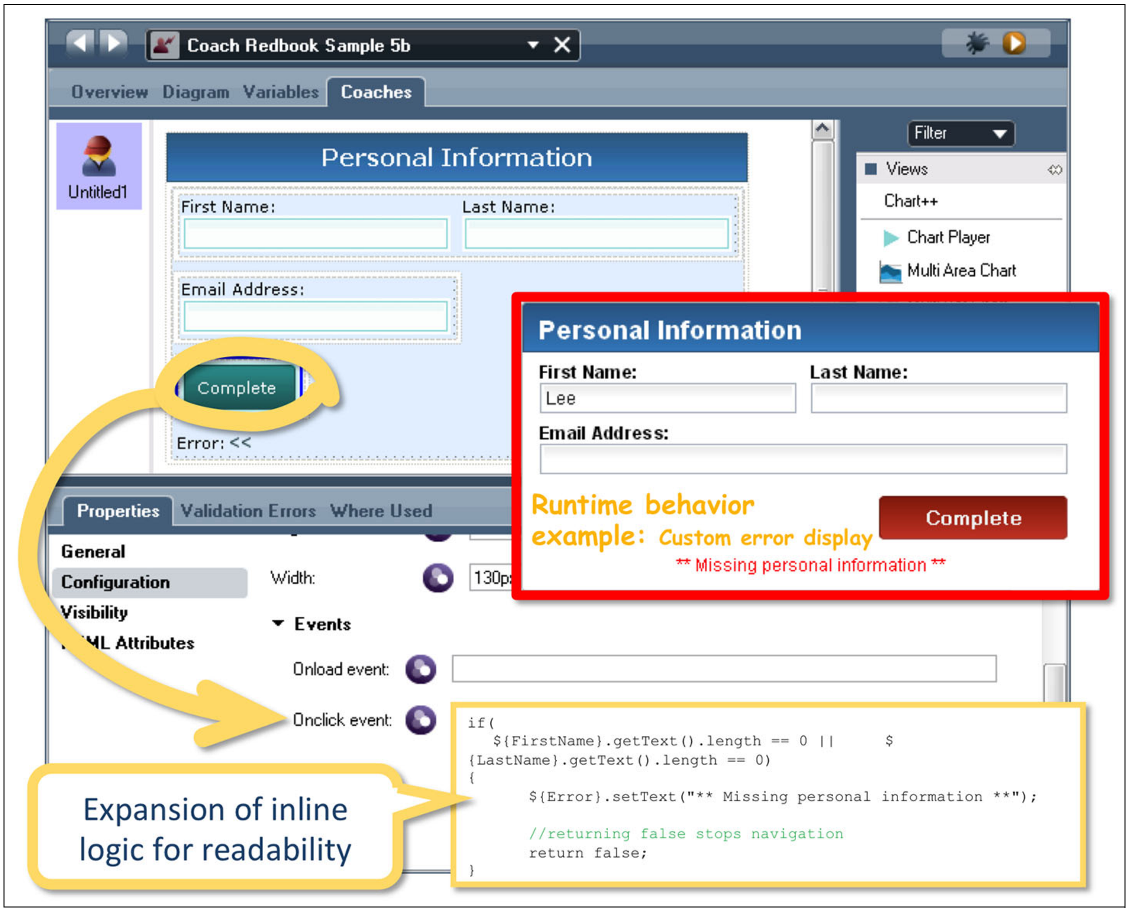
  The validation is triggered on click of the “Complete” button. The Coach Page contains the following relevant fields:

  – Control type: Text; Control id: FirstName
  – Control type: Text; Control id: LastName
  – Control type: Output Text; Control id: Error

– Control type: Button; Control id: *<irrelevant for this example>*

Basic logic: Onclick of button 1) The length of the text entered in the FirstName and LastName fields is checked. 2) If one or both contain no data, the text of the Error field is set to display "** Missing personal information **".

Figure 7-26 shows this approach in the context of Process Designer.



```
if(
   ${FirstName}.getText().length == 0 ||      $
{LastName}.getText().length == 0)
{
      ${Error}.setText("** Missing personal information **");

      //returning false stops navigation
      return false;
}
```

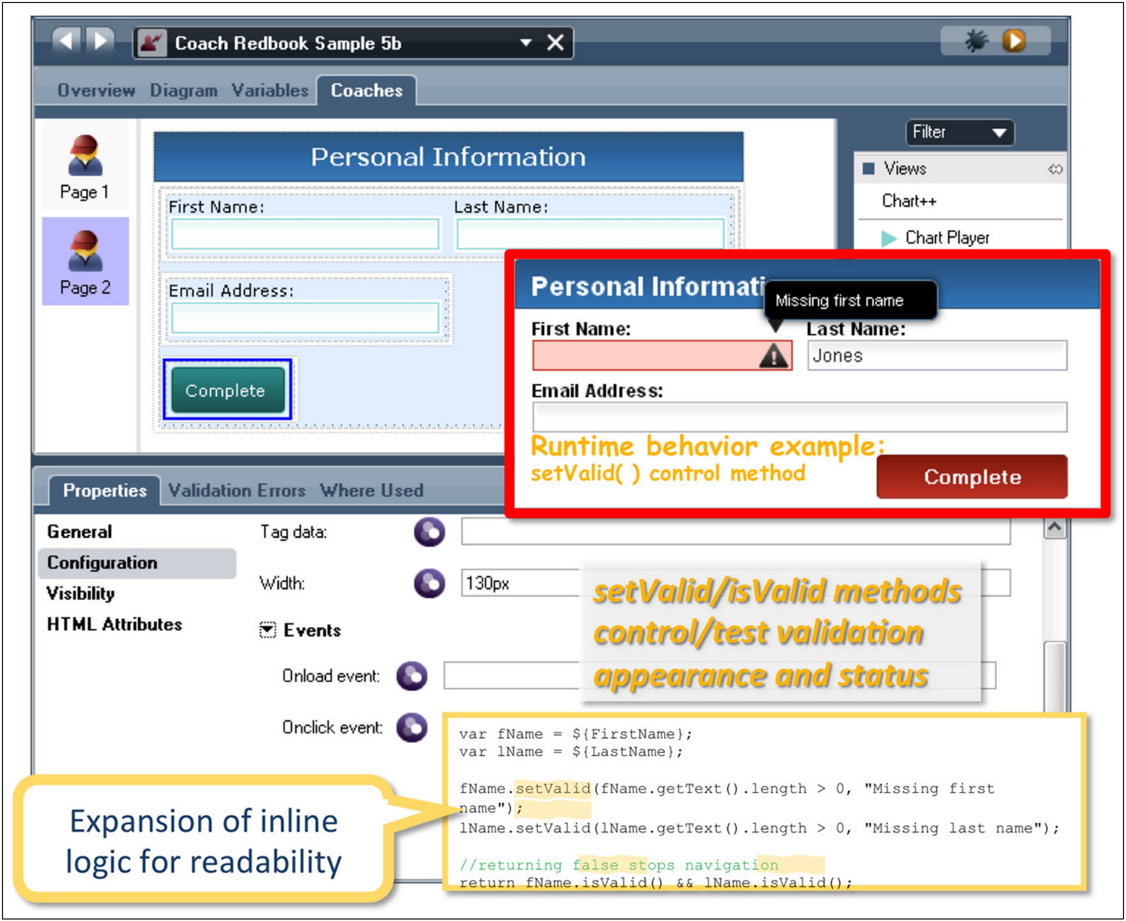
Figure 7-26 Validation example #1. Custom error display on failed validation: Inline validation logic

- Example 2:

  The second example exploits the visual validation aspects of Coach NG through EmeriConVIEWS-specific methods. Most EmeriConVIEWS controls support the following methods:

  – `setValid(booleanFlag, msgIfInvalid)`
  – `isValid()`

These methods can be used to set or get the client-side validation status of a control. If `false` is passed to `setValid()`, the message also passed is displayed. Figure 7-27 shows how both `setValid()` and `isValid()` are used in the button onclick validation logic.



```
var fName = ${FirstName};
var lName = ${LastName};

fName.setValid(fName.getText().length > 0, "Missing first
name");
lName.setValid(lName.getText().length > 0, "Missing last name");

//returning false stops navigation
return fName.isValid() && lName.isValid();
```

Figure 7-27 Validation example #2. Built-in validation activation with inline client-side logic

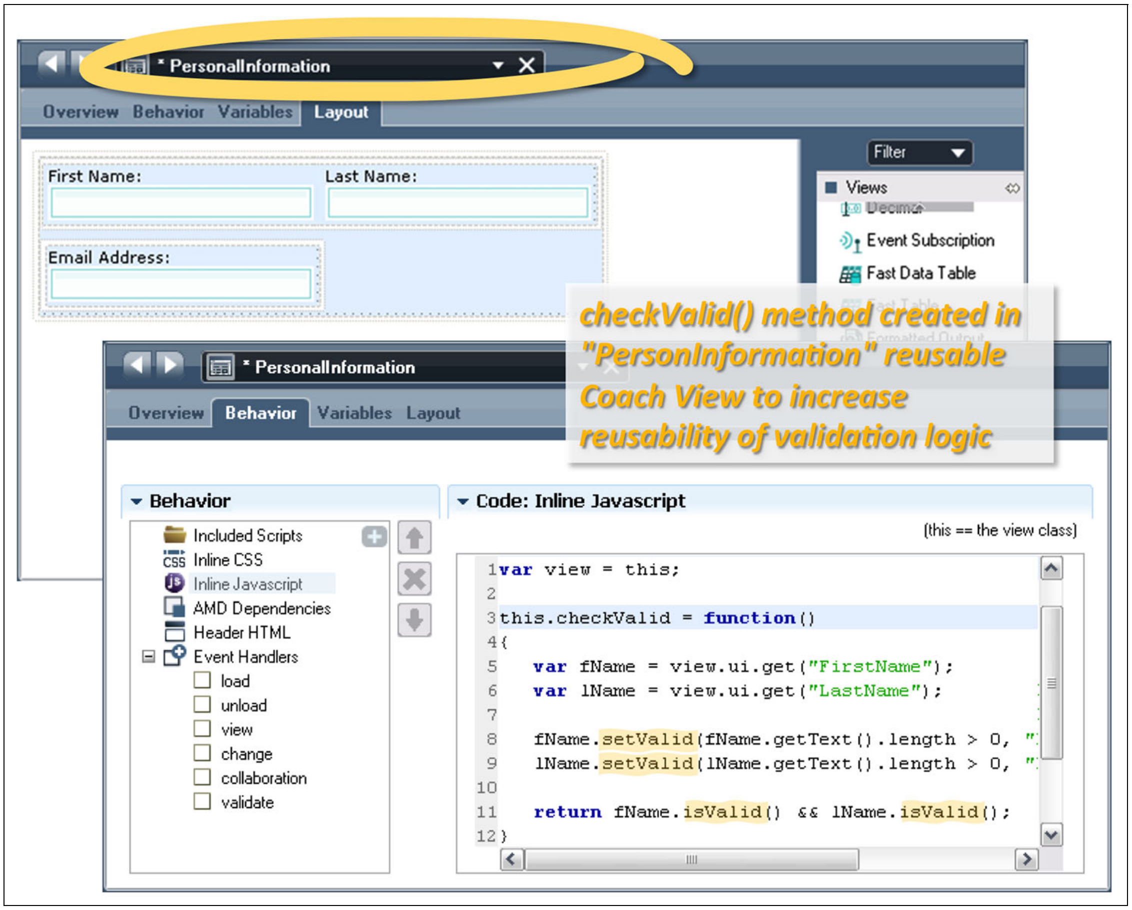
The logic above works as follows:

- The first `setValid` receives false as its first argument because the length of first name is 0, which causes a validation error rendering of the Text control, showing the “Missing first name” tooltip
- The second `setValid` receives true as its first argument because length of last name > 0, which renders the Text control as validation error-free, therefore the “Missing last name” tooltip is not shown

– Lastly, the returning statement checks on the last valid state of each control with `isValid`, which is `false` for first name and `true` for last name, resulting in a logical `false` which in turns prevents the button from triggering the Boundary Event

- Example 3:

  The third case shows how to encapsulate validation logic into a reusable Coach View with a “Personal Information” example as shown in Figure 7-28. A `checkValid` method is defined on the view.

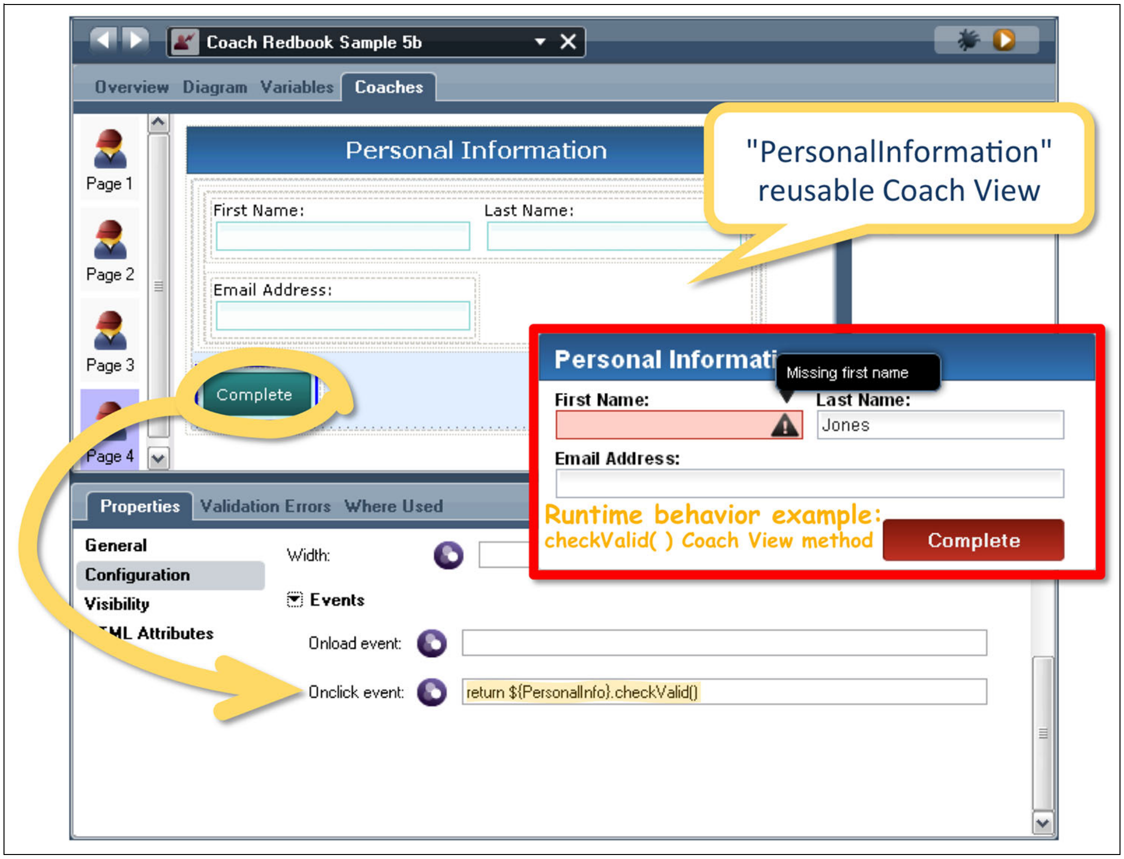


checkValid() method created in "PersonInformation" reusable Coach View to increase reusability of validation logic

```
var view = this;

this.checkValid = function()
{
   var fName = view.ui.get("FirstName");
   var lName = view.ui.get("LastName");

   fName.setValid(fName.getText().length > 0, "
   lName.setValid(lName.getText().length > 0, "

   return fName.isValid() && lName.isValid();
}
```

Figure 7-28 Validation example #3A. Encapsulating validation in reusable Coach View: Logic definition

The `checkValid` method is then used in the event handling logic for button click at the Coach Page level to both trigger the display of invalid data hints and to prevent Coach Page navigation if `checkValid` returns false as shown in Figure 7-29.



```
return ${PersonalInfo}.checkValid()
```

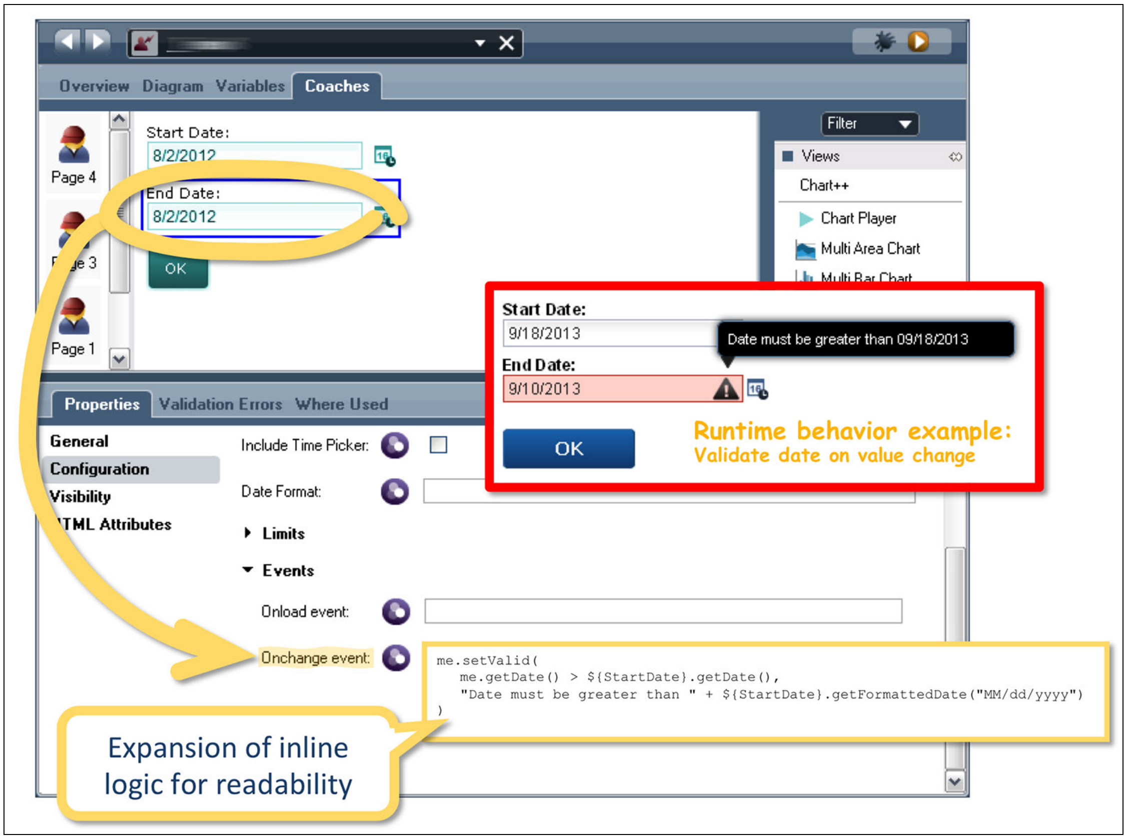
Figure 7-29 Validation example #3A. Reusing Coach View validation logic in Coach Page

Other Event-Triggered Examples

Button clicks are not the only triggers of validation with EmeriConVIEWS. Validation (custom or with the `setValid` method) can be triggered by any event on a Coach Page or Coach View.

Often (assuming there is sufficient context), providing validation feedback as close to the problem as possible can provide a better user experience and efficiency.

Figure 7-30 shows how an End Date control can be validated on data entry against the value of a Start Date control to ensure that the End Date is at least one day later than the Start Date.



```
me.setValid(
   me.getDate() > ${StartDate}.getDate(),
   "Date must be greater than " + ${StartDate}.getFormattedDate("MM/dd/yyyy")
)
```

Figure 7-30 Immediate validation trigger example. Validating input on control value changes

To make this particular solution complete, one could also apply similar logic upon changing the Start Date, either with similar inline logic, or by having `onchange` events for both Start Date and End Date call a common validation function. Alternatively, the StartDate `onchange` event could simply clear the End Date by calling:

```
${EndDate}.setDate(null)
```

Server-side validation

EmeriConVIEWS is fully compatible with the Coach NG server-side validation mechanism and does not interfere or modify this capability in any way. Coach Pages or Coach Views that use EmeriConVIEWS work unmodified with out-of-the-box validation firing, Stay On Page events, the `tw.system.coachValidation` object and the `tw.system` methods to add, clear, remove, and update validation status. The topic of server-side validation for Coaches is addressed in greater detail earlier in this publication.

7.2.6 Sampling of EmeriConVIEWS controls

EmeriConVIEWS provides all the standard (out-of-the-box) Coaches Toolkit controls with specific extensions for addressing, event handling, formula support, and general ease of use enhancements through additional configuration options and methods.

This section provides a high-level sampling of a few of the additional EmeriConVIEWS controls to further enhance the effectiveness of BPM UI Authors and the experience of BPM UI end-users.

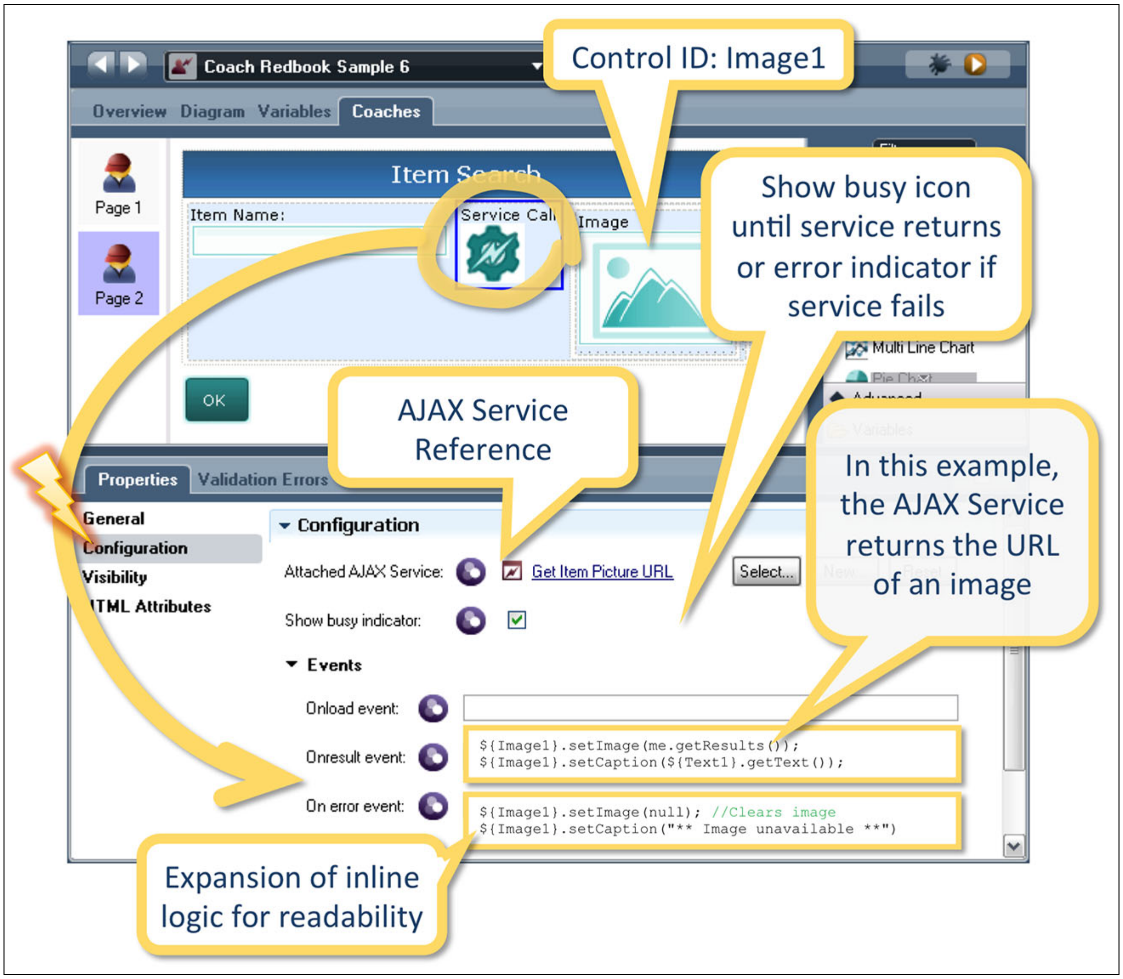
Service Call

The Service Call control provides a way to make Ajax service calls more intuitively and flexibly in an authoring environment like Process Designer.

Note: This control is presented first because other examples later in this section use it.

Usage Example

Figure 7-31 on page 270 shows how the URL and Caption of an Image control are set when the invocation of a Service Call control returns (asynchronously).



Configuration

Attached AJAX Service: Get Item Picture URL

Show busy indicator: ✓

Events

Onload event:

Onresult event:

```
${Image1}.setImage(me.getResults());
${Image1}.setCaption(${Text1}.getText());
```

On error event:

```
${Image1}.setImage(null); //Clears image
${Image1}.setCaption("** Image unavailable **")
```

Figure 7-31 Basic UI interactions for Service Call response example

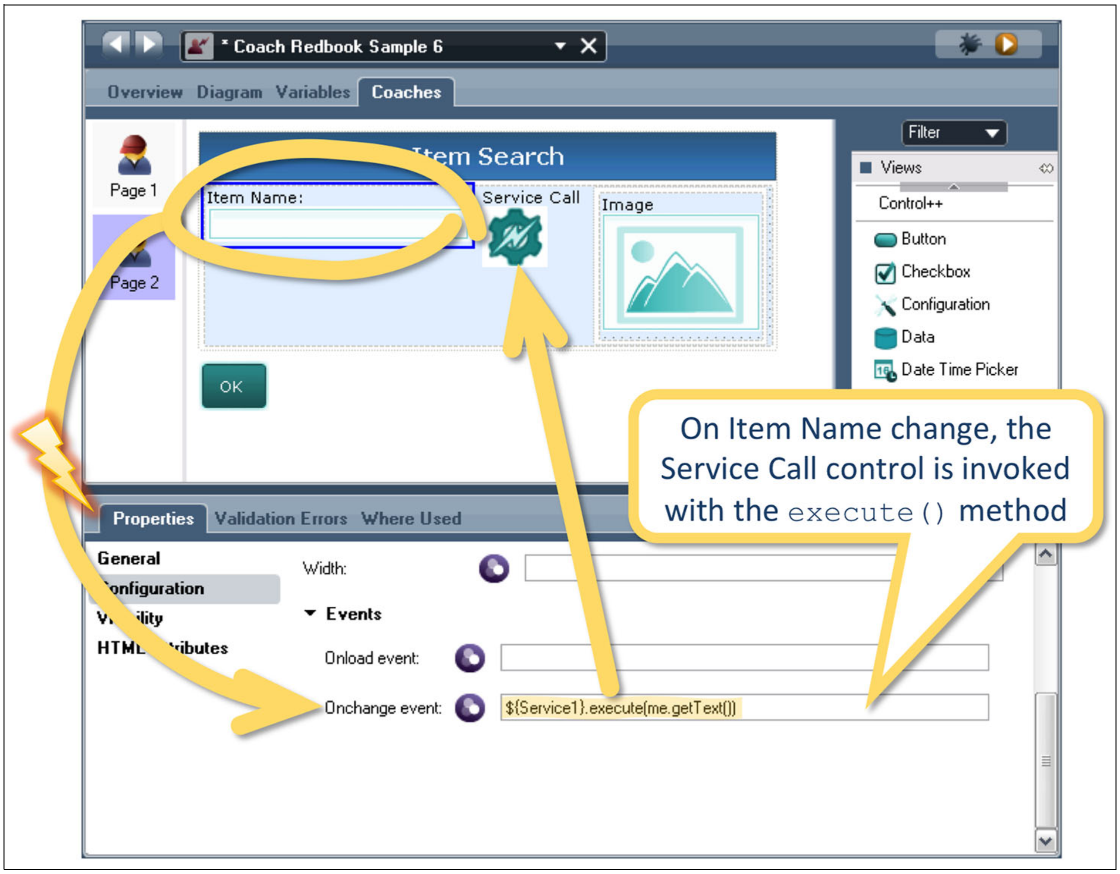
Upon successful return, the *on result* event is triggered, which activates the logic to set the image URL to the URL returned by the Ajax service (and sets the caption of the image to the item name). If the service invocation returns an error, the *on error* event is triggered instead, which clears the image and sets the caption to the text “** Image unavailable **”.

The Service Call control method `getResults()` provides the response from the Ajax service. If the response contains a complex type, the complex type is navigated by simply accessing the additional attributes of the returned object.

For example, assuming a returned variable of type Customer containing the attributes name and age, the result data could be extracted as follows:

```
var res = ${Service1}.getResults();
var custName = res.name;
var custAge = res.age;
```

Figure 7-31 on page 270 shows how the UI reacts based on the response from an Ajax service. But how is the Service Call control invoked or triggered in the first place? Figure 7-32 shows how a change in the Item Name Text control triggers the invocation of the Service Call control.



On Item Name change, the Service Call control is invoked with the `execute()` method

Onchange event: `${Service1}.execute(me.getText())`

Figure 7-32   Basic UI interactions for Service Call request example

Note how the data passed to the `execute()` method comes from the Item Name Text control. The data passed to the `execute()` method is sent directly (as JSON) to the Ajax service. The type of the data can be simple or complex. If the data is complex, the structure and attribute names must match exactly those of the input complex type expected by the Ajax service. Figure 7-33 shows the behavior of the example at run time.

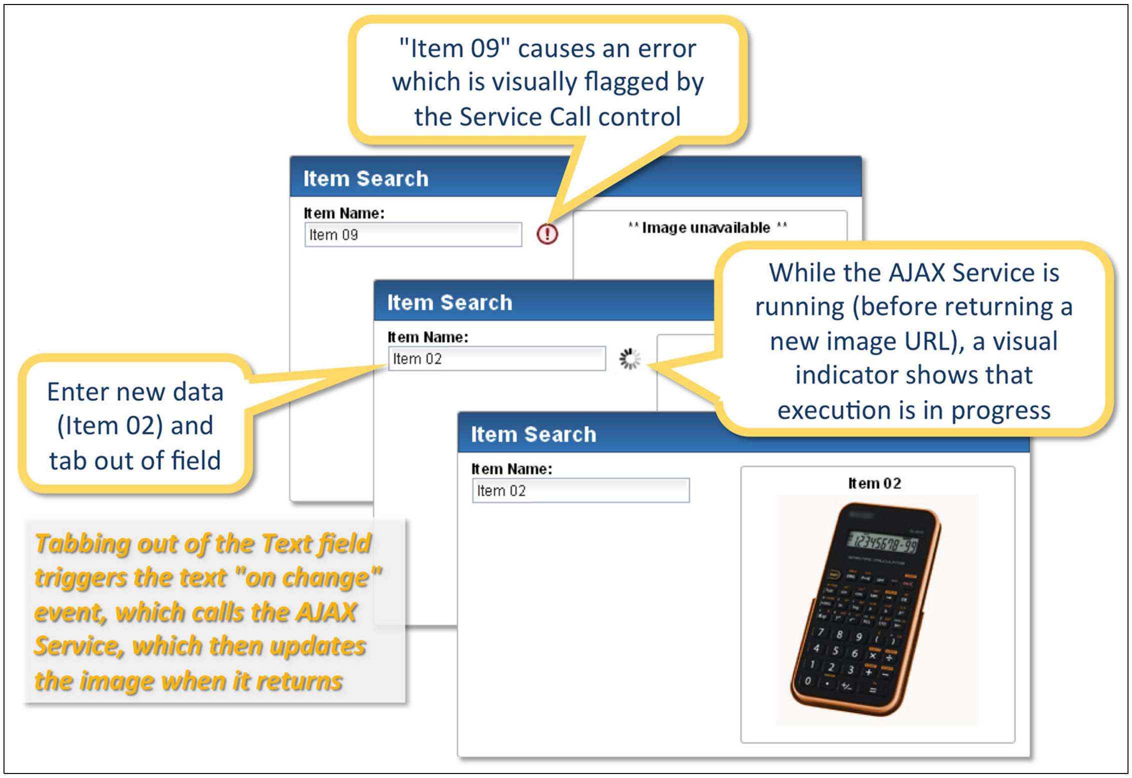

"Item 09" causes an error which is visually flagged by the Service Call control
Item Search
Item Name:
Item 09
** Image unavailable **
Item Search
Item Name:
Item 02
While the AJAX Service is running (before returning a new image URL), a visual indicator shows that execution is in progress
Enter new data (Item 02) and tab out of field
Item Search
Item Name:
Item 02
Item 02
Tabbing out of the Text field triggers the text "on change" event, which calls the AJAX Service, which then updates the image when it returns

Figure 7-33 Runtime behavior of Service Call example

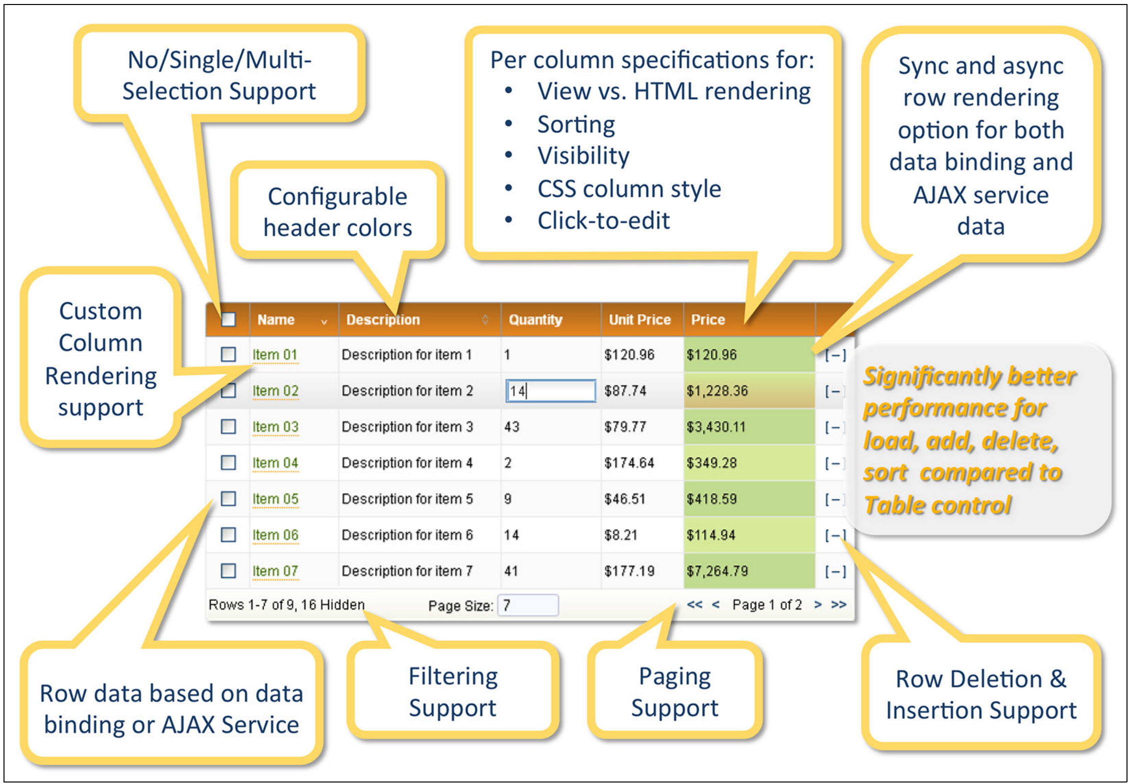
Good practices

Take a look at some good practices.

- Using event handlers from various visual controls (for example Text, Select, Radio Buttons, Date Time Picker, and so on) to call the Service Call control can create very interactive pages that mix client-side and server-side logic
- Although not the default, using the *Show busy indicator* option provides an end-user visual feedback that the service is executing (or encountered a problem) without disrupting the user experience. Omitting the visual indicator may leave the end-user guessing unless the progress of the Ajax service is manifested in some other way
- The Service Call control can also be used to effectively and straightforwardly perform server-side validation that is based on situations that would not typically trigger a Boundary Event

Fast Table

The Fast Table control provides a high-performance alternative to the Dojo-based standard Table control, with support for sorting, paging, row-based searching, or filtering. A sample rendering of the Fast Table control, with feature overview, is shown in Figure 7-34.


No/Single/Multi-Selection Support
Configurable header colors
Per column specifications for:
• View vs. HTML rendering
• Sorting
• Visibility
• CSS column style
• Click-to-edit
Sync and async row rendering option for both data binding and AJAX service data
Custom Column Rendering support
Significantly better performance for load, add, delete, sort compared to Table control
Row data based on data binding or AJAX Service
Filtering Support
Paging Support
Row Deletion & Insertion Support

	Name	Description	Quantity	Unit Price	Price	
☐	Item 01	Description for item 1	1	$120.96	$120.96	[−]
☐	Item 02	Description for item 2	14	$87.74	$1,228.36	[−]
☐	Item 03	Description for item 3	43	$79.77	$3,430.11	[−]
☐	Item 04	Description for item 4	2	$174.64	$349.28	[−]
☐	Item 05	Description for item 5	9	$46.51	$418.59	[−]
☐	Item 06	Description for item 6	14	$8.21	$114.94	[−]
☐	Item 07	Description for item 7	41	$177.19	$7,264.79	[−]

Rows 1-7 of 9, 16 Hidden   Page Size: 7   << < Page 1 of 2 > >>

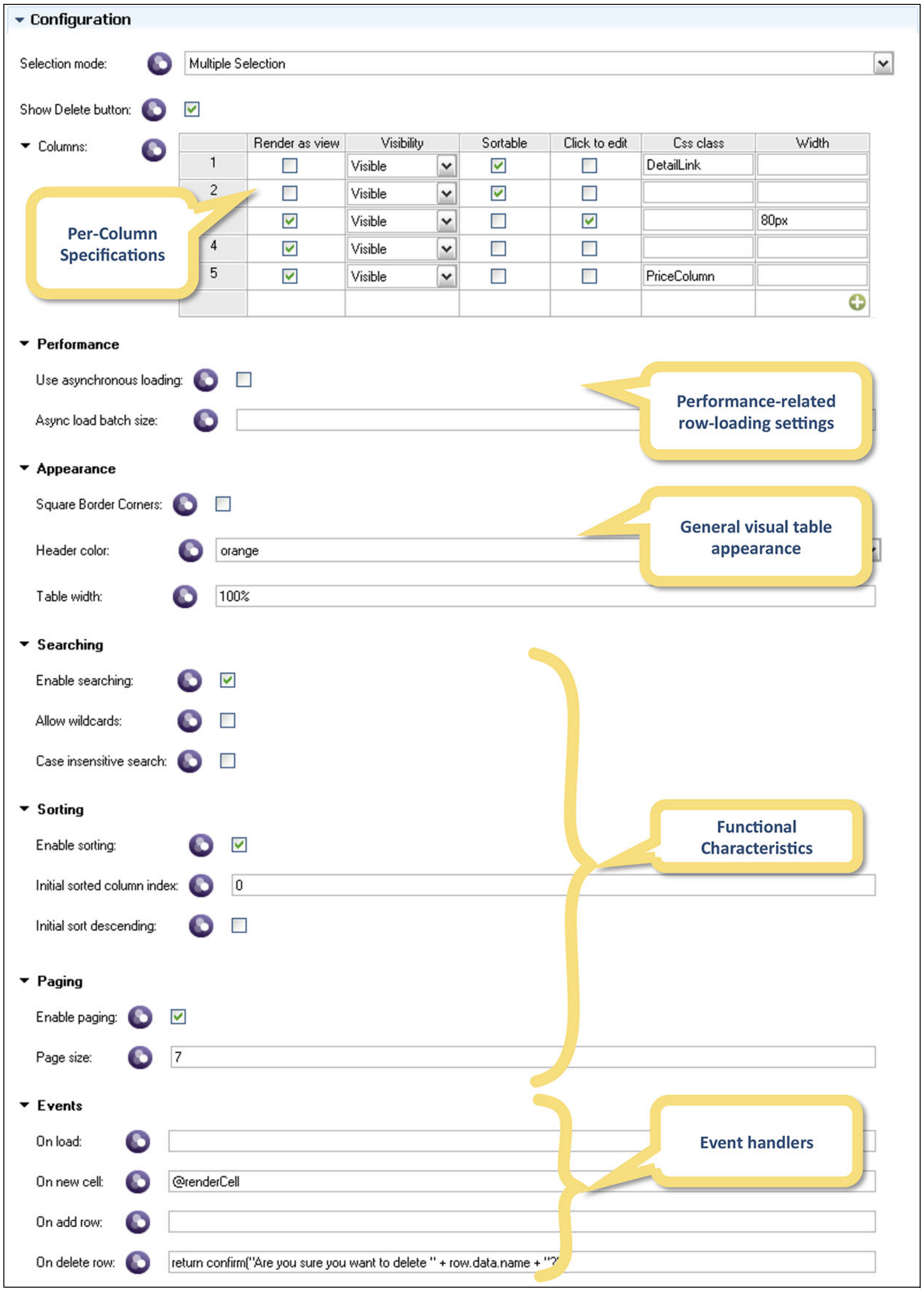
Figure 7-34 Fast Table control at a glance

To display data from Ajax Services, use the Fast Data Table control, which is the Ajax counterpart for Fast Table.

Configuration options

The Fast Table controls provide a number of configuration options to control visual aspects, functional and performance capabilities, and to handle key table control events.

Figure 7-35 on page 274 expands the Fast Table properties as seen in Process Designer.


Per-Column Specifications
Performance-related row-loading settings
General visual table appearance
Functional Characteristics
Event handlers

Configuration

Selection mode: Multiple Selection

Show Delete button: ☑

Columns:

	Render as view	Visibility	Sortable	Click to edit	Css class	Width
1	☐	Visible	☑	☐	DetailLink	
2	☐	Visible	☑	☐		
	☑	Visible	☐	☑		80px
4	☑	Visible	☐	☐		
5	☑	Visible	☐	☐	PriceColumn	

Performance

Use asynchronous loading: ☐

Async load batch size:

Appearance

Square Border Corners: ☐

Header color: orange

Table width: 100%

Searching

Enable searching: ☑

Allow wildcards: ☐

Case insensitive search: ☐

Sorting

Enable sorting: ☑

Initial sorted column index: 0

Initial sort descending: ☐

Paging

Enable paging: ☑

Page size: 7

Events

On load:

On new cell: @renderCell

On add row:

On delete row: return confirm("Are you sure you want to delete " + row.data.name + "?

Figure 7-35 Fast Table configuration options

Performance options

Tabular data can be expensive to display. The sample measurements shown in Table 7-1 contrast the cost of rendering 100 rows over five columns between the regular Table control and the Fast Table control.

To keep the comparison simple, Fast Table rendering was performed with all columns displayed as Views and then with all columns displayed as “Simple HTML” Cells.

Table 7-1 Sample measurements

	Full load		**Delete row**		**Add row**	
100 rows/5 columns	Time (sec)	1000 Calls	Time (sec)	1000 Calls	Time (sec)	1000 Calls
Regular Table ControlRegular Table Control	94	3520	43	1710	46	1733
Fast Table – View Rendering	28	1360	0.46	2.17	0.51	17
Fast Table – Simple HTML Rendering	0.98	52	0.14	0.44	0.09	5.9

If many columns need to be rendered as Coach Views, the cost of rendering a single row could still be high. Two options are available to mitigate the impact of rendering an expensive row on end-user experience:

- Paging (which is also available for the standard Table control) reduces the overall cost of rendering the table by only displaying a (hopefully small) subset of the rows
- In cases where, for business reasons, paging might not be an option, Fast Table provides a tunable asynchronous load mode (see *Use asynchronous loading* and *Async load batch size* settings), which fully preserves browser responsiveness even while loading a very large row set

Usage example

In the Fast Table example in Figure 7-36 on page 276, a table containing line items is displayed containing item name, description, quantity, unit price, and computed line item price. Clicking the name of an item in the table displays a dialog with basic item details and a picture of the item, the URL of which is retrieved from an Ajax service. This example is built inside a reusable Coach View named OrderTable (although it could just as well have been built directly at the Coach Page level).

Figure 7-36 on page 276 built below contains a list of OrderItem objects. OrderItem has the following structure:

- Name (String)
- Description (String)
- Quantity (Integer)
- Unit Price (Decimal)

Figure 7-36 illustrates the previous requirements in context of the Process Designer Coach Editor and the controls placed on the UI.

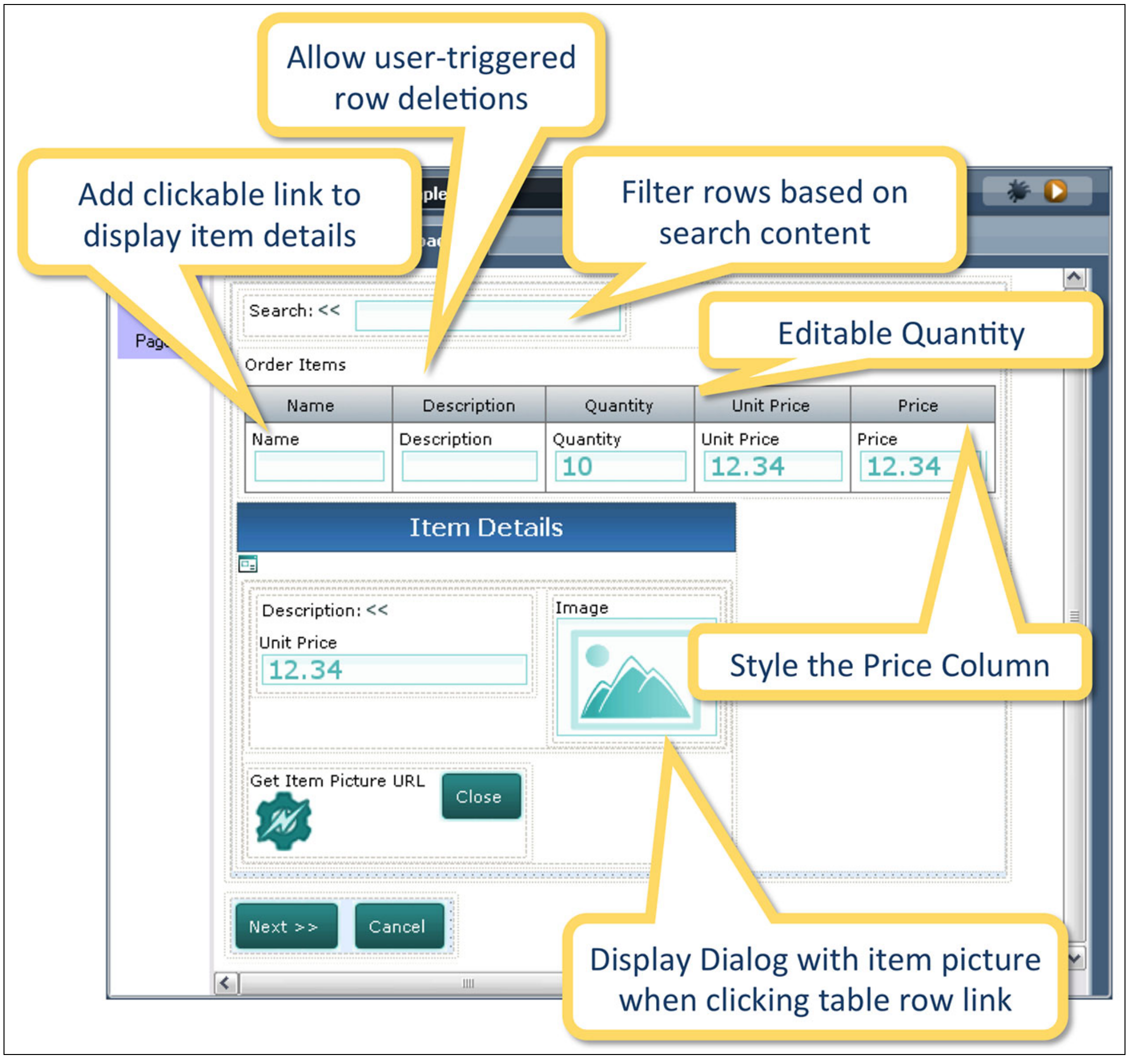


Figure 7-36 Basic requirements for Fast Table example

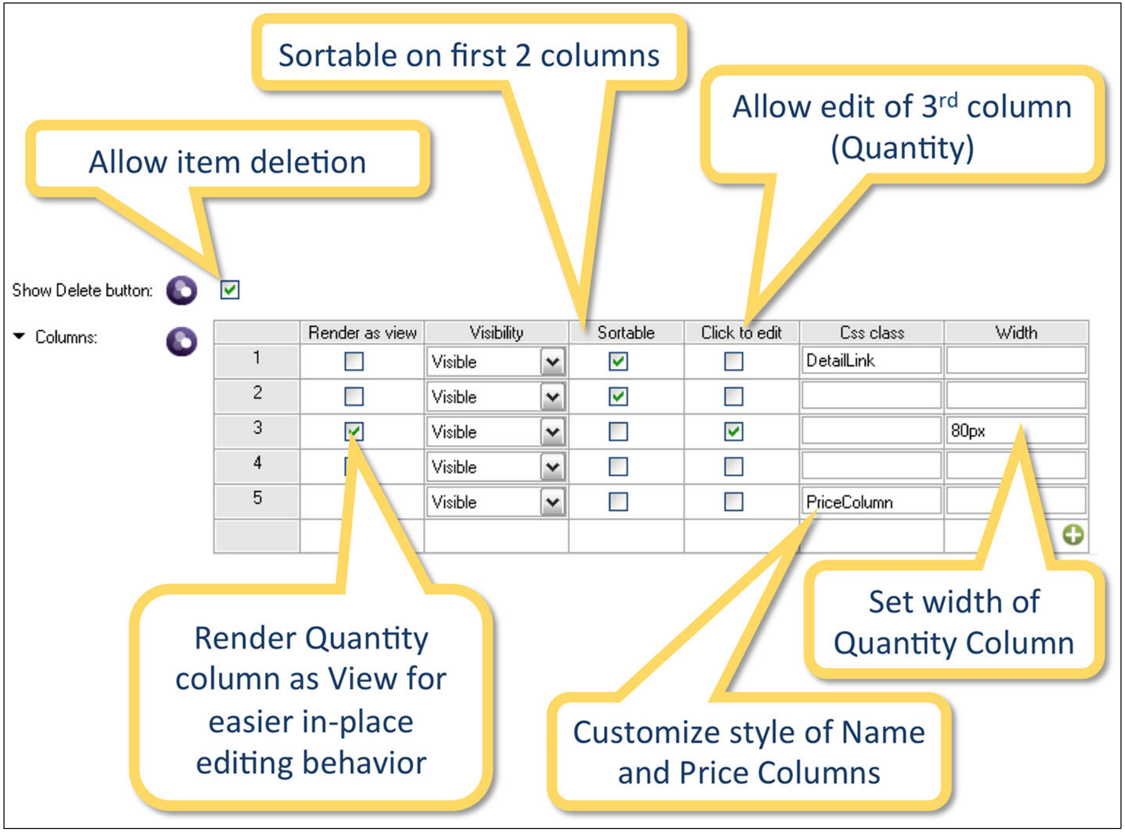
The following Fast Table capabilities are used to implement the preceding requirements:

- Custom-rendering of cells: This is used to render the clickable link (with logic attached to display the item details dialog) in the Name column
- Search feature: To filter in rows that match a certain search text and hide the other rows (the footer of the table indicates how many records are hidden if the record set is filtered)
- Click-to-edit: Similar to the basic Table control. This allows the end-user to click a cell and edit its content. Only the Quantity column should be editable
- Custom-styling of columns by specifying a CSS class on a column, in this case on the Name (first) and Price (fifth) column

The rest of the requirements are implemented through other EmeriConVIEWS controls and aspects. The approach that is used to implement those requirements will also be detailed.

Table configuration

The following section details the configuration options that are specified on the Fast Table control to support the sample requirements as shown in Figure 7-37 and Figure 7-38 on page 279.

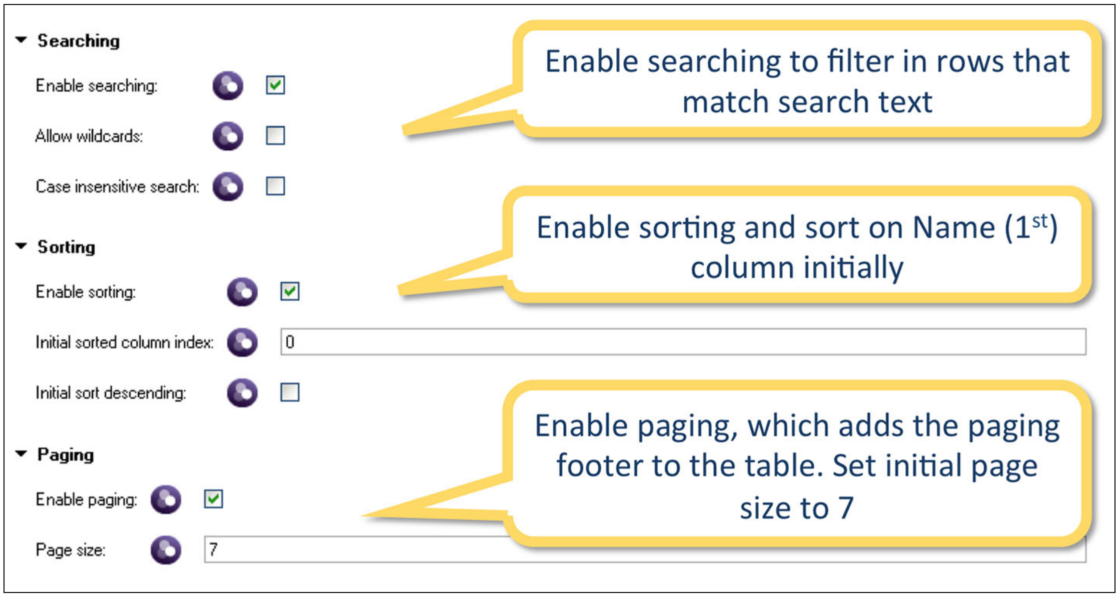

Sortable on first 2 columns
Allow edit of 3rd column (Quantity)
Allow item deletion
Render Quantity column as View for easier in-place editing behavior
Set width of Quantity Column
Customize style of Name and Price Columns

Show Delete button: ✓

Columns:

	Render as view	Visibility	Sortable	Click to edit	Css class	Width
1	☐	Visible	✓	☐	DetailLink	
2	☐	Visible	✓	☐		
3	✓	Visible	☐	✓		80px
4	☐	Visible	☐	☐		
5		Visible	☐	☐	PriceColumn	

Figure 7-37 Configuration of columns for Fast Table example



Searching

Enable searching: ✓

Allow wildcards: ☐

Case insensitive search: ☐

Sorting

Enable sorting: ✓

Initial sorted column index: 0

Initial sort descending: ☐

Paging

Enable paging: ✓

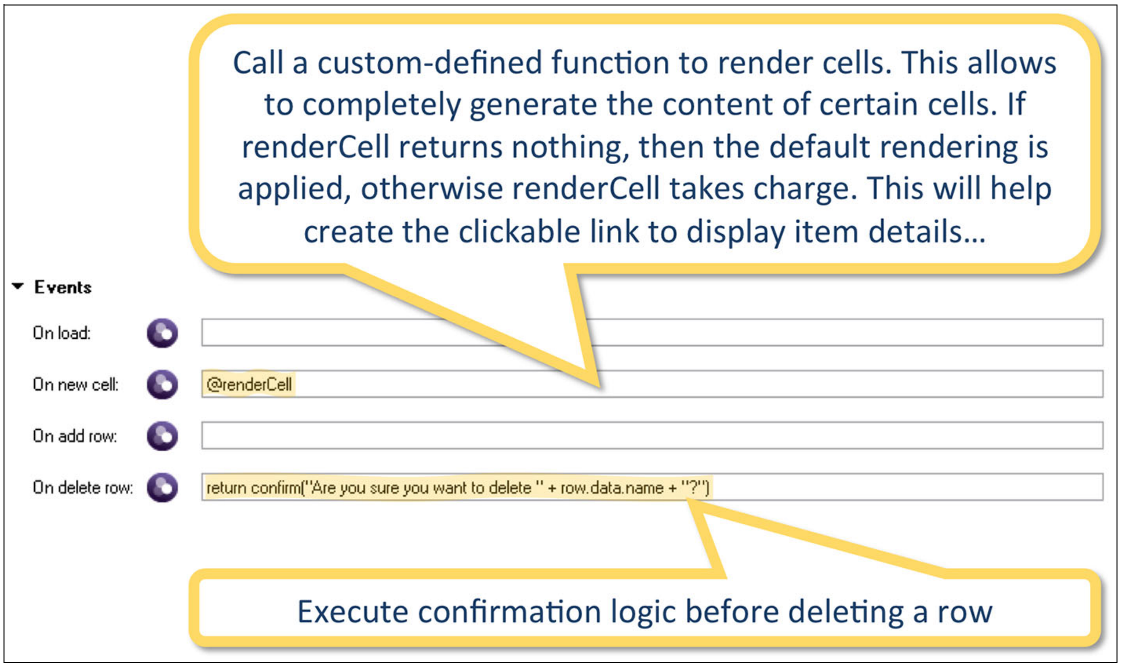
Page size: 7

Enable searching to filter in rows that match search text
Enable sorting and sort on Name (1st) column initially
Enable paging, which adds the paging footer to the table. Set initial page size to 7

Figure 7-38 Configuration of functional table aspects for Fast Table example

This last configuration section manages Fast Table events. In this example, the *On new cell* event is used to allow custom rendering of certain cells in the table (in this case, all cells in the first column).

Additional logic is also added in the *On delete row* event to help prevent the accidental deletion of rows as shown in Figure 7-39.


Call a custom-defined function to render cells. This allows to completely generate the content of certain cells. If renderCell returns nothing, then the default rendering is applied, otherwise renderCell takes charge. This will help create the clickable link to display item details...
Events
On load:
On add row:
Execute confirmation logic before deleting a row

On new cell: @renderCell

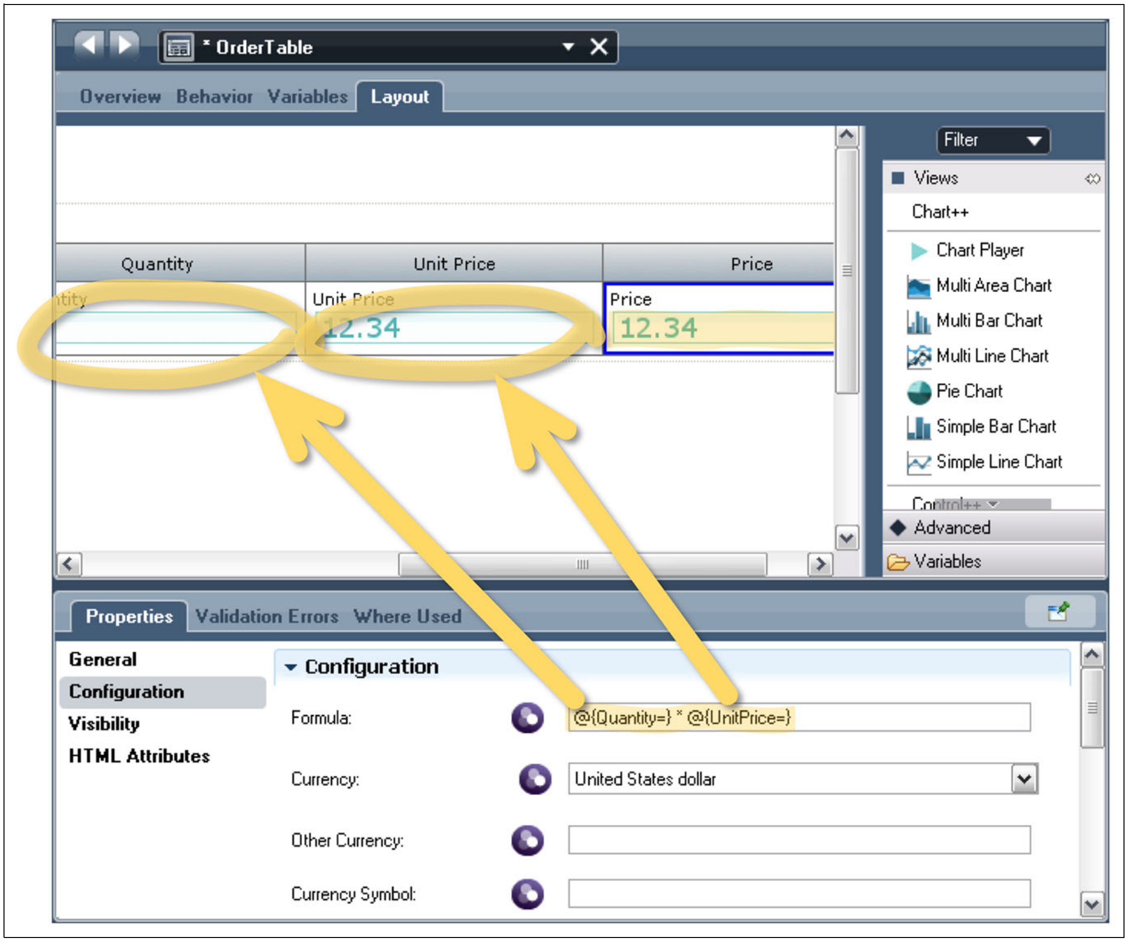
On delete row: return confirm("Are you sure you want to delete " + row.data.name + "?")

Figure 7-39   Configuration of event handling for Fast Table example

Note the use of the row variable in the row deletion event handler. The row variable holds a reference to the record in the table at the position of the deletion. If the table contains a list of a complex type with attributes `name`, `unitPrice`, `description`, `quantity`, those same attributes can be accessed using `row.name`, `row.data.unitPrice`, `row.data.quantity`, and so on.

Price calculations

This section details the price calculation aspects of the example requirements. By now, formula-based calculations should be straightforward: Price = Quantity x Unit Price. This translates in the following formula for the Price control as shown in Figure 7-40.



Configuration

Formula: @{Quantity=} * @{UnitPrice=}

Currency: United States dollar

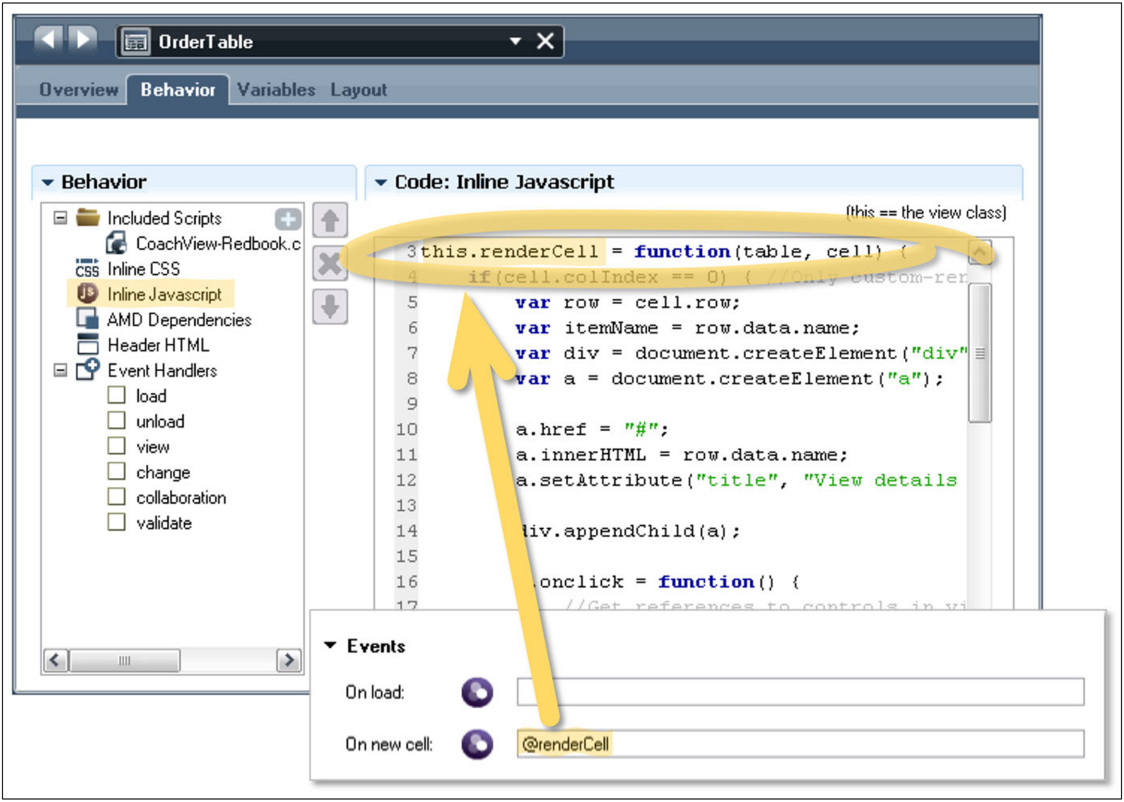
Other Currency:

Currency Symbol:

Figure 7-40 Price calculations for Fast Table example

Custom-rendering of first column

Next is the custom rendering of the first column in the table. The *On new cell* event for the table specifies that a `renderCell` method should be invoked. See Figure 7-41.



```
this.renderCell = function(table, cell) {
    if(cell.colIndex == 0) { //Only custom-ren
        var row = cell.row;
        var itemName = row.data.name;
        var div = document.createElement("div"
        var a = document.createElement("a");

        a.href = "#";
        a.innerHTML = row.data.name;
        a.setAttribute("title", "View details

        iv.appendChild(a);

        .onclick = function() {
            //Get references to controls in vi
```

On load:
On new cell: @renderCell

Figure 7-41   Event handler calling custom cell rendering function for Fast Table example

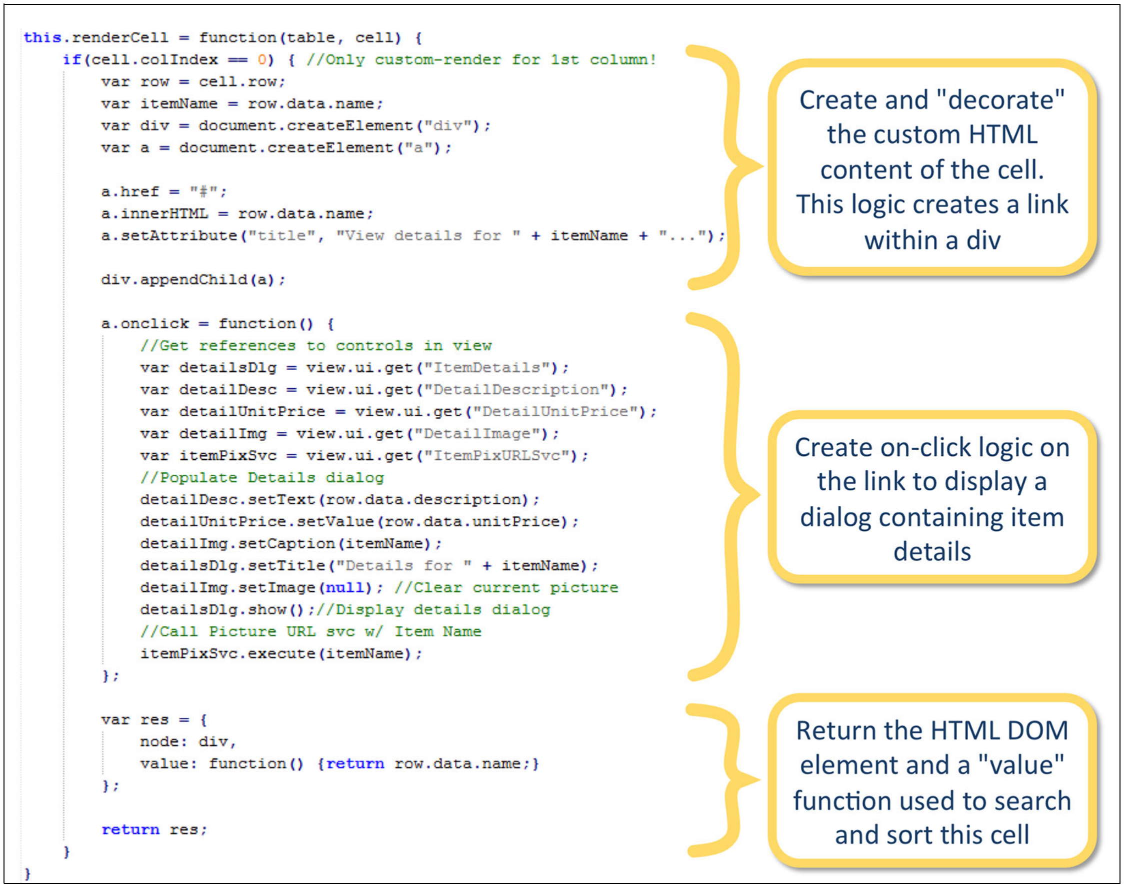
When the *On new cell* event handler calls its handling function, the following arguments are passed to the function and can be used by the logic inside the function:

- Table: The reference to the Fast Table
- Cell: An object containing contextual information for the cell being rendered. Specifically, the cell object contains the following data elements:
  - controlType: The kind of control that is in the table column in the Process Designer editor. In this example, the value would be *Text*
  - colIndex: The index of the column for the cell to be rendered

– parentNode: The Document Object Model (DOM) node reference for the parent table cell

– Row: An object representing the row containing the cell currently being rendered. The row object contains a data object that points to the data record for this particular row. In this example, row.data would contain name, description, quantity, and unitPrice

The full definition of the `renderCell` function (in the Inline JavaScript section of the OrderTable Coach View) is provided in Figure 7-42.



```
this.renderCell = function(table, cell) {
    if(cell.colIndex == 0) { //Only custom-render for 1st column!
        var row = cell.row;
        var itemName = row.data.name;
        var div = document.createElement("div");
        var a = document.createElement("a");

        a.href = "#";
        a.innerHTML = row.data.name;
        a.setAttribute("title", "View details for " + itemName + "...");

        div.appendChild(a);

        a.onclick = function() {
            //Get references to controls in view
            var detailsDlg = view.ui.get("ItemDetails");
            var detailDesc = view.ui.get("DetailDescription");
            var detailUnitPrice = view.ui.get("DetailUnitPrice");
            var detailImg = view.ui.get("DetailImage");
            var itemPixSvc = view.ui.get("ItemPixURLSvc");
            //Populate Details dialog
            detailDesc.setText(row.data.description);
            detailUnitPrice.setValue(row.data.unitPrice);
            detailImg.setCaption(itemName);
            detailsDlg.setTitle("Details for " + itemName);
            detailImg.setImage(null); //Clear current picture
            detailsDlg.show();//Display details dialog
            //Call Picture URL svc w/ Item Name
            itemPixSvc.execute(itemName);
        };

        var res = {
            node: div,
            value: function() {return row.data.name;}
        };

        return res;
    }
}
```

Create and "decorate" the custom HTML content of the cell. This logic creates a link within a div
Create on-click logic on the link to display a dialog containing item details
Return the HTML DOM element and a "value" function used to search and sort this cell

Figure 7-42 Custom cell rendering function detail for Fast Table example

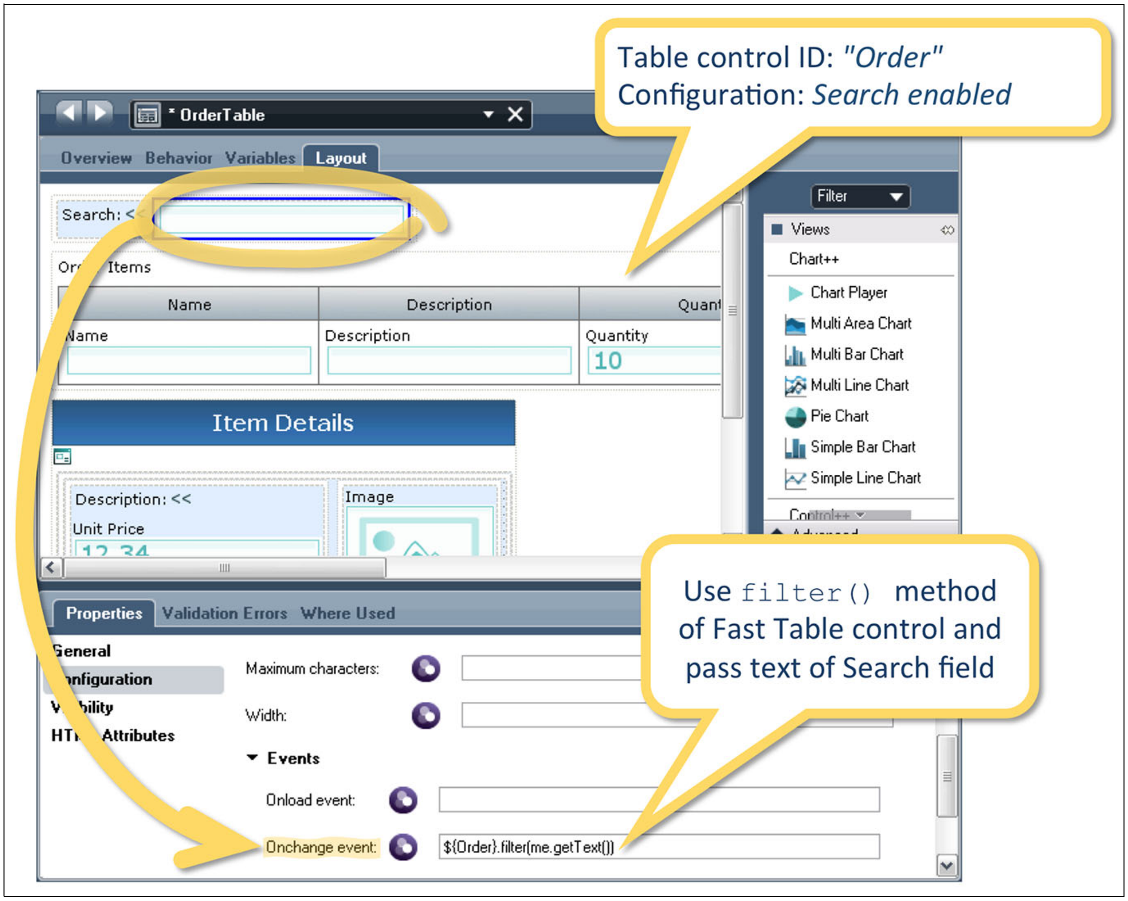
Search feature

The cross-column search feature of Fast Table allows a user to specify a search string (optionally containing wildcards), which is applied to the table to filter *in* all matching rows. The search feature relies on the `filter()` method of the Fast Table control.

The search feature must be enabled in the Fast Table configuration, otherwise the filter method is inoperative.

At loading time (assuming searching is enabled), the content of each record is made searchable. Columns that contain custom-rendered content are searchable (and sortable) based on the `value` function in the object returned to the *On new cell* event.

Figure 7-43 shows how to apply a table filter that is based on the content of a Text control.



Onchange event: `${Order}.filter(me.getText())`

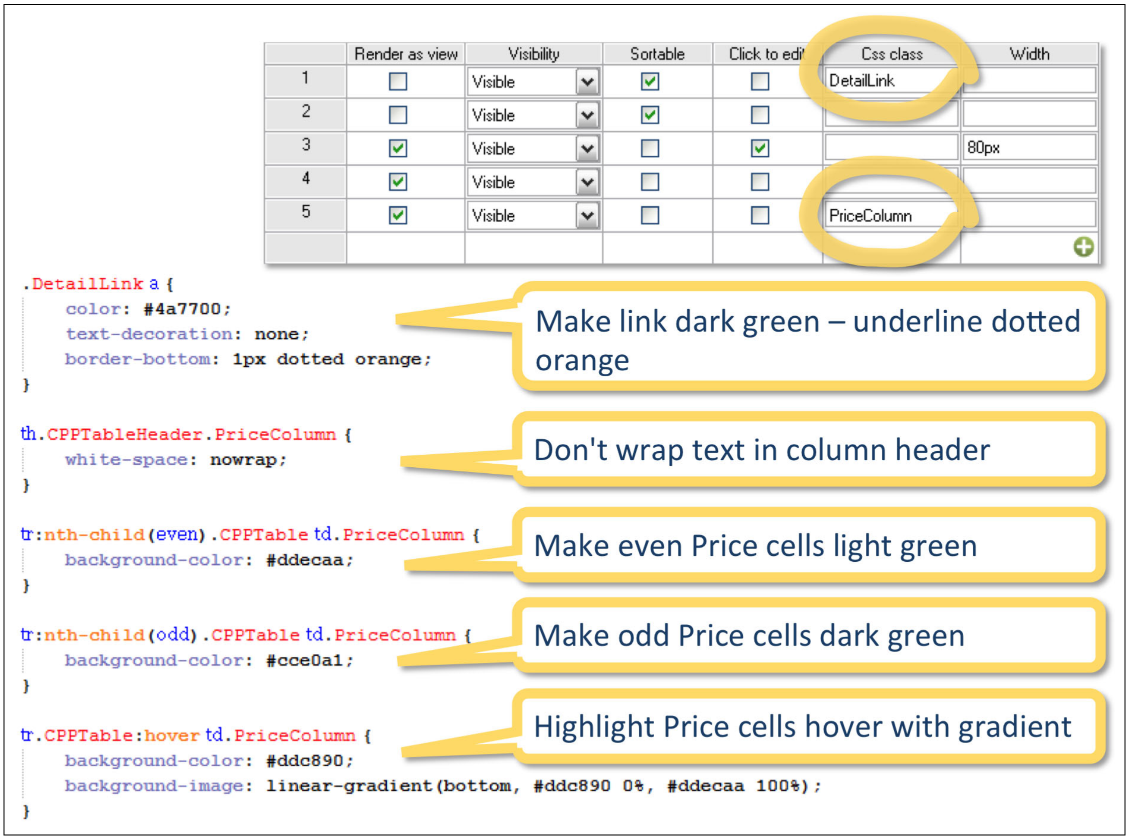
Figure 7-43 Search feature for Fast Table example

To trigger the search, the filter function is triggered from the Onchange event of the Text field that is used to specify the search string. The filter method removes any filtering if an empty string is specified, so clearing the Text field content effectively clears the filter.

Styling of columns

The Fast Table control renders rows, columns, and cells with specific CSS classes to style the table. Additional CSS classes can be specified on a per-column basis.

Figure 7-44 puts in context the per-column configuration for the Fast Table control specifying the DetailLink CSS class for the first column (Name) and the PriceColumn CSS class for the fifth column (Price).



	Render as view	Visibility	Sortable	Click to edi	Css class	Width
1	☐	Visible	☑	☐	DetailLink	
2	☐	Visible	☑	☐		
3	☑	Visible	☐	☑		80px
4	☑	Visible	☐	☐		
5	☑	Visible	☐	☐	PriceColumn	

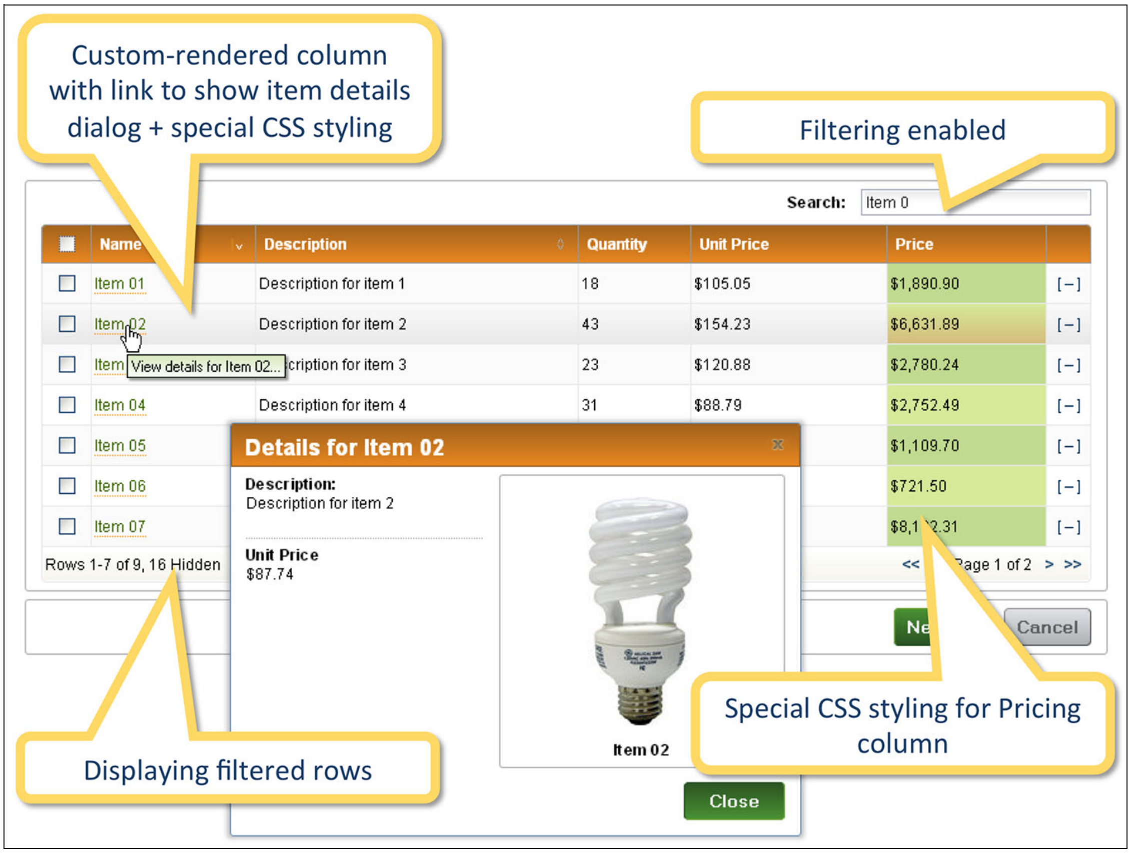
```
.DetailLink a {
    color: #4a7700;
    text-decoration: none;
    border-bottom: 1px dotted orange;
}

th.CPPTableHeader.PriceColumn {
    white-space: nowrap;
}

tr:nth-child(even).CPPTable td.PriceColumn {
    background-color: #ddecaa;
}

tr:nth-child(odd).CPPTable td.PriceColumn {
    background-color: #cce0a1;
}

tr.CPPTable:hover td.PriceColumn {
    background-color: #ddc890;
    background-image: linear-gradient(bottom, #ddc890 0%, #ddecaa 100%);
}
```

Make link dark green – underline dotted orange
Don't wrap text in column header
Make even Price cells light green
Make odd Price cells dark green
Highlight Price cells hover with gradient

Figure 7-44 CSS styling for Fast Table example

Finished example

Figure 7-45 is the last illustration for this Fast Table example shows the finished product, including the custom-rendered, styled, and filtered aspects of the table control.


Custom-rendered column with link to show item details dialog + special CSS styling
Filtering enabled
Displaying filtered rows
Special CSS styling for Pricing column
Ne
Cancel
Close

Search: Item 0

	Name	Description	Quantity	Unit Price	Price	
☐	Item 01	Description for item 1	18	$105.05	$1,890.90	[–]
☐	Item 02	Description for item 2	43	$154.23	$6,631.89	[–]
☐	Item	cription for item 3	23	$120.88	$2,780.24	[–]
☐	Item 04	Description for item 4	31	$88.79	$2,752.49	[–]
☐	Item 05				$1,109.70	[–]
☐	Item 06				$721.50	[–]
☐	Item 07				$8,1[illegible]2.31	[–]

Rows 1-7 of 9, 16 Hidden

<< Page 1 of 2 > >>

View details for Item 02...

Details for Item 02

Description:
Description for item 2

Unit Price
$87.74

Item 02

Figure 7-45 Finished Fast Table example

Also included but not explicitly called out in this example is paging, sorting support for the first two columns, multi-selection support (which appropriately tracks selection for the list of bound data), and the use of the on row delete event to confirm row deletion (see Figure 7-39 on page 280).

In summary

Fast Table and Fast Data Table are versatile high-performance tabular data display controls. They can be used to display data from various sources, such as:

- Business data from the process/task
- Business data from a SQL query or logic executed in an Ajax service

- Data obtained from a Web Service call
- A tabular report to supplement a chart
- A custom task list populated through TWSearch

The controls provide intuitive configuration options combined with event-handling and customization capabilities that support the implementation of sophisticated requirements in an intuitive manner.

Good practices

Take a look at some good practices:

- For a large expected volume of rows in a table, UI authors should take advantage of the *Render as view, Use asynchronous loading*, and *Async load batch size* configuration options, balanced with imperatives from business requirements, to optimize the end-user experience.
- Fast Table (and its Ajax service-backed Fast Data Table counterpart) can be used to display data from data bindings, from Ajax services, and even fully programmatically to add and delete rows through the deleteRecord or appendRecord methods.
- Use Fast Table instead of the standard Table control, both to mitigate potential performance concerns for large volumes of rows or to take advantage of the additional rendering customization capabilities of the control.

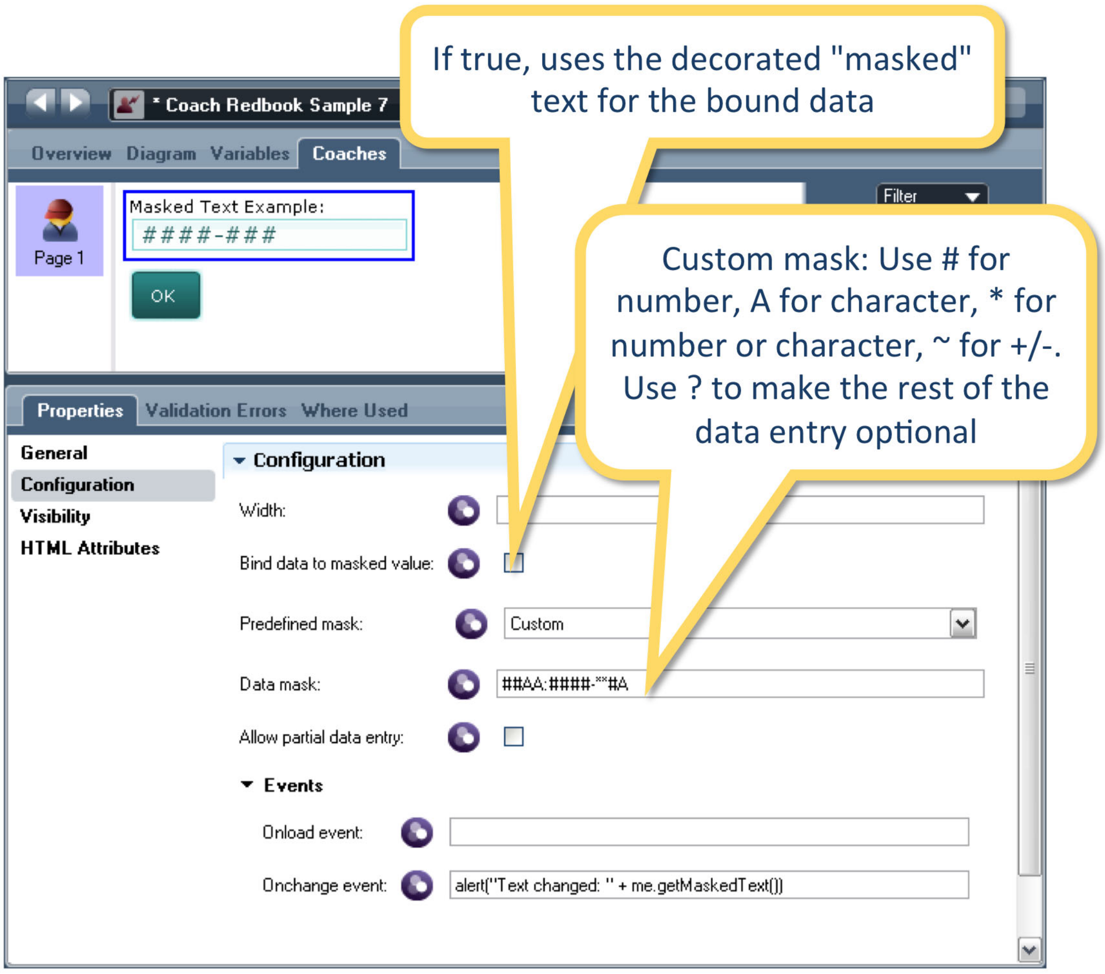
Masked Text

The Masked Text control is an example of preemptive client-side validation, which both guides the end-user in the format of the data entry and prevents the entry of invalid characters (instead of flagging an error on the control after the fact). A Masked Text control can still use validation (`setValid/isValid`. See 7.2.5, “Handling validation” on page 261) to flag errors that go beyond the simple mask specification.

Configuration options

The Masked Text control provides a number of configuration options to control visual aspects, functional and performance capabilities, and to handle key table control events.

Figure 7-46 on page 288 shows the Masked Text configuration options as seen in Process Designer.



If true, uses the decorated "masked" text for the bound data

Custom mask: Use # for number, A for character, * for number or character, ~ for +/-. Use ? to make the rest of the data entry optional

Masked Text Example: ####-###

Width:
Bind data to masked value: ☐
Predefined mask: Custom
Data mask: ##AA:####-**#A
Allow partial data entry: ☐

Onload event:
Onchange event: alert("Text changed: " + me.getMaskedText())

Figure 7-46 Configuration options: Masked Text control

The Masked Text control behaves at runtime as shown in Figure 7-47 on page 289.

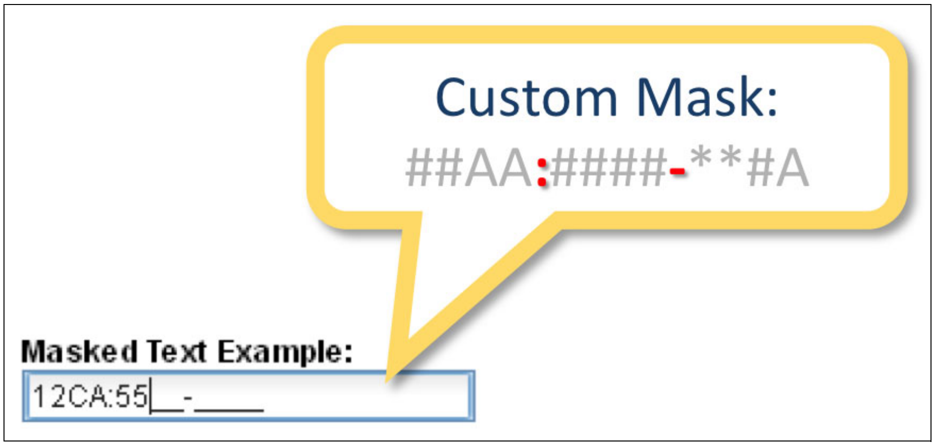


Figure 7-47 Runtime example: Masked Text control

A Masked Text control, if bound to Coach data, can control whether its data reflects the masked text (meaning with the decorations from the mask, for example in the case above: 12CA:5501-A11E) or without the decorations (again using the above example: 12CA5501A11E). Masked Text provides special Text getters/setters to distinguish between masked and unmasked data, namely: `get/setMaskedText()` and `get/setUnmaskedText()`.

Charts

Charts provide end-users the ability to view aggregate data returned by Ajax services. EmeriConVIEWS provides the following chart types:

- Pie Chart
- Single-data Series Bar Chart
- Single-data Series Line Chart
- Multi-data Series Bar Chart
- Multi-data Series Line Chart
- Multi-data Series Area Chart

Figure 7-48 on page 290 shows all the EmeriConVIEWS chart types (enumerated above) rendered at run time.

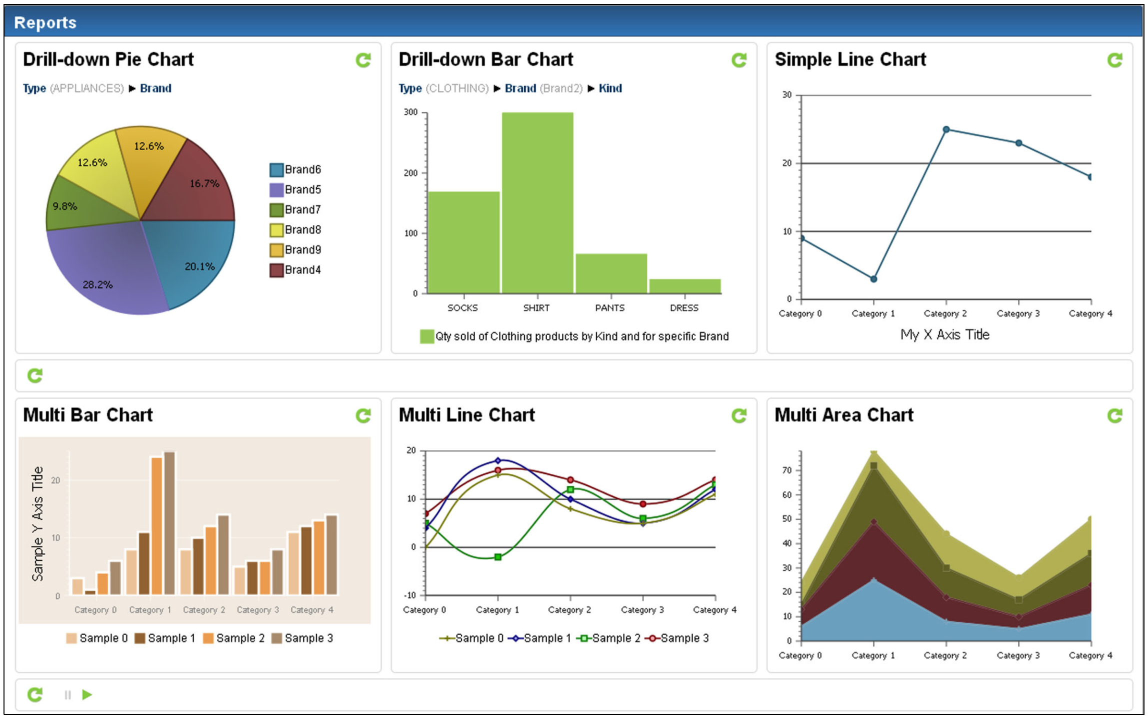
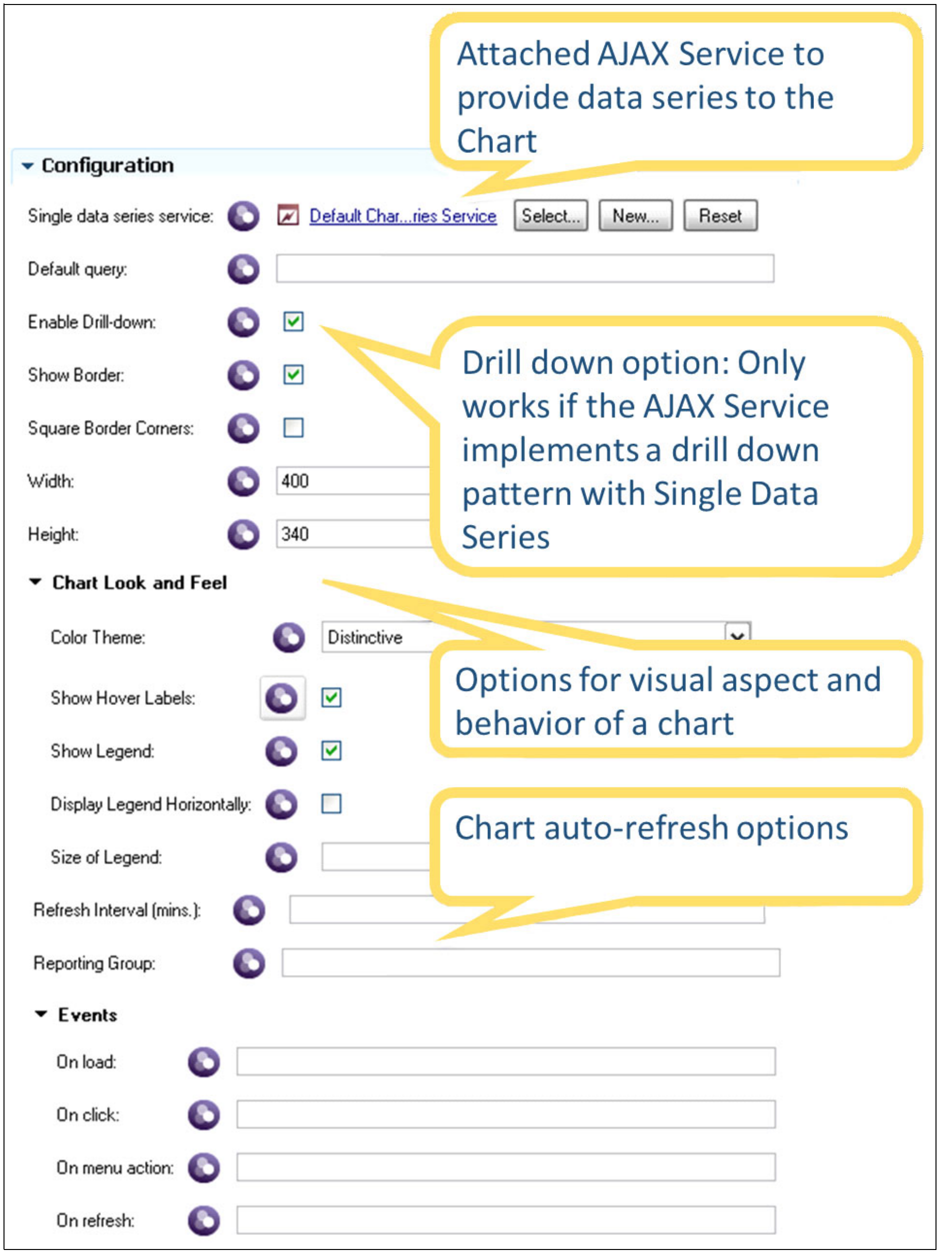


Figure 7-48   Runtime example: Chart controls

Configuration options

Charts use their data binding as the input for the Data Series Ajax Service. In addition to the service query input, a number of additional configuration parameters can be specified for each chart. Configuration parameters vary slightly from one chart type to another, but many options are common among charts.

Figure 7-49 shows configuration options for the Pie Chart.


Attached AJAX Service to provide data series to the Chart

Configuration

Single data series service: Default Char...ries Service Select... New... Reset

Default query:

Enable Drill-down: ✓

Show Border: ✓

Square Border Corners: ☐

Width: 400

Height: 340

Drill down option: Only works if the AJAX Service implements a drill down pattern with Single Data Series

Chart Look and Feel

Color Theme: Distinctive

Show Hover Labels: ✓

Show Legend: ✓

Display Legend Horizontally: ☐

Size of Legend:

Options for visual aspect and behavior of a chart

Refresh Interval (mins.):

Reporting Group:

Chart auto-refresh options

Events

On load:

On click:

On menu action:

On refresh:

Figure 7-49 Pie Chart sample configuration options

Refreshing chart data

Because EmeriConVIEWS charts obtain data from Ajax services, each chart on a Coach Page or View can be refreshed individually without requiring a reload. The refresh of the chart can be triggered:

- Manually by an explicit request by the end-user using the chart refresh button.
- Programmatically by calling the `refresh()` method on the chart control, new data can be passed into the query by calling the `setQueryData()` method first.
- Automatically through the *Refresh Interval* configuration option (note that auto-refreshes can be suspended if the end-user clicks the pause button).
- Automatically through the refresh triggered by the reporting group specified in the *Reporting Group* configuration option (see explanation below).

If two charts belong to a reporting group *G1* and a Chart Player control (see Figure 7-50 on page 293) is configured for Reporting Group *G1*, both charts will refresh based on triggers from the Chart Player control. A Chart Player is a convenience control so the end-user does not have to click several chart refresh buttons or so that the UI author does not have to configure the same refresh intervals for many charts.

Figure 7-50 provides an example of two charts and a Chart Player control (under the charts in this example).

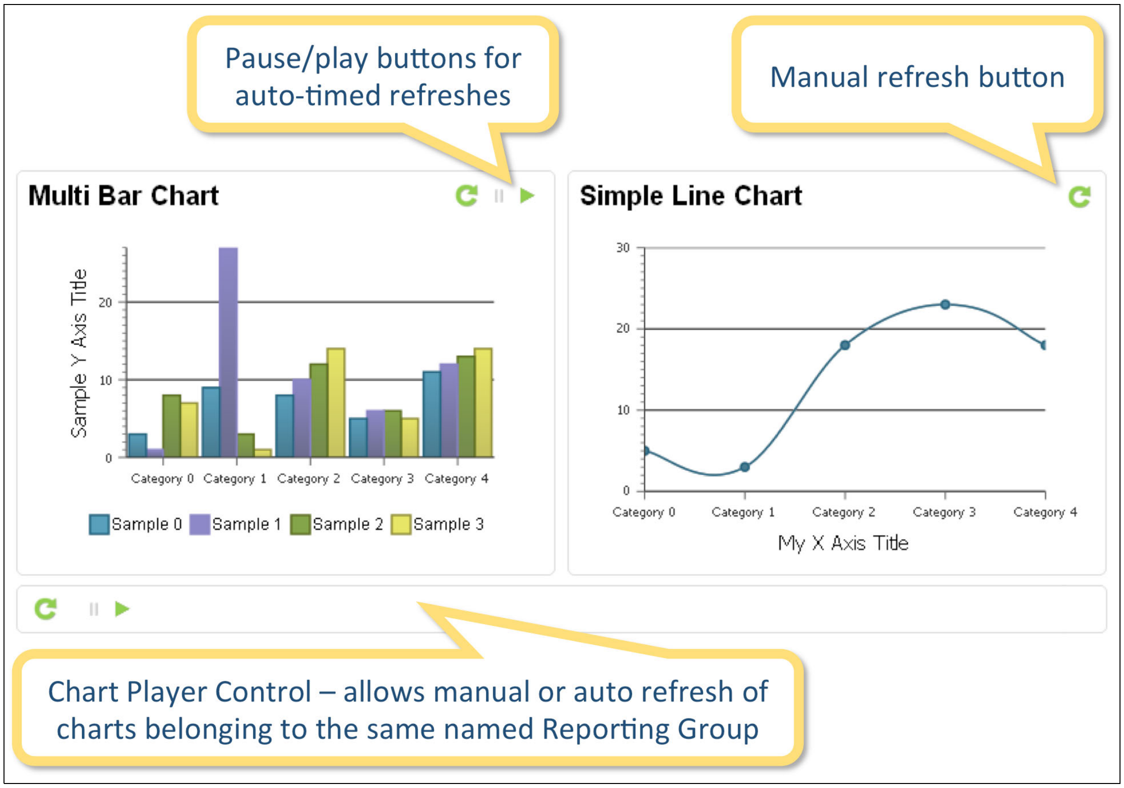


Figure 7-50 Chart refresh options at run time

Chart Player control

The use of a Chart Player control is not mandatory to refresh charts. It is merely a convenience when many charts are on a Coach Page or View to refresh an entire group of charts at the same time.

The Chart Player control is manifested visually as shown in Figure 7-50 on page 293. Its configuration options are shown in Figure 7-51.

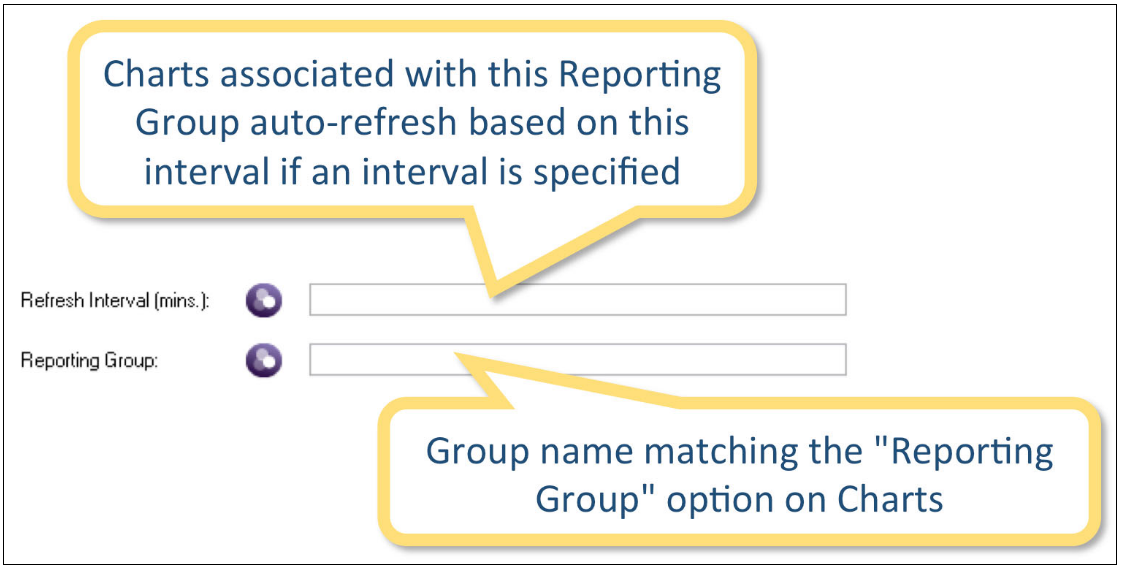
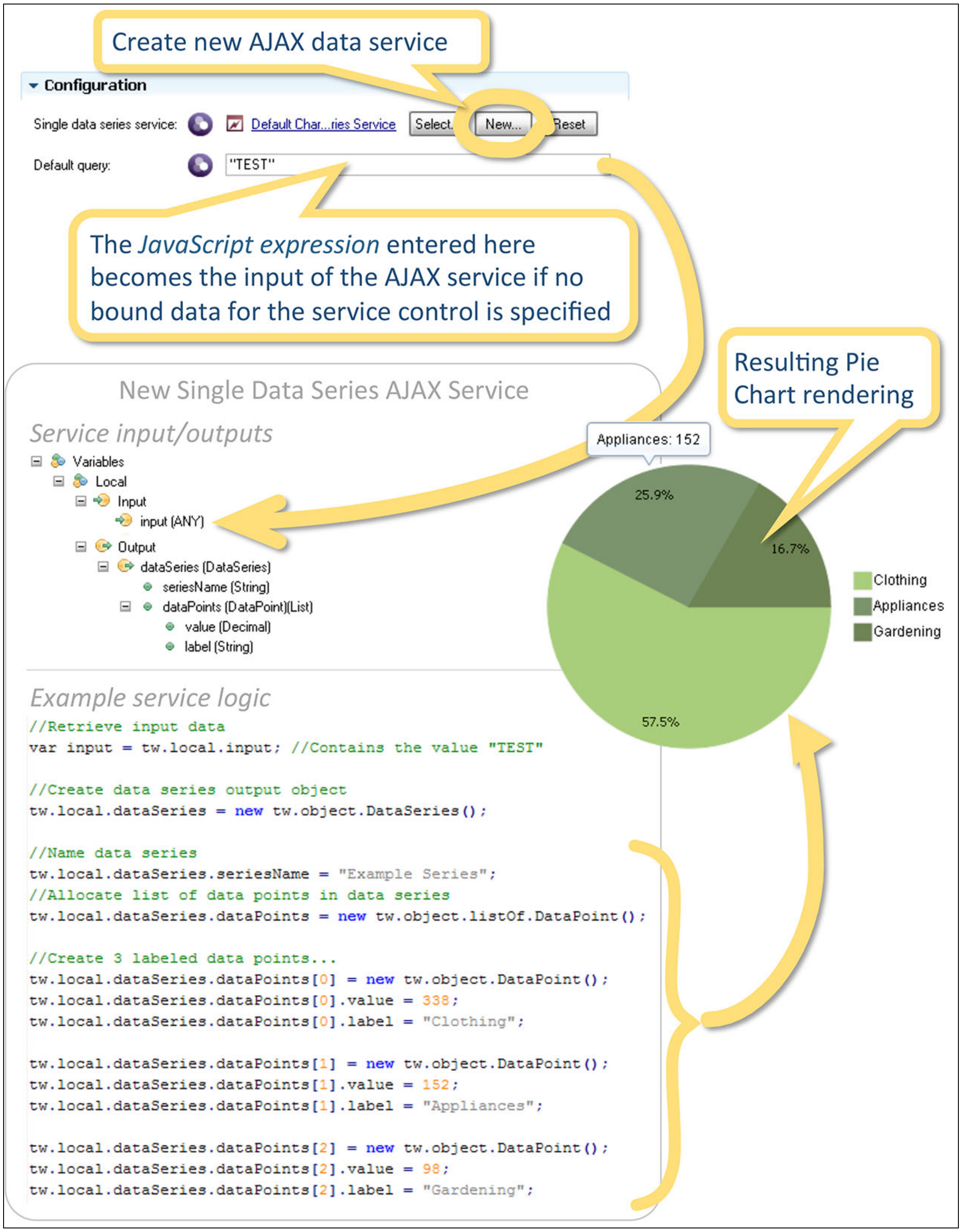

Charts associated with this Reporting Group auto-refresh based on this interval if an interval is specified
Refresh Interval (mins.):
Reporting Group:
Group name matching the "Reporting Group" option on Charts

Figure 7-51 Chart Player configuration options

Ajax Services for charts

To create an Ajax service for a chart, open the chart's configuration options, locate the attached data service option at the top, create a new service, and add its logic. Figure 7-52 on page 295 shows a single data series service that is created with sample data for a Pie Chart.


Create new AJAX data service
Configuration
Single data series service:
Default Char...ries Service
Select.
New...
Reset
Default query:
"TEST"
The *JavaScript expression* entered here becomes the input of the AJAX service if no bound data for the service control is specified
Resulting Pie Chart rendering
New Single Data Series AJAX Service
Service input/outputs
Variables
Local
Input
input (ANY)
Output
dataSeries (DataSeries)
seriesName (String)
dataPoints (DataPoint)(List)
value (Decimal)
label (String)
Appliances: 152
25.9%
16.7%
57.5%
Clothing
Appliances
Gardening
Example service logic

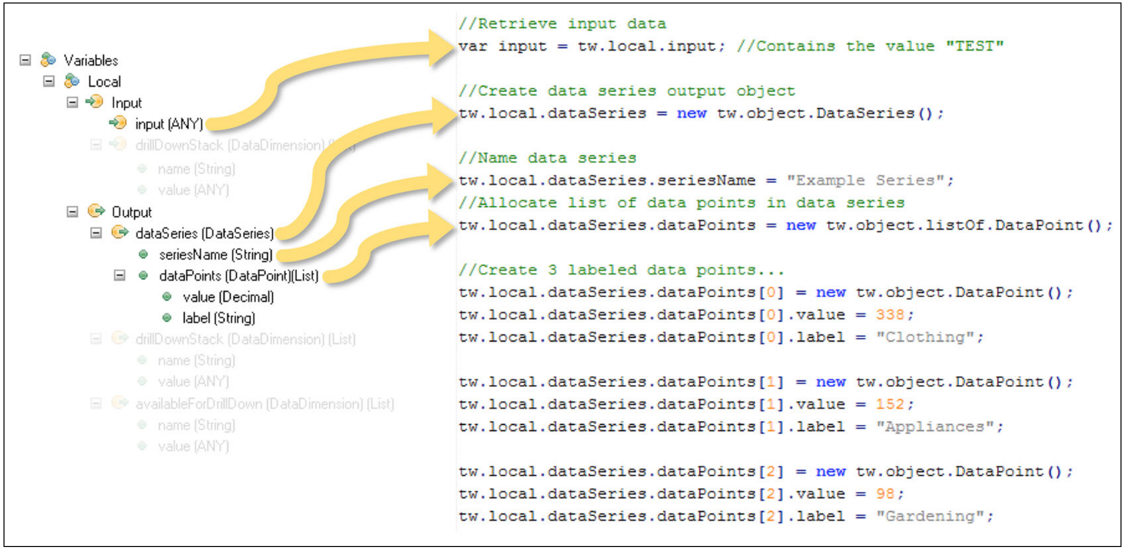
```
//Retrieve input data
var input = tw.local.input; //Contains the value "TEST"

//Create data series output object
tw.local.dataSeries = new tw.object.DataSeries();

//Name data series
tw.local.dataSeries.seriesName = "Example Series";
//Allocate list of data points in data series
tw.local.dataSeries.dataPoints = new tw.object.listOf.DataPoint();

//Create 3 labeled data points...
tw.local.dataSeries.dataPoints[0] = new tw.object.DataPoint();
tw.local.dataSeries.dataPoints[0].value = 338;
tw.local.dataSeries.dataPoints[0].label = "Clothing";

tw.local.dataSeries.dataPoints[1] = new tw.object.DataPoint();
tw.local.dataSeries.dataPoints[1].value = 152;
tw.local.dataSeries.dataPoints[1].label = "Appliances";

tw.local.dataSeries.dataPoints[2] = new tw.object.DataPoint();
tw.local.dataSeries.dataPoints[2].value = 98;
tw.local.dataSeries.dataPoints[2].label = "Gardening";
```

Figure 7-52 Creating an Ajax Data Service for a chart

Data series Ajax services can contain JavaScript logic, SQL queries, Web Service calls, and any other construct available in the BPMN service engine to create and populate the data series to be returned to the chart.

Single-data series Ajax service example

Figure 7-53 provides a basic example of a single-data series Ajax service (the same as shown in Figure 7-52 on page 295), with explicit mappings between service input and output variables and the sample JavaScript logic. The disabled variables in the image are for drill-down-enabled service logic and will be discussed shortly.



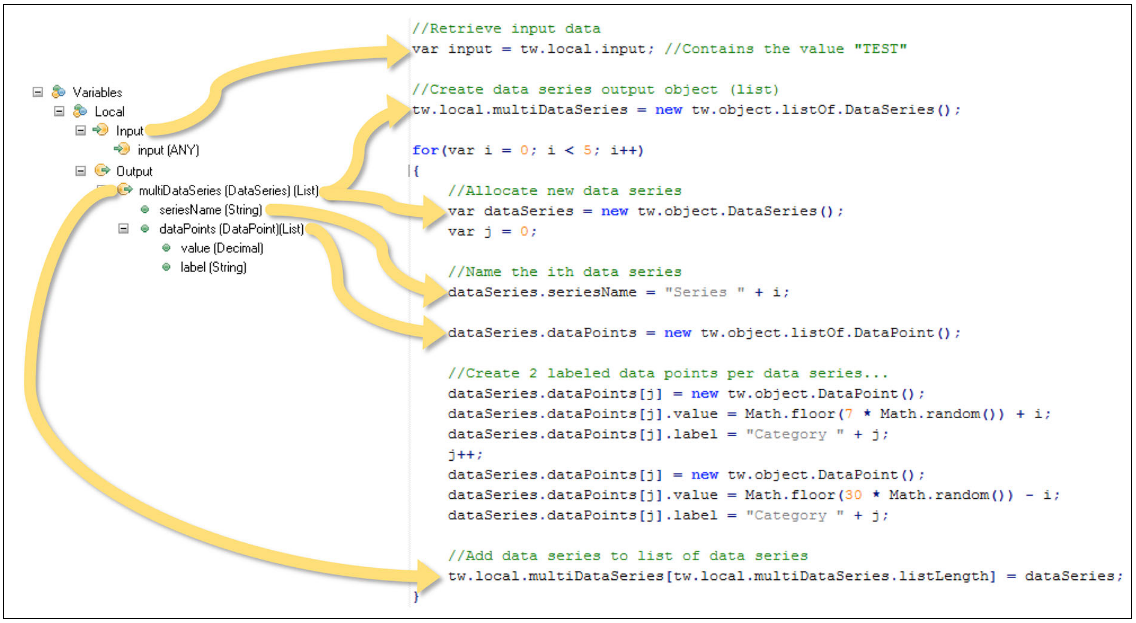
```
//Retrieve input data
var input = tw.local.input; //Contains the value "TEST"

//Create data series output object
tw.local.dataSeries = new tw.object.DataSeries();

//Name data series
tw.local.dataSeries.seriesName = "Example Series";
//Allocate list of data points in data series
tw.local.dataSeries.dataPoints = new tw.object.listOf.DataPoint();

//Create 3 labeled data points...
tw.local.dataSeries.dataPoints[0] = new tw.object.DataPoint();
tw.local.dataSeries.dataPoints[0].value = 338;
tw.local.dataSeries.dataPoints[0].label = "Clothing";

tw.local.dataSeries.dataPoints[1] = new tw.object.DataPoint();
tw.local.dataSeries.dataPoints[1].value = 152;
tw.local.dataSeries.dataPoints[1].label = "Appliances";

tw.local.dataSeries.dataPoints[2] = new tw.object.DataPoint();
tw.local.dataSeries.dataPoints[2].value = 98;
tw.local.dataSeries.dataPoints[2].label = "Gardening";
```

Figure 7-53 Single-data series Ajax service example

Multi-data series Ajax service example

Multi-data series charts require a slightly different type of Ajax service to return not one but many data series.

Figure 7-54 shows the slightly different signature and different implementation of a multi-data series Ajax service.



```
//Retrieve input data
var input = tw.local.input; //Contains the value "TEST"

//Create data series output object (list)
tw.local.multiDataSeries = new tw.object.listOf.DataSeries();

for(var i = 0; i < 5; i++)
{
    //Allocate new data series
    var dataSeries = new tw.object.DataSeries();
    var j = 0;

    //Name the ith data series
    dataSeries.seriesName = "Series " + i;

    dataSeries.dataPoints = new tw.object.listOf.DataPoint();

    //Create 2 labeled data points per data series...
    dataSeries.dataPoints[j] = new tw.object.DataPoint();
    dataSeries.dataPoints[j].value = Math.floor(7 * Math.random()) + i;
    dataSeries.dataPoints[j].label = "Category " + j;
    j++;
    dataSeries.dataPoints[j] = new tw.object.DataPoint();
    dataSeries.dataPoints[j].value = Math.floor(30 * Math.random()) - i;
    dataSeries.dataPoints[j].label = "Category " + j;

    //Add data series to list of data series
    tw.local.multiDataSeries[tw.local.multiDataSeries.listLength] = dataSeries;
}
```

Figure 7-54 Multi-data series Ajax service example

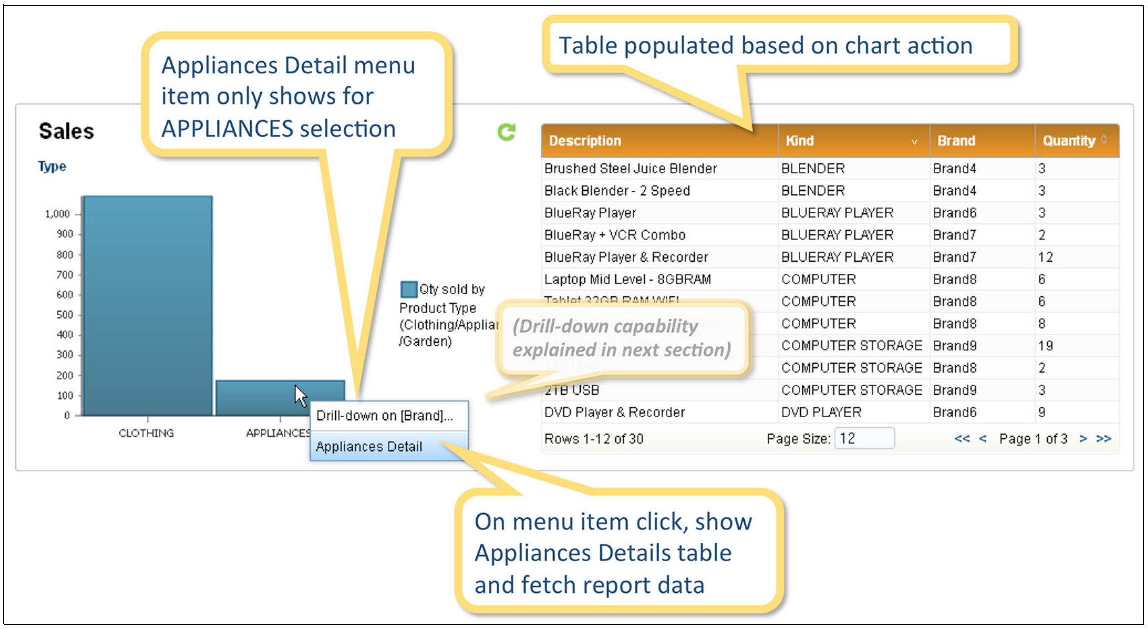
Selecting New on the multi-data series service in the configuration options of a multi-data series chart automatically creates an Ajax service that is based on the proper multi-data series output signature.

Event support in charts

Sophisticated charting and reporting behavior can be achieved by combining the use of the *on click*, *on menu action*, and *on result* events for charts with the `setQueryData()`, `refresh()`, and `addMenuAction()` methods of charts.

This allows one chart to react to an action or a drill-down on another chart. Or it can, for example, allow a Fast Data Table to be populated with details about a selected chart element. The end result can provide a highly interactive and meaningful reporting experience for end-users.

In Figure 7-55, a Fast Data Table fetches business data when a custom menu item is clicked in a Bar Chart.

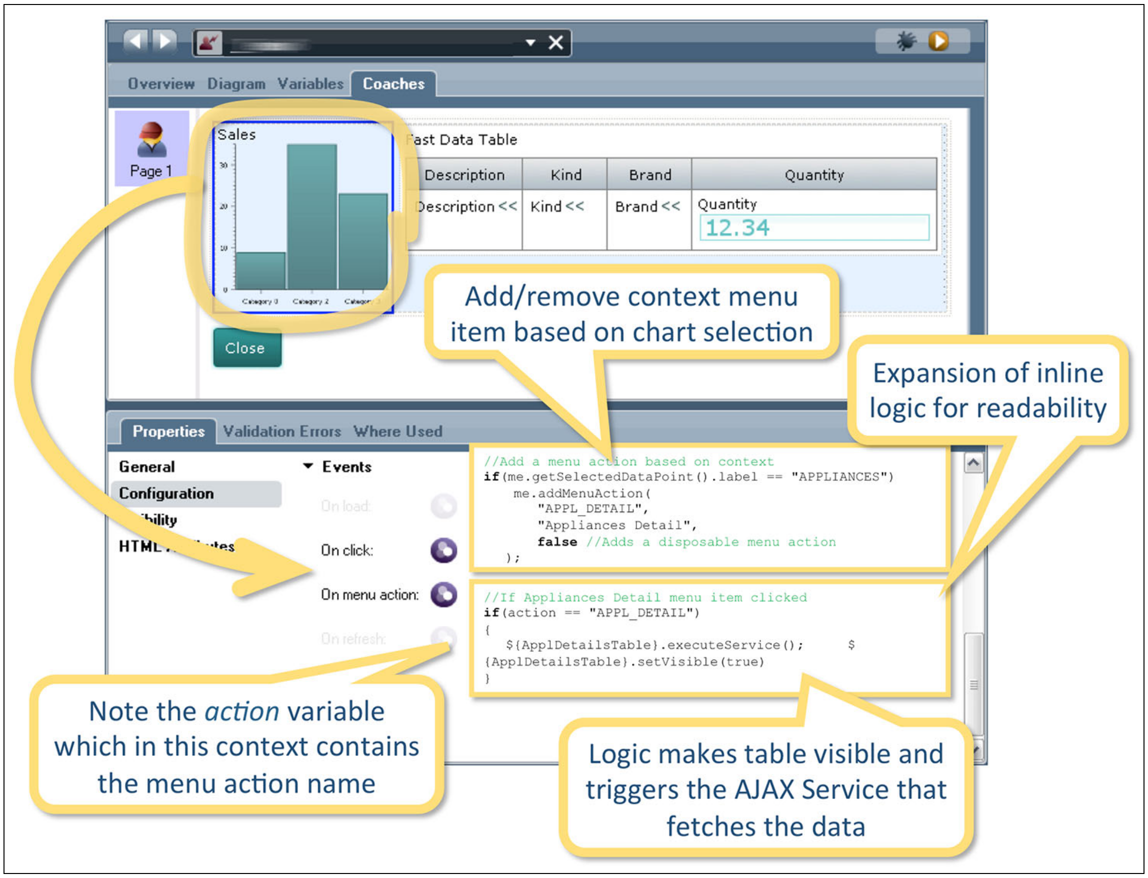

Appliances Detail menu item only shows for APPLIANCES selection
Table populated based on chart action
Sales
Type
1,000
900
800
700
600
500
400
300
200
100
0
CLOTHING
APPLIANCES
🟦 Qty sold by Product Type (Clothing/Applia /Garden)
Drill-down on [Brand]...
Appliances Detail
(Drill-down capability explained in next section)
On menu item click, show Appliances Details table and fetch report data

Description	Kind	Brand	Quantity
Brushed Steel Juice Blender	BLENDER	Brand4	3
Black Blender - 2 Speed	BLENDER	Brand4	3
BlueRay Player	BLUERAY PLAYER	Brand6	3
BlueRay + VCR Combo	BLUERAY PLAYER	Brand7	2
BlueRay Player & Recorder	BLUERAY PLAYER	Brand7	12
Laptop Mid Level - 8GBRAM	COMPUTER	Brand8	6
Tablet 32GB RAM WIFI	COMPUTER	Brand8	6
[illegible]	COMPUTER	Brand8	8
[illegible]	COMPUTER STORAGE	Brand9	19
[illegible]	COMPUTER STORAGE	Brand8	2
2TB USB	COMPUTER STORAGE	Brand9	3
DVD Player & Recorder	DVD PLAYER	Brand6	9

Rows 1-12 of 30 Page Size: 12 << < Page 1 of 3 > >>

Figure 7-55 Example of interactive chart behavior to display data details

The Process Designer Editor illustration that is seen in Figure 7-56 shows how the chart to table reporting behavior is created.



```
//Add a menu action based on context
if(me.getSelectedDataPoint().label == "APPLIANCES")
    me.addMenuAction(
       "APPL_DETAIL",
       "Appliances Detail",
       false //Adds a disposable menu action
   );
```

```
//If Appliances Detail menu item clicked
if(action == "APPL_DETAIL")
{
   ${ApplDetailsTable}.executeService();      $
{ApplDetailsTable}.setVisible(true)
}
```

Figure 7-56 Interactive chart behavior authoring

Chart menu actions can be durable or disposable. Disposable menu actions are usually very context-specific, such as the menu action illustrated in Figure 7-56, which only makes sense if the end-user clicks the chart bar labeled APPLIANCES. Such actions self-cleanup after they are invoked or after the menu disappears. These actions are usually created dynamically as part of a Chart *on click* event. To create a disposable action, the `addMenuAction()` method is called with `false` (meaning not durable) as the third argument.

A durable action is created by calling `addMenuAction()` with true (meaning durable) as the third argument.

With the ability to add chart-specific or context-specific menu items to charts (pie, bar, or line charts) and to attach behaviors to each menu action through configuration-based event logic, Coach UI authors can create sophisticated interdependent behaviors to not only display initial reporting data but also to create meaningful analysis capabilities for report end-users.

Data analysis through drill-down support

Single data series charts also support data drill-down. To enable drill-down, the following conditions are necessary:

- The chart configuration's *Enable drilldown* option must be selected
- The Ajax service that is attached to the chart must support the drill-down pattern (described below)

A drill down-enabled chart can support the following behavior as shown in Figure 7-57.

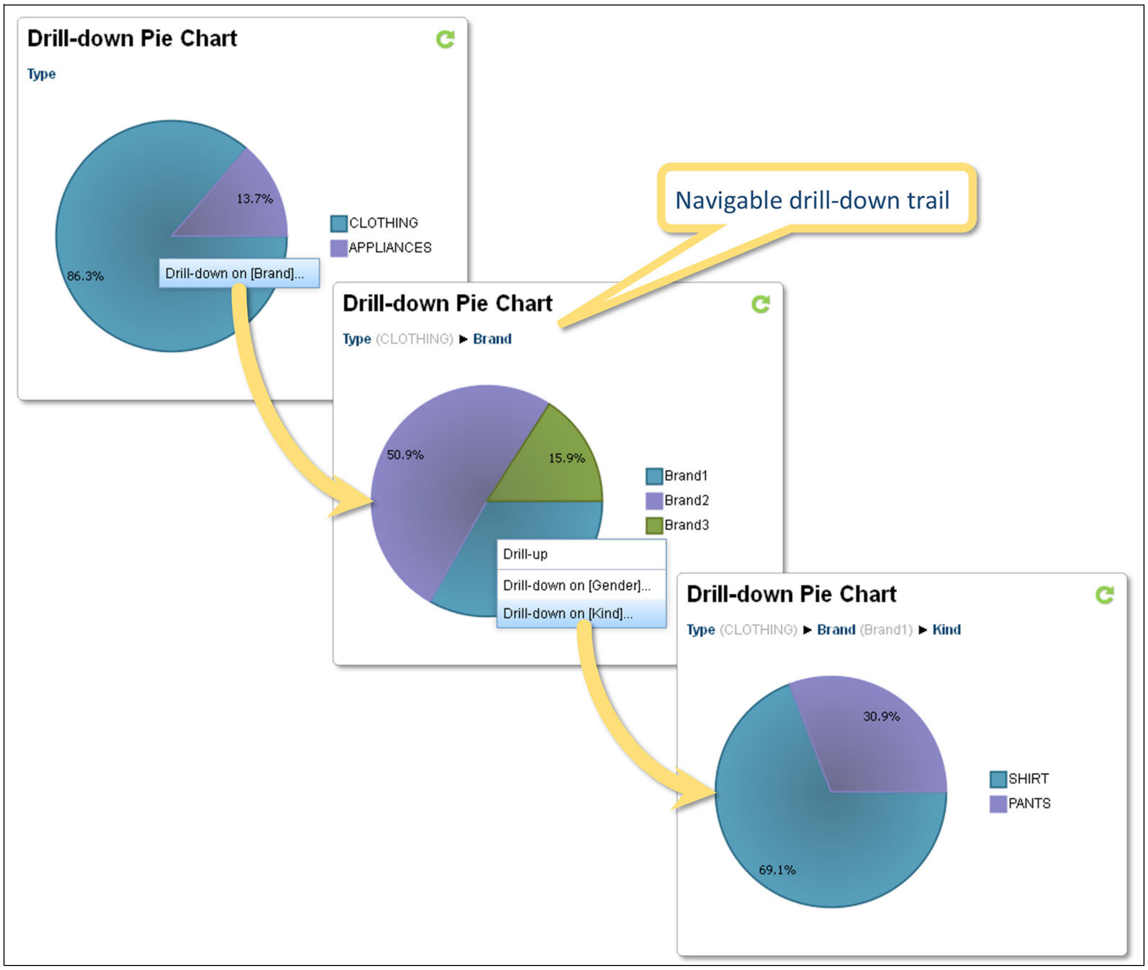


Figure 7-57 Runtime drill-down enabled chart behavior example

The drill-down service pattern for EmeriConVIEWS charts allows authors to define a drill-down tree (see Figure 7-58 on page 302). The drill-down tree defines the allowed drill-down dimensions and paths that are taken to the data drill-down analysis.

Figure 7-58 shows a tree describing all allowed drill-down paths for a chart (actually for all charts using this particular service implementing this tree).

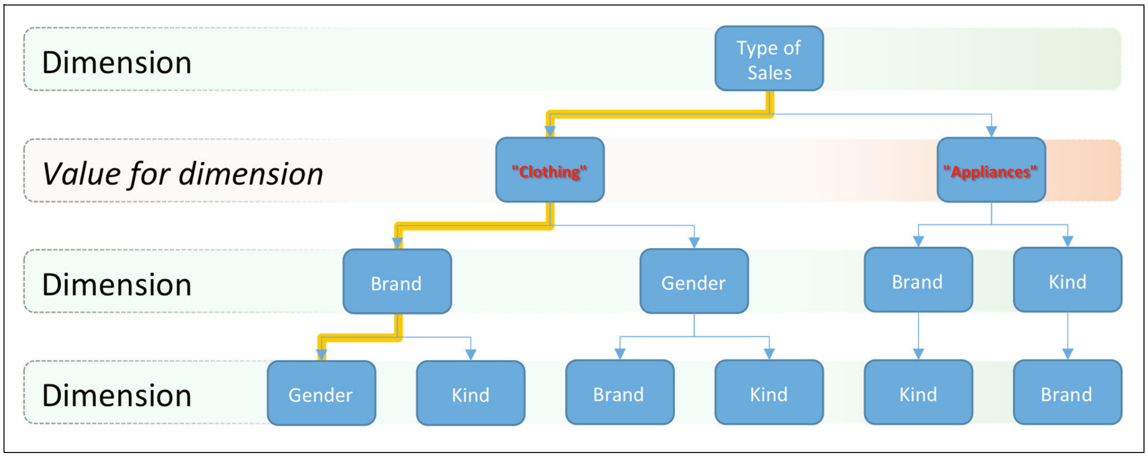


Figure 7-58 Drill-down tree example

Note how the navigation of a drill-down tree isn't solely based on dimension, but also potentially based on the *value* of a dimension. For example, the drill-down tree in Figure 7-58 allows Brand and Gender as drill down choices for Clothing sales, whereas Brand and Kind are the available choices for Appliance sales.

Figure 7-59 on page 303 shows the behavior of a Pie Chart (including the available drill-down) choices through one of the drill-down paths (following the orange highlight).

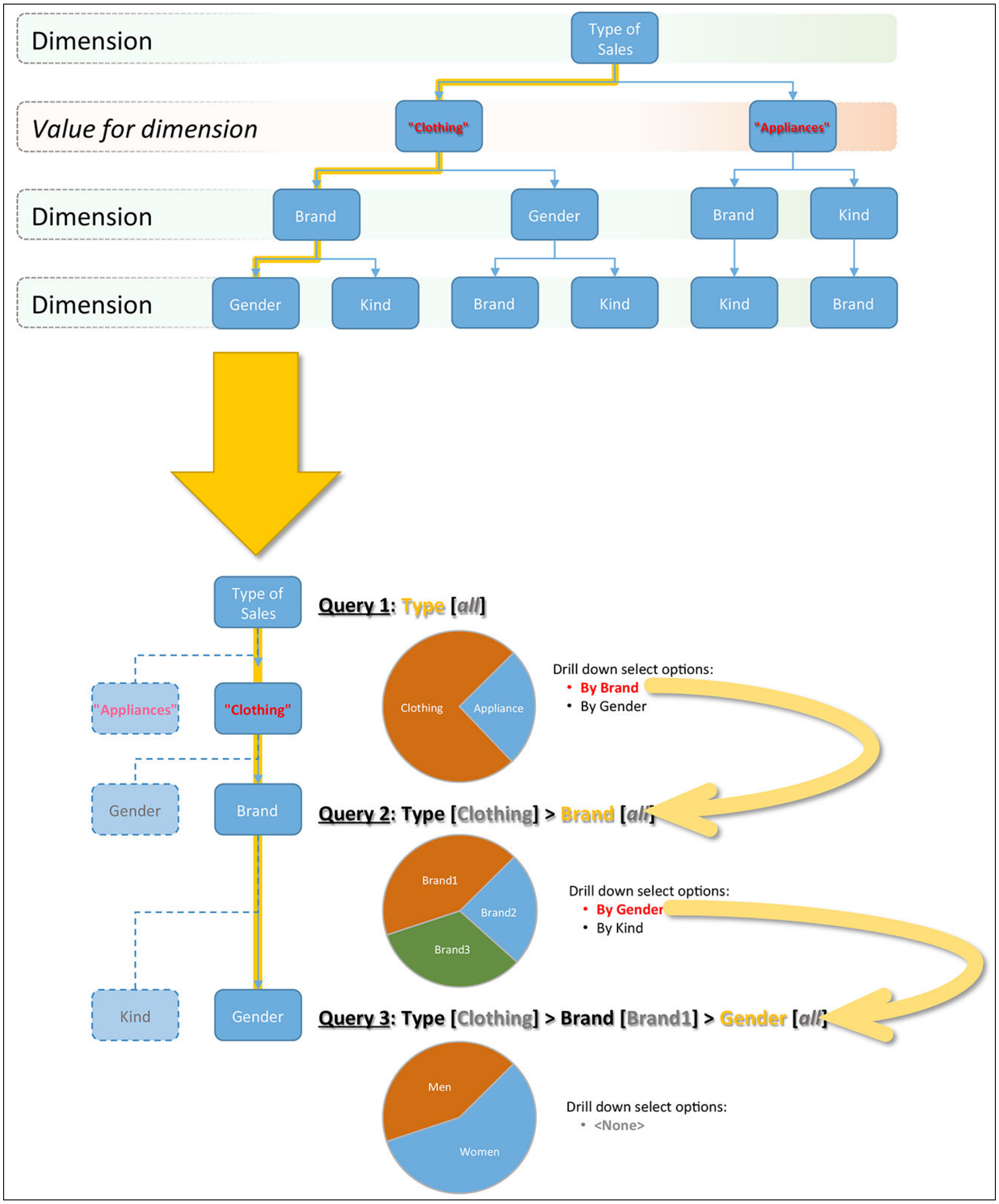


Figure 7-59 Pie chart following a path in a drill-down tree

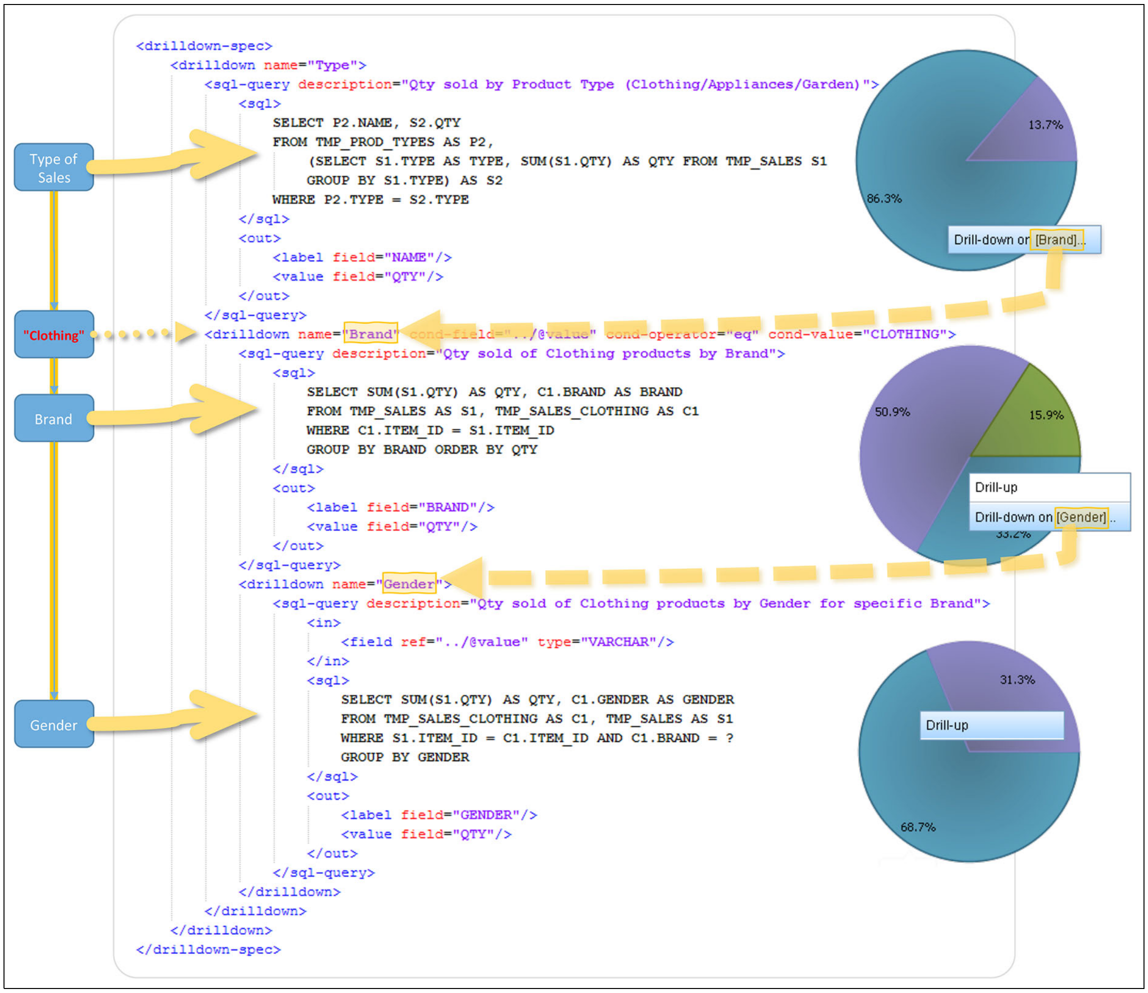
The effort that is required to implement the drill-down logic and behavior, statelessly keep track of each path taken, correlate parent and child queries, support navigation of the tree, and to synchronize the chart accordingly could be non-trivial. The EmeriConVIEWS toolkit contains a special (*Execute Drill-down Query*) General System Service that can interpret and navigate a *drill-down tree*. This enables the author to focus exclusively on the business and data analysis requirements.

Once a drill-down tree is defined at a high level, the work that is required is limited to defining SQL queries or JavaScript querying logic behind each node in the tree to implement the Ajax service.

The next section shows how to create the tree needed to implement the drill-down example above. For clarity and brevity, only one path through the drill-down tree is provided as an example.

Drill-down service example

Figure 7-60 on page 305 is an introduction to the drill-down tree structure, in context with the drill-down path (on the left) and the charts (on the right). In the case shown in Figure 7-60 on page 305, each node in the tree is backed by a SQL query. A query can also be implemented by JavaScript code using a `<js-query>` instead of a `<sql-query>` tag.



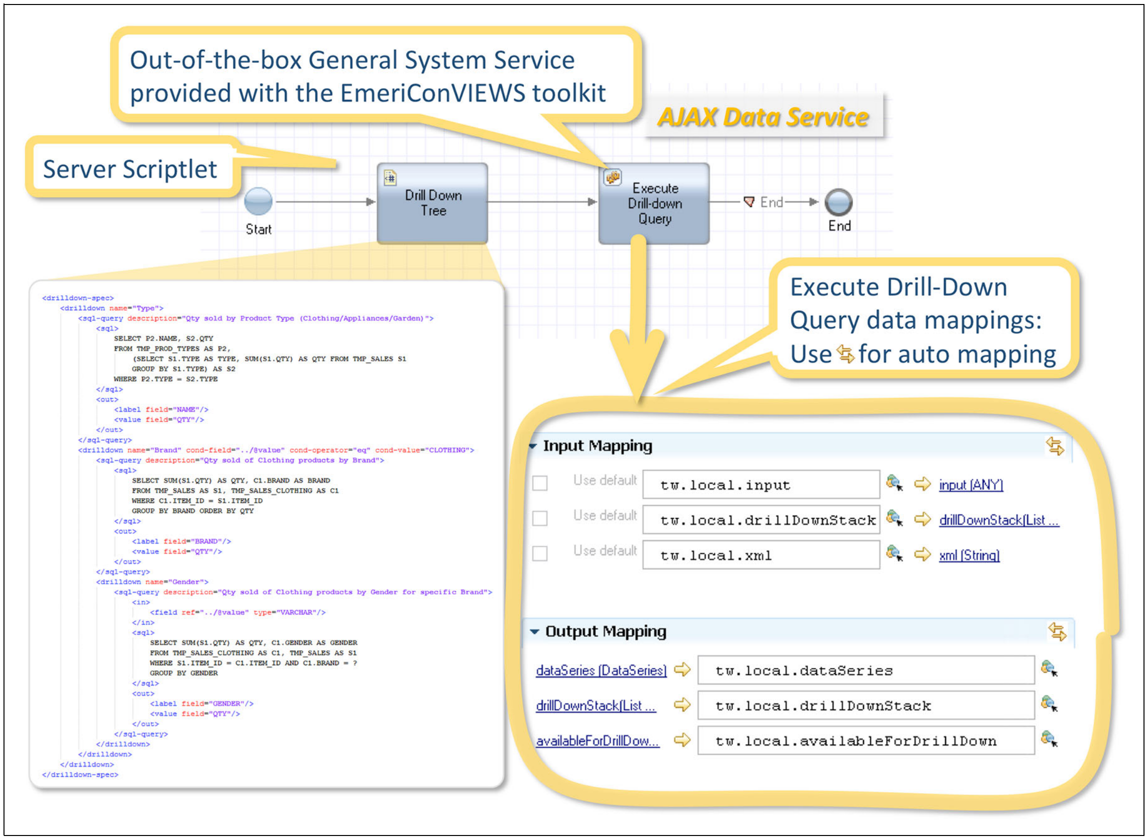
```
<drilldown-spec>
    <drilldown name="Type">
        <sql-query description="Qty sold by Product Type (Clothing/Appliances/Garden)">
            <sql>
                SELECT P2.NAME, S2.QTY
                FROM TMP_PROD_TYPES AS P2,
                    (SELECT S1.TYPE AS TYPE, SUM(S1.QTY) AS QTY FROM TMP_SALES S1
                    GROUP BY S1.TYPE) AS S2
                WHERE P2.TYPE = S2.TYPE
            </sql>
            <out>
                <label field="NAME"/>
                <value field="QTY"/>
            </out>
        </sql-query>
        <drilldown name="Brand" cond-field="../@value" cond-operator="eq" cond-value="CLOTHING">
            <sql-query description="Qty sold of Clothing products by Brand">
                <sql>
                    SELECT SUM(S1.QTY) AS QTY, C1.BRAND AS BRAND
                    FROM TMP_SALES AS S1, TMP_SALES_CLOTHING AS C1
                    WHERE C1.ITEM_ID = S1.ITEM_ID
                    GROUP BY BRAND ORDER BY QTY
                </sql>
                <out>
                    <label field="BRAND"/>
                    <value field="QTY"/>
                </out>
            </sql-query>
            <drilldown name="Gender">
                <sql-query description="Qty sold of Clothing products by Gender for specific Brand">
                    <in>
                        <field ref="../@value" type="VARCHAR"/>
                    </in>
                    <sql>
                        SELECT SUM(S1.QTY) AS QTY, C1.GENDER AS GENDER
                        FROM TMP_SALES_CLOTHING AS C1, TMP_SALES AS S1
                        WHERE S1.ITEM_ID = C1.ITEM_ID AND C1.BRAND = ?
                        GROUP BY GENDER
                    </sql>
                    <out>
                        <label field="GENDER"/>
                        <value field="QTY"/>
                    </out>
                </sql-query>
            </drilldown>
        </drilldown>
    </drilldown>
</drilldown-spec>
```

Figure 7-60 Drill-down tree specification example

Note how the structure of the XML drill-down tree mirrors the structure of the conceptual drill-down path. The tree, as specified above, is almost ready to execute. Note also how the name attribute of each `<drilldown>` element gives the chart the hint that is needed to display which dimensions are available to drill into next (see dashed arrows in Figure 7-60).

In the previous example, the first two drill-down elements only have one immediate child drill-down, but this is only for compactness of the example. In practice, a drill-down element can have several child drill-down elements (as shown, for example, in the middle chart of Figure 7-57 on page 301), resulting in chart behaviors providing more extensive data analysis capabilities.

When combined with the *Execute Drill-down Query* service (see Figure 7-61), the XML drill-down tree specification above is all the development that is needed to create a drill-down-capable service. The XML tree above can be added to a Server Scriptlet in the Ajax Service that is attached to the chart. The service Figure 7-61 shows how the Ajax service in implemented.



Out-of-the-box General System Service provided with the EmeriConVIEWS toolkit
AJAX Data Service
Server Scriptlet
Start
Drill Down Tree
Execute Drill-down Query
End
End
Execute Drill-Down Query data mappings: Use ⇆ for auto mapping

```
<drilldown-spec>
    <drilldown name="Type">
        <sql-query description="Qty sold by Product Type (Clothing/Appliances/Garden)">
            <sql>
                SELECT P2.NAME, S2.QTY
                FROM TMP_PROD_TYPES AS P2,
                    (SELECT S1.TYPE AS TYPE, SUM(S1.QTY) AS QTY FROM TMP_SALES S1
                    GROUP BY S1.TYPE) AS S2
                WHERE P2.TYPE = S2.TYPE
            </sql>
            <out>
                <label field="NAME"/>
                <value field="QTY"/>
            </out>
        </sql-query>
        <drilldown name="Brand" cond-field="../@value" cond-operator="eq" cond-value="CLOTHING">
            <sql-query description="Qty sold of Clothing products by Brand">
                <sql>
                    SELECT SUM(S1.QTY) AS QTY, C1.BRAND AS BRAND
                    FROM TMP_SALES AS S1, TMP_SALES_CLOTHING AS C1
                    WHERE C1.ITEM_ID = S1.ITEM_ID
                    GROUP BY BRAND ORDER BY QTY
                </sql>
                <out>
                    <label field="BRAND"/>
                    <value field="QTY"/>
                </out>
            </sql-query>
            <drilldown name="Gender">
                <sql-query description="Qty sold of Clothing products by Gender for specific Brand">
                    <in>
                        <field ref="../@value" type="VARCHAR"/>
                    </in>
                    <sql>
                        SELECT SUM(S1.QTY) AS QTY, C1.GENDER AS GENDER
                        FROM TMP_SALES_CLOTHING AS C1, TMP_SALES AS S1
                        WHERE S1.ITEM_ID = C1.ITEM_ID AND C1.BRAND = ?
                        GROUP BY GENDER
                    </sql>
                    <out>
                        <label field="GENDER"/>
                        <value field="QTY"/>
                    </out>
                </sql-query>
            </drilldown>
        </drilldown>
    </drilldown>
</drilldown-spec>
```

Input Mapping

☐ Use default	tw.local.input	⇨	input (ANY)
☐ Use default	tw.local.drillDownStack	⇨	drillDownStack(List ...
☐ Use default	tw.local.xml	⇨	xml (String)

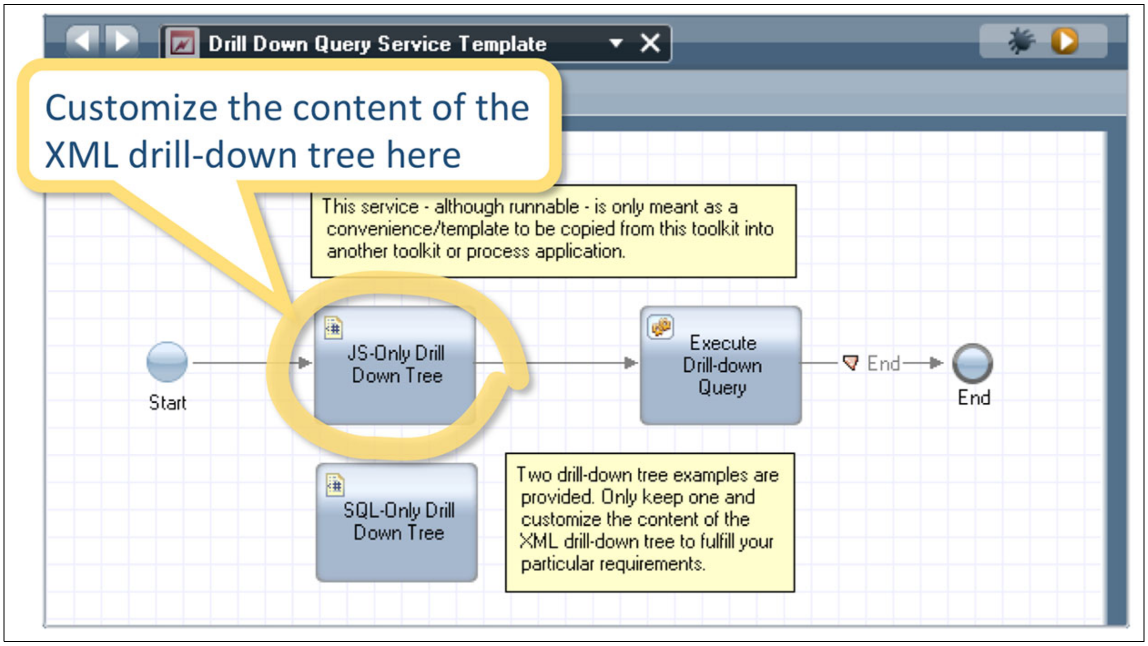
Output Mapping

dataSeries (DataSeries)	⇨	tw.local.dataSeries
drillDownStack(List ...	⇨	tw.local.drillDownStack
availableForDrillDow...	⇨	tw.local.availableForDrillDown

Figure 7-61 Drill-down tree in context of Ajax service

For convenience, the *Drill Down Query Template Service Template* service is provided in the EmeriConVIEWS toolkit to be copied to the wanted Toolkit or Process Application.

When copied, the service (shown in Figure 7-62 on page 307) can be customized by changing the content of the drill-down tree. No other change is required unless more sophisticated processing (for example Web Service integration) is needed.


Drill Down Query Service Template
Customize the content of the XML drill-down tree here
Start
JS-Only Drill Down Tree
Execute Drill-down Query
End
End
SQL-Only Drill Down Tree

This service - although runnable - is only meant as a convenience/template to be copied from this toolkit into another toolkit or process application.

Two drill-down tree examples are provided. Only keep one and customize the content of the XML drill-down tree to fulfill your particular requirements.

Figure 7-62 Drill-down Query Service Template content

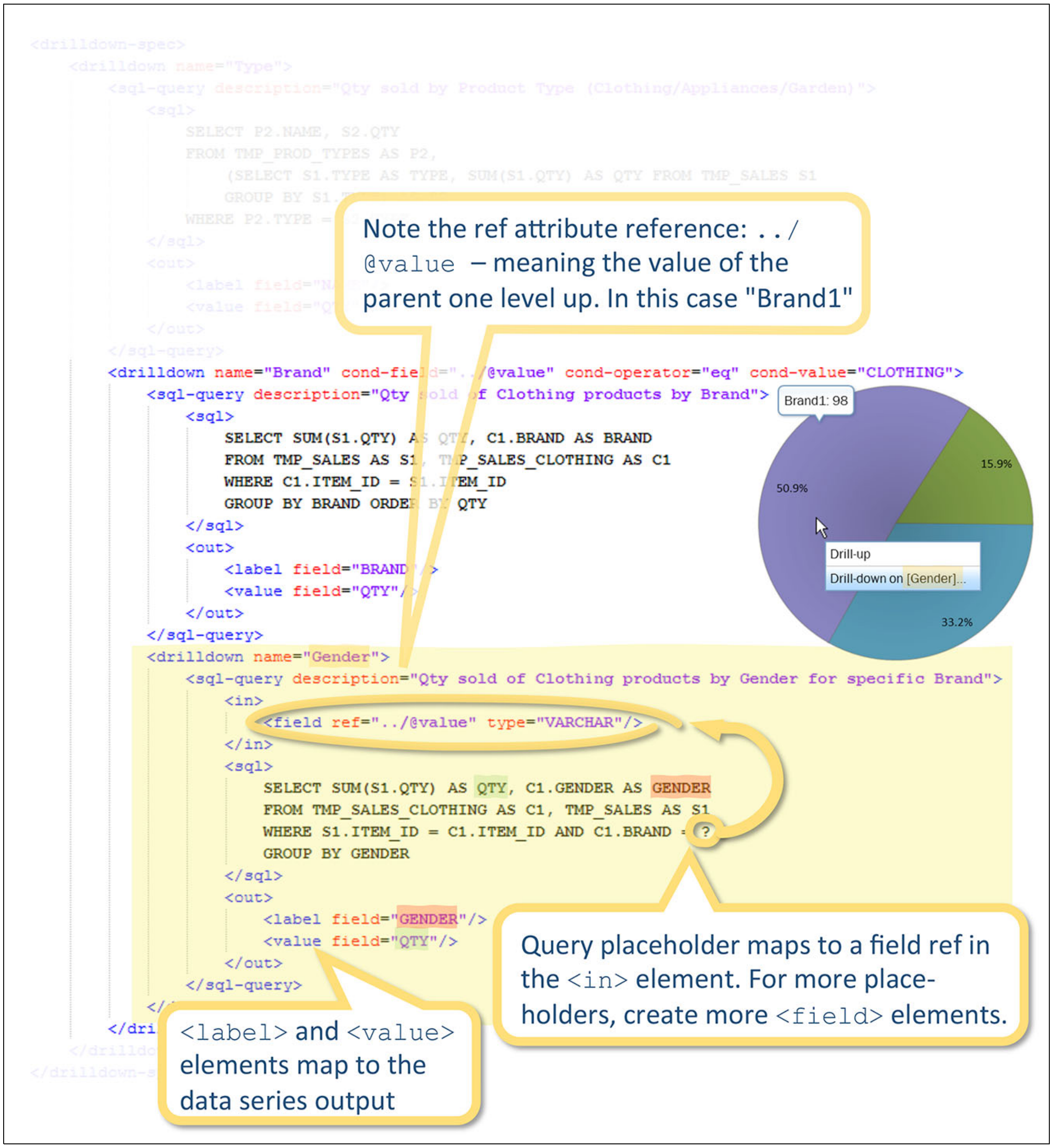
Drill-down tree basics

A drill-down tree is an XML document made of embedded `<drilldown>` elements. The `<drilldown>` elements are arranged in such a way as to replicate the structure of the various drill-down paths that are needed to fulfill data analysis requirements. A `<drilldown>` element can contain one `<sql-query>` or one `<js-query>` element, and zero or more child `<drilldown>` elements.

Figure 7-63 on page 308 shows:

- Where SQL queries are specified in the tree
- How drill-down values are referenced and inserted into the queries
- How the output of SQL queries is mapped to the returned data series

The emphasis/point of interest is on the yellow-highlighted drill-down element.



```
<drilldown-spec>
    <drilldown name="Type">
        <sql-query description="Qty sold by Product Type (Clothing/Appliances/Garden)">
            <sql>
                SELECT P2.NAME, S2.QTY
                FROM TMP_PROD_TYPES AS P2,
                    (SELECT S1.TYPE AS TYPE, SUM(S1.QTY) AS QTY FROM TMP_SALES S1
                    GROUP BY S1.T
                WHERE P2.TYPE =
            </sql>
            <out>
                <label field="N
                <value field="Q
            </out>
        </sql-query>
        <drilldown name="Brand" cond-field="../@value" cond-operator="eq" cond-value="CLOTHING">
            <sql-query description="Qty sold of Clothing products by Brand">
                <sql>
                    SELECT SUM(S1.QTY) AS QTY, C1.BRAND AS BRAND
                    FROM TMP_SALES AS S1, TMP_SALES_CLOTHING AS C1
                    WHERE C1.ITEM_ID = S1.ITEM_ID
                    GROUP BY BRAND ORDER BY QTY
                </sql>
                <out>
                    <label field="BRAND"/>
                    <value field="QTY"/>
                </out>
            </sql-query>
            <drilldown name="Gender">
                <sql-query description="Qty sold of Clothing products by Gender for specific Brand">
                    <in>
                        <field ref="../@value" type="VARCHAR"/>
                    </in>
                    <sql>
                        SELECT SUM(S1.QTY) AS QTY, C1.GENDER AS GENDER
                        FROM TMP_SALES_CLOTHING AS C1, TMP_SALES AS S1
                        WHERE S1.ITEM_ID = C1.ITEM_ID AND C1.BRAND = ?
                        GROUP BY GENDER
                    </sql>
                    <out>
                        <label field="GENDER"/>
                        <value field="QTY"/>
                    </out>
                </sql-query>
            </
        </dri
    </drilldo
</drilldown-s
```

Note the ref attribute reference: `../@value` – meaning the value of the parent one level up. In this case "Brand1"

Query placeholder maps to a field ref in the `<in>` element. For more place-holders, create more `<field>` elements.

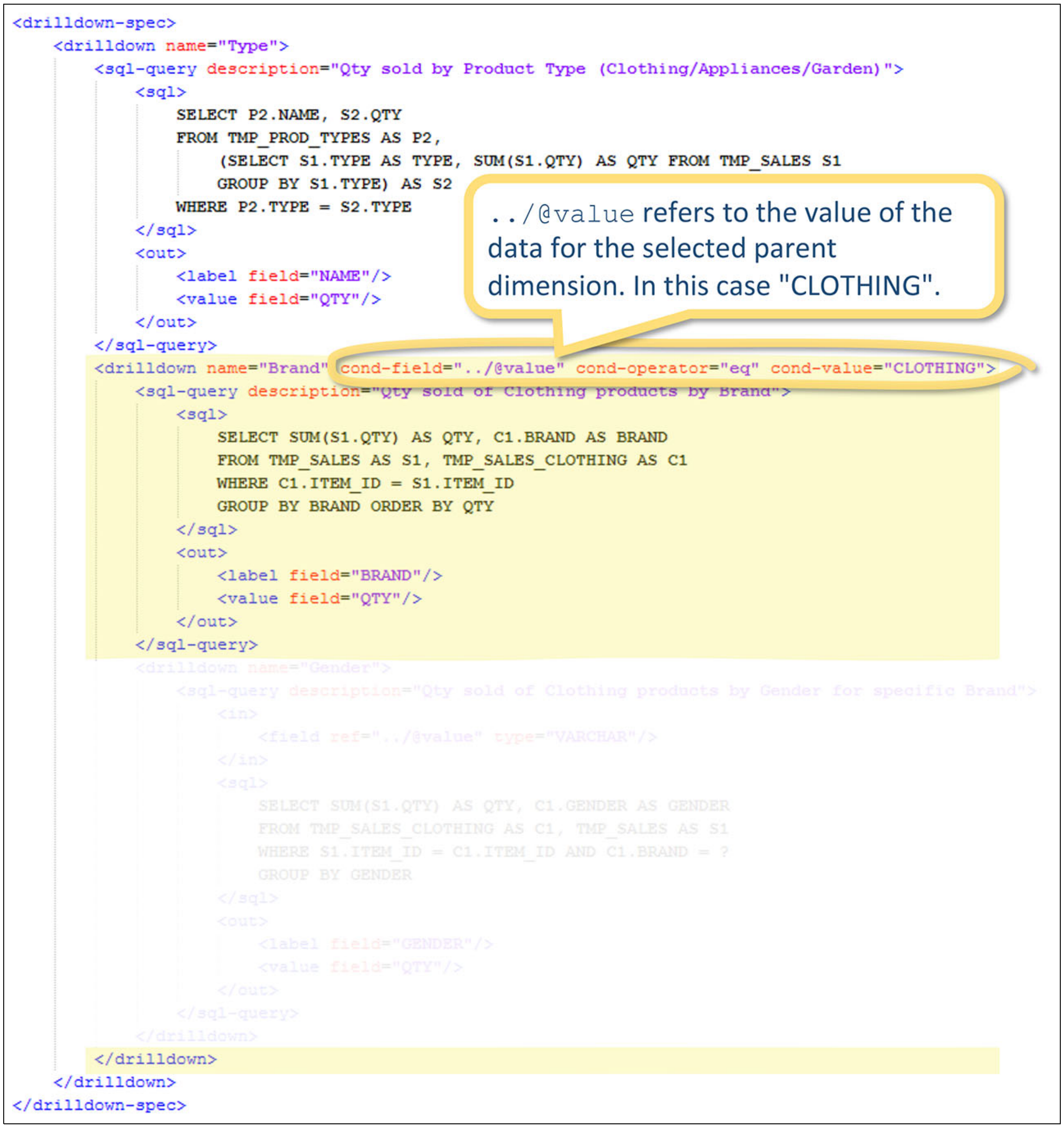
`<label>` and `<value>` elements map to the data series output

Figure 7-63 Drill-down tree detail

One last concept is needed to complete the drill-down picture: *Conditional* drill-downs elements. A conditional element allows the drill-down interaction to take a different direction in the drill-down path based on the value of a selected slice/bar/data point in the chart instead of just the dimension selected.

For example, Figure 7-58 on page 302 shows how a different path is taken if the value of the Type of Sales dimension equals “CLOTHING”.

Those conditions are expressed on a `<drilldown>` element as shown in Figure 7-64 on page 310.



```
<drilldown-spec>
    <drilldown name="Type">
        <sql-query description="Qty sold by Product Type (Clothing/Appliances/Garden)">
            <sql>
                SELECT P2.NAME, S2.QTY
                FROM TMP_PROD_TYPES AS P2,
                    (SELECT S1.TYPE AS TYPE, SUM(S1.QTY) AS QTY FROM TMP_SALES S1
                    GROUP BY S1.TYPE) AS S2
                WHERE P2.TYPE = S2.TYPE
            </sql>
            <out>
                <label field="NAME"/>
                <value field="QTY"/>
            </out>
        </sql-query>
        <drilldown name="Brand" cond-field="../@value" cond-operator="eq" cond-value="CLOTHING">
            <sql-query description="Qty sold of Clothing products by Brand">
                <sql>
                    SELECT SUM(S1.QTY) AS QTY, C1.BRAND AS BRAND
                    FROM TMP_SALES AS S1, TMP_SALES_CLOTHING AS C1
                    WHERE C1.ITEM_ID = S1.ITEM_ID
                    GROUP BY BRAND ORDER BY QTY
                </sql>
                <out>
                    <label field="BRAND"/>
                    <value field="QTY"/>
                </out>
            </sql-query>
            <drilldown name="Gender">
                <sql-query description="Qty sold of Clothing products by Gender for specific Brand">
                    <in>
                        <field ref="../@value" type="VARCHAR"/>
                    </in>
                    <sql>
                        SELECT SUM(S1.QTY) AS QTY, C1.GENDER AS GENDER
                        FROM TMP_SALES_CLOTHING AS C1, TMP_SALES AS S1
                        WHERE S1.ITEM_ID = C1.ITEM_ID AND C1.BRAND = ?
                        GROUP BY GENDER
                    </sql>
                    <out>
                        <label field="GENDER"/>
                        <value field="QTY"/>
                    </out>
                </sql-query>
            </drilldown>
        </drilldown>
    </drilldown>
</drilldown-spec>
```

`../@value` refers to the value of the data for the selected parent dimension. In this case "CLOTHING".

Figure 7-64 Drill-down tree detail: Conditional drill-down elements

The condition is expressed as on the drill-down element as attributes:

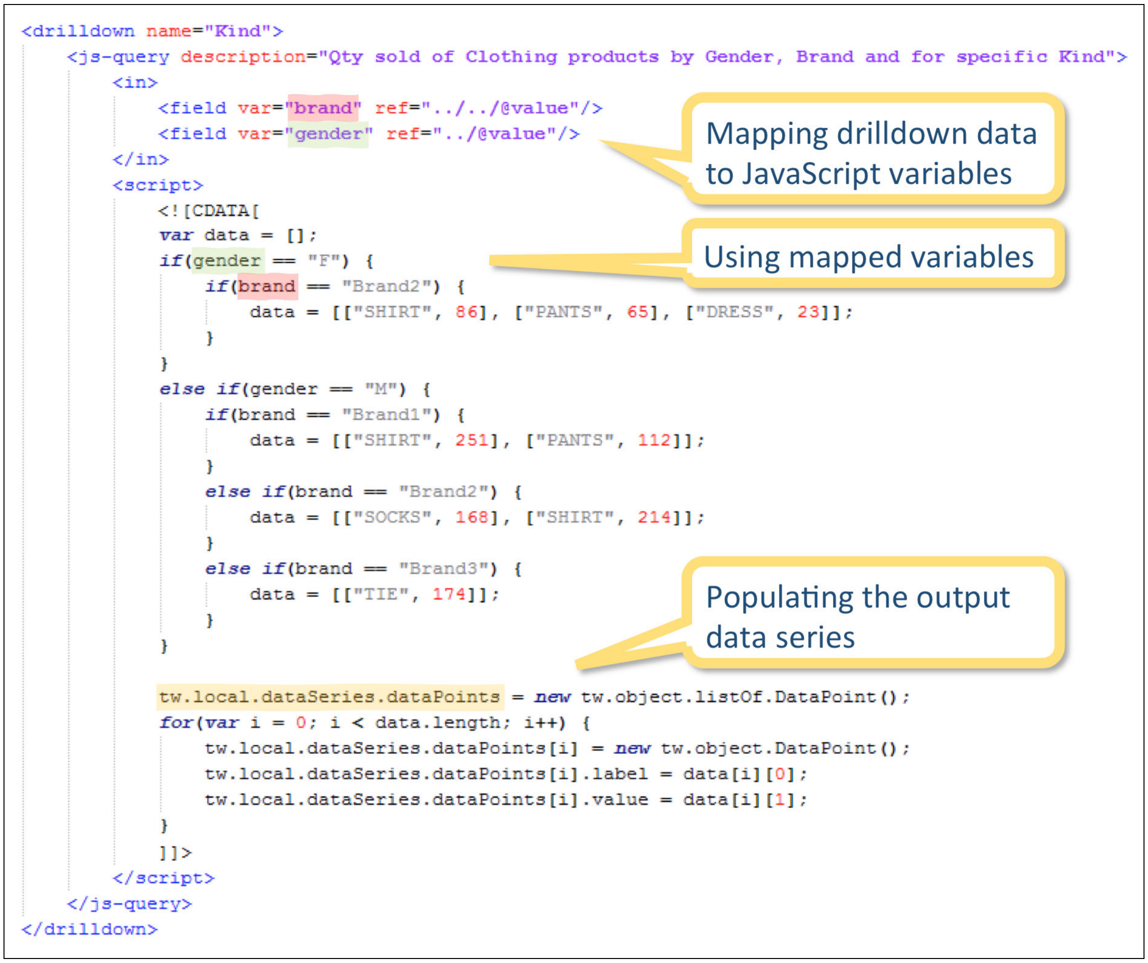
- `cond-field="../@value"` (meaning value of parent drill-down selection: use "`../../@value`" for the value of the grand-parent drill-down, and so on)

- `cond-operator=”eq”` (meaning equals)
- `cond-value=”CLOTHING”` the literal value against which to test the condition

Practically-speaking, the conditional selector on a drill-down element helps control what drill-down choices are displayed on the chart given the value of the currently-selected dimension.

JavaScript drill-down queries

JavaScript queries can also be specified on a drill-down element (a particular drill-down element can only have one query style or the other, not both). The sample illustration that is shown in Figure 7-65 shows a JavaScript query using inputs and the populated output data series (the logic below simply builds a data series based on mocked-up data).



```
<drilldown name="Kind">
    <js-query description="Qty sold of Clothing products by Gender, Brand and for specific Kind">
        <in>
            <field var="brand" ref="../../@value"/>
            <field var="gender" ref="../@value"/>
        </in>
        <script>
            <![CDATA[
            var data = [];
            if(gender == "F") {
                if(brand == "Brand2") {
                    data = [["SHIRT", 86], ["PANTS", 65], ["DRESS", 23]];
                }
            }
            else if(gender == "M") {
                if(brand == "Brand1") {
                    data = [["SHIRT", 251], ["PANTS", 112]];
                }
                else if(brand == "Brand2") {
                    data = [["SOCKS", 168], ["SHIRT", 214]];
                }
                else if(brand == "Brand3") {
                    data = [["TIE", 174]];
                }
            }

            tw.local.dataSeries.dataPoints = new tw.object.listOf.DataPoint();
            for(var i = 0; i < data.length; i++) {
                tw.local.dataSeries.dataPoints[i] = new tw.object.DataPoint();
                tw.local.dataSeries.dataPoints[i].label = data[i][0];
                tw.local.dataSeries.dataPoints[i].value = data[i][1];
            }
            ]]>
        </script>
    </js-query>
</drilldown>
```

Mapping drilldown data to JavaScript variables
Using mapped variables
Populating the output data series

Figure 7-65 JavaScript drill-down query example

Summary

EmeriConVIEWS charts are versatile controls with events, methods, configuration options, auto-refresh, and analytics capabilities that help create simple to sophisticated reporting capabilities in a BPM solution. Charts can become aware of, and react to, each other with minimal effort. Combined with the Ajax Service-backed Fast Data Table control, one can create roll-up and detailed views on a reporting dashboard.

Charts are not limited to Performance Data Warehouse-based reporting since the Ajax Data Series services to which they are attached can query other SQL data sources, or can use JavaScript logic to populate the data series they produce.

Signature

The Signature control provides a canvas that can be used to capture a human signature. It can be used for scenarios where, for example, a signature is captured during one step in the process, then stored or rendered at a later step.

The BPD fragment that is shown in Figure 7-66 on page 313 illustrates such a scenario.

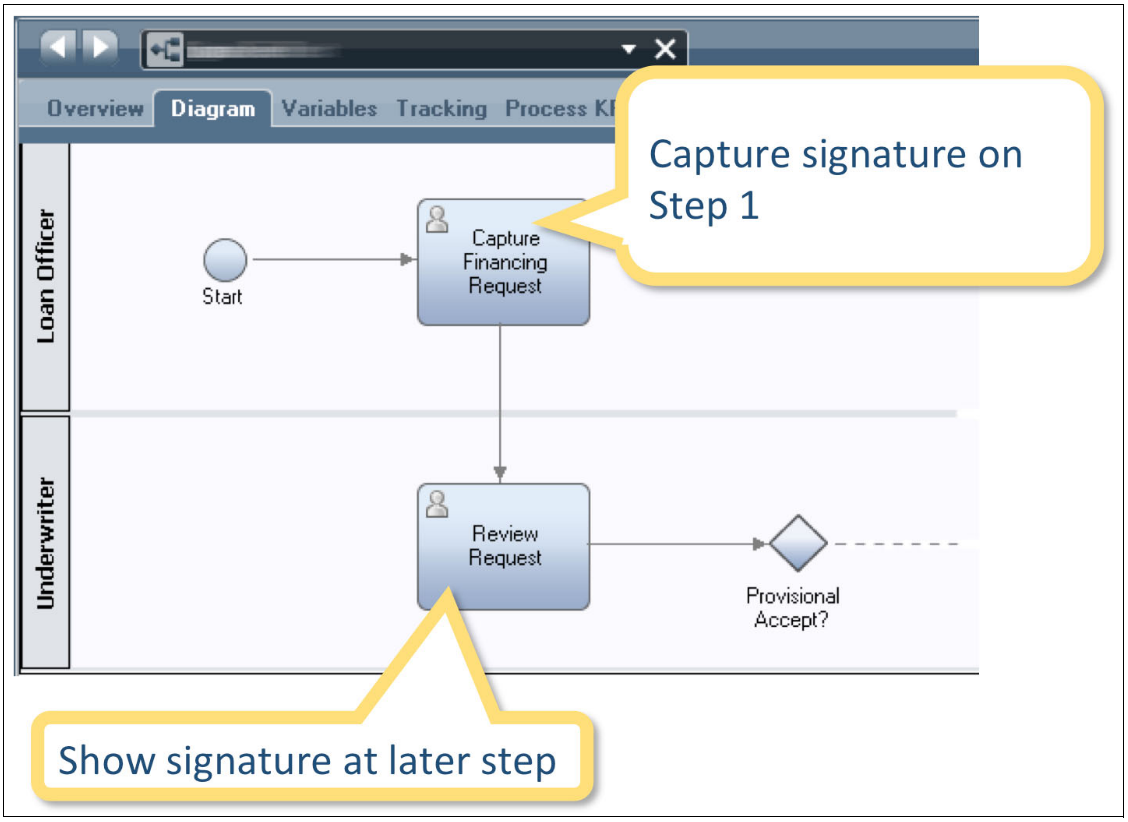


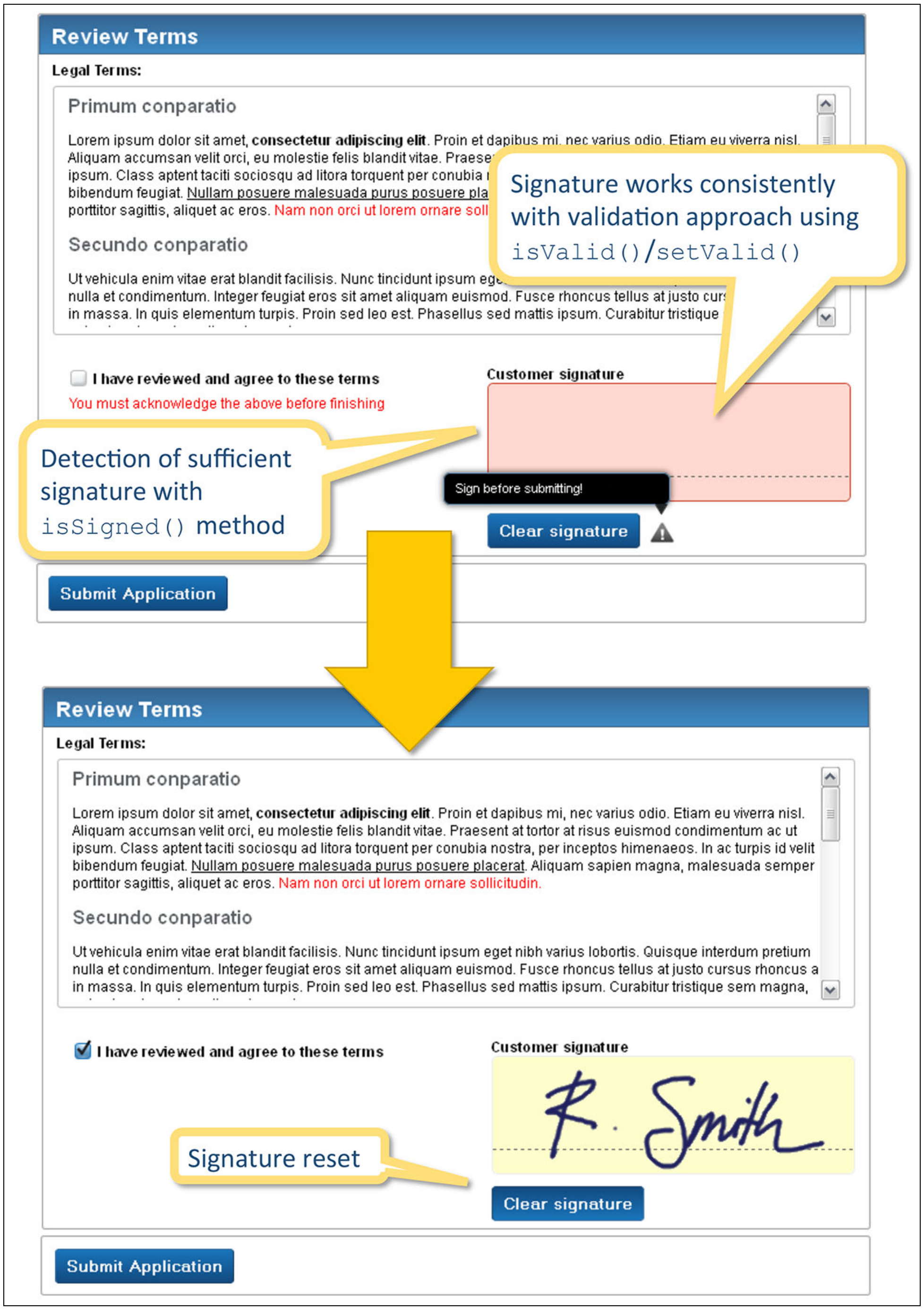
Figure 7-66 Sample process scenario for signatures

The signature data is stored in bound data in the Coach (which can eventually be stored in process data) and can be redisplayed later.

Signature controls are placed on the canvas and are typically bound to data (of the string type). Like other EmeriConVIEWS controls, the Signature Coach View includes a number of methods, such as:

- setValid() / isValid()
- isSigned() which determines if the signature in the canvas is sufficiently long
- clear() which programmatically clears the signature

Figure 7-67 on page 314 shows the behavior of Signature control for capturing signatures at run time.


Signature works consistently with validation approach using isValid()/setValid()
Detection of sufficient signature with isSigned() method
Signature reset
Sign before submitting!
Clear signature
Submit Application

Review Terms

Legal Terms:

Primum conparatio

Lorem ipsum dolor sit amet, **consectetur adipiscing elit**. Proin et dapibus mi, nec varius odio. Etiam eu viverra nisl. Aliquam accumsan velit orci, eu molestie felis blandit vitae. Praese
ipsum. Class aptent taciti sociosqu ad litora torquent per conubia n
bibendum feugiat. Nullam posuere malesuada purus posuere pla
porttitor sagittis, aliquet ac eros. Nam non orci ut lorem ornare soll

Secundo conparatio

Ut vehicula enim vitae erat blandit facilisis. Nunc tincidunt ipsum ege
nulla et condimentum. Integer feugiat eros sit amet aliquam euismod. Fusce rhoncus tellus at justo curs
in massa. In quis elementum turpis. Proin sed leo est. Phasellus sed mattis ipsum. Curabitur tristique

☐ I have reviewed and agree to these terms

You must acknowledge the above before finishing

Customer signature

Review Terms

Legal Terms:

Primum conparatio

Lorem ipsum dolor sit amet, **consectetur adipiscing elit**. Proin et dapibus mi, nec varius odio. Etiam eu viverra nisl. Aliquam accumsan velit orci, eu molestie felis blandit vitae. Praesent at tortor at risus euismod condimentum ac ut ipsum. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. In ac turpis id velit bibendum feugiat. Nullam posuere malesuada purus posuere placerat. Aliquam sapien magna, malesuada semper porttitor sagittis, aliquet ac eros. Nam non orci ut lorem ornare sollicitudin.

Secundo conparatio

Ut vehicula enim vitae erat blandit facilisis. Nunc tincidunt ipsum eget nibh varius lobortis. Quisque interdum pretium nulla et condimentum. Integer feugiat eros sit amet aliquam euismod. Fusce rhoncus tellus at justo cursus rhoncus a in massa. In quis elementum turpis. Proin sed leo est. Phasellus sed mattis ipsum. Curabitur tristique sem magna,

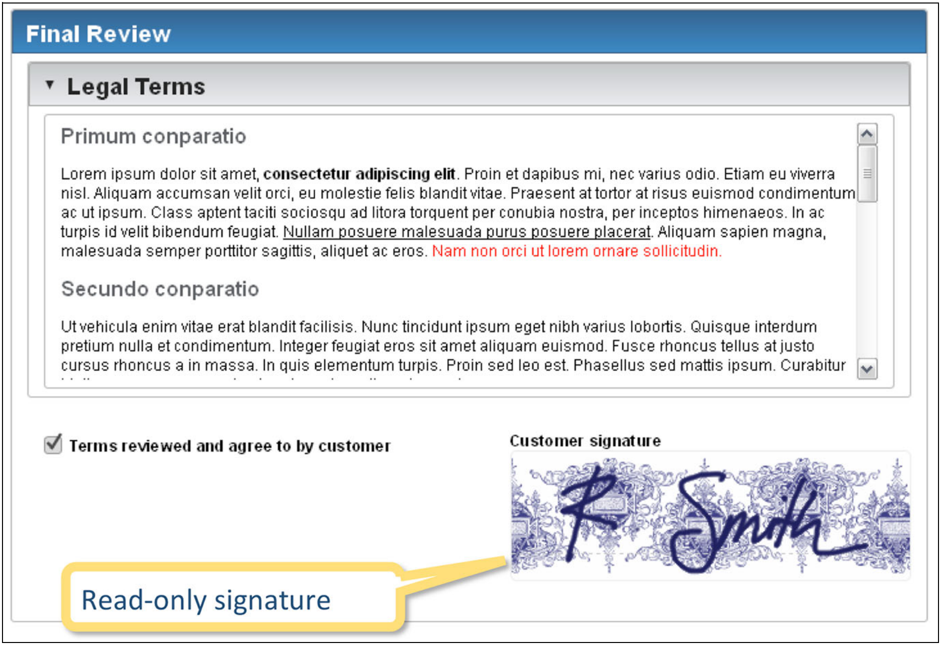
☑ I have reviewed and agree to these terms

Customer signature

R. Smith

Figure 7-67 Runtime example for using Signature control to capture a signature

Once the signature is captured, it can be viewed at a later step in the process, as shown in Figure 7-68.



Final Review

Legal Terms

Primum conparatio

Lorem ipsum dolor sit amet, **consectetur adipiscing elit**. Proin et dapibus mi, nec varius odio. Etiam eu viverra nisl. Aliquam accumsan velit orci, eu molestie felis blandit vitae. Praesent at tortor at risus euismod condimentum ac ut ipsum. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. In ac turpis id velit bibendum feugiat. Nullam posuere malesuada purus posuere placerat. Aliquam sapien magna, malesuada semper porttitor sagittis, aliquet ac eros. Nam non orci ut lorem ornare sollicitudin.

Secundo conparatio

Ut vehicula enim vitae erat blandit facilisis. Nunc tincidunt ipsum eget nibh varius lobortis. Quisque interdum pretium nulla et condimentum. Integer feugiat eros sit amet aliquam euismod. Fusce rhoncus tellus at justo cursus rhoncus a in massa. In quis elementum turpis. Proin sed leo est. Phasellus sed mattis ipsum. Curabitur

☑ Terms reviewed and agree to by customer

Customer signature

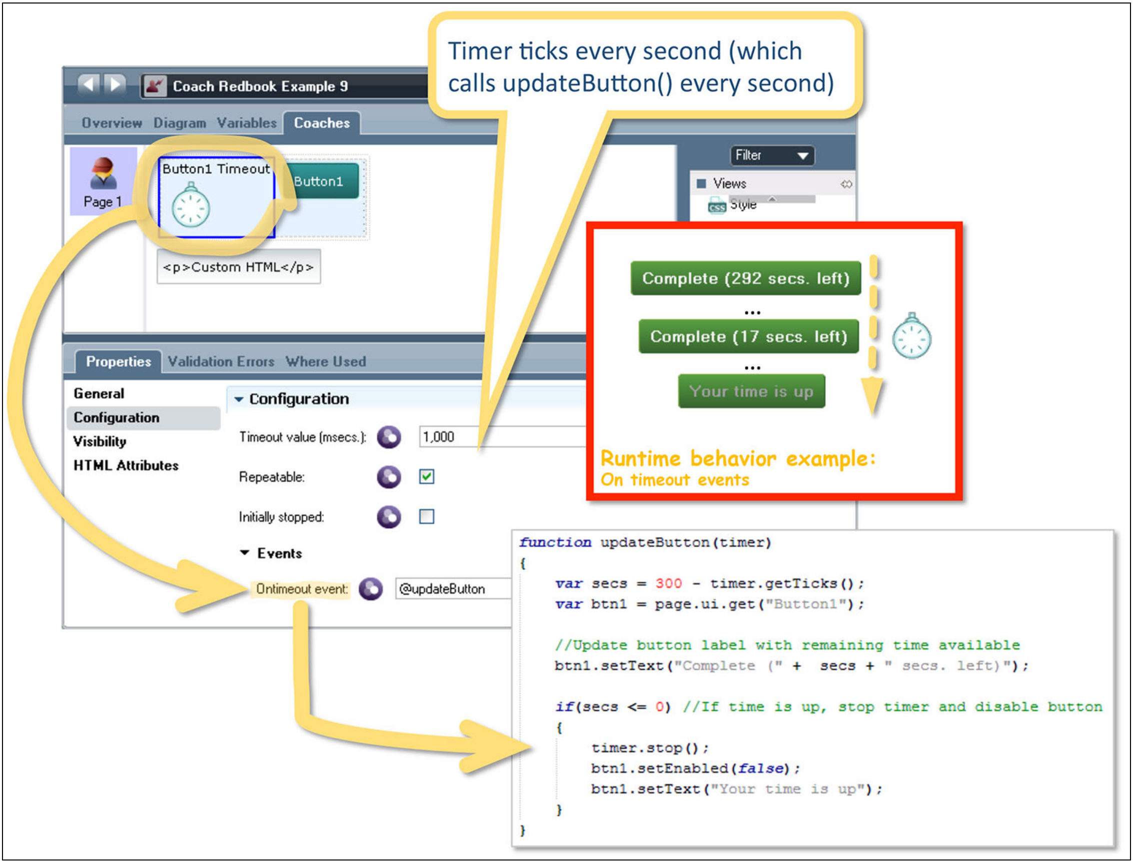
Figure 7-68 Runtime example for using Signature control to review a signature

Timer

The Timer control provides time-based triggering capabilities (such as regular updates of controls to reflect elapsed time, scheduled service calls, auto Coach completion after a certain amount of time) to Coaches.

Example

Figure 7-69 on page 316 shows how a button label is updated every second until a particular session expires.



Configuration

Timeout value (msecs.): 1,000

Repeatable: ✓

Initially stopped: ☐

Events

Ontimeout event: @updateButton

```
function updateButton(timer)
{
    var secs = 300 - timer.getTicks();
    var btn1 = page.ui.get("Button1");

    //Update button label with remaining time available
    btn1.setText("Complete (" +  secs + " secs. left)");

    if(secs <= 0) //If time is up, stop timer and disable button
    {
        timer.stop();
        btn1.setEnabled(false);
        btn1.setText("Your time is up");
    }
}
```

Figure 7-69 Usage example for Timer control

The Timer control provides methods such as start, stop, getTicks, resetTicks, setTimeout, isRunning to help the author formulate time-based logic using simple constructs.

Time-based Boundary Events

Timer controls can fire Boundary Events if:

- The event corresponding to the timer is wired in the Human Service diagram
- The logic called from the *on timeout* event does not return false

Events from the timer are treated the same way as other Boundary Events, as shown in Figure 7-70 on page 317.

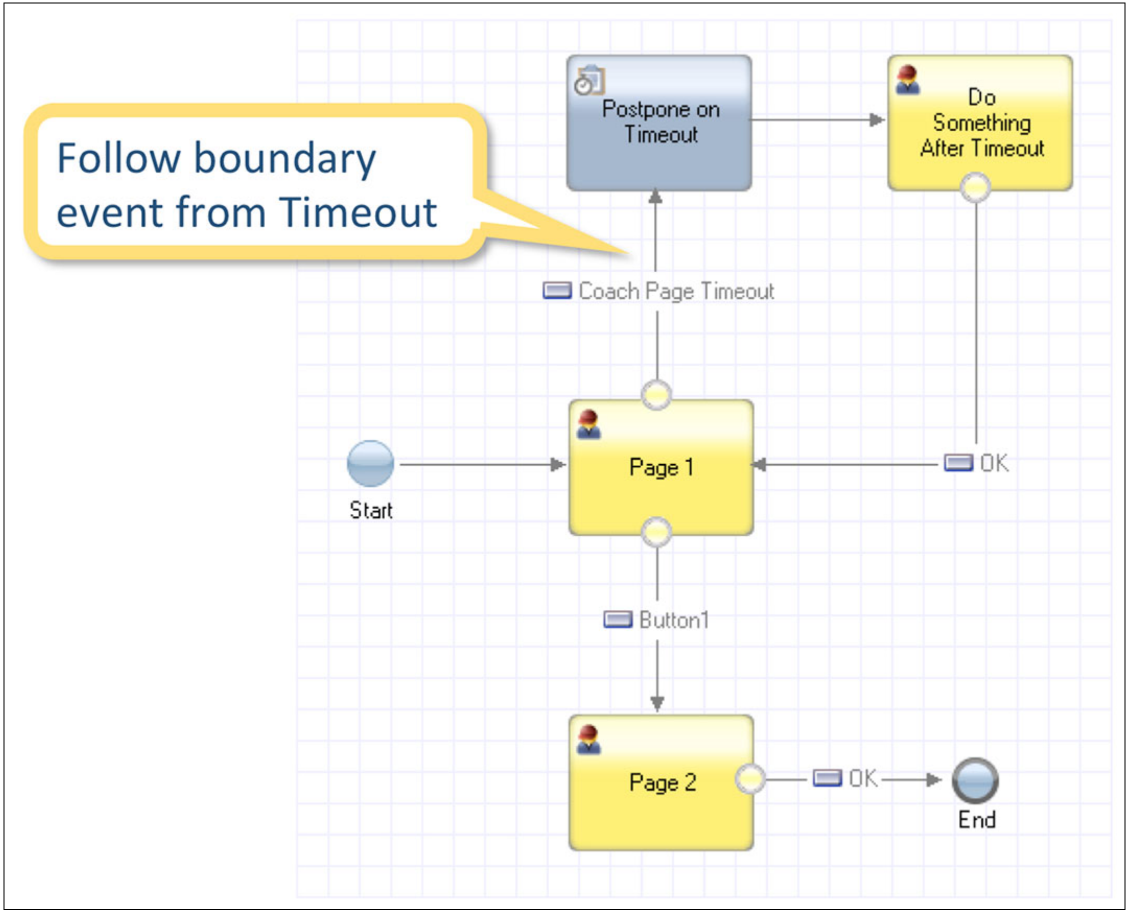


Figure 7-70 Boundary Events from Timer controls

Event subscription

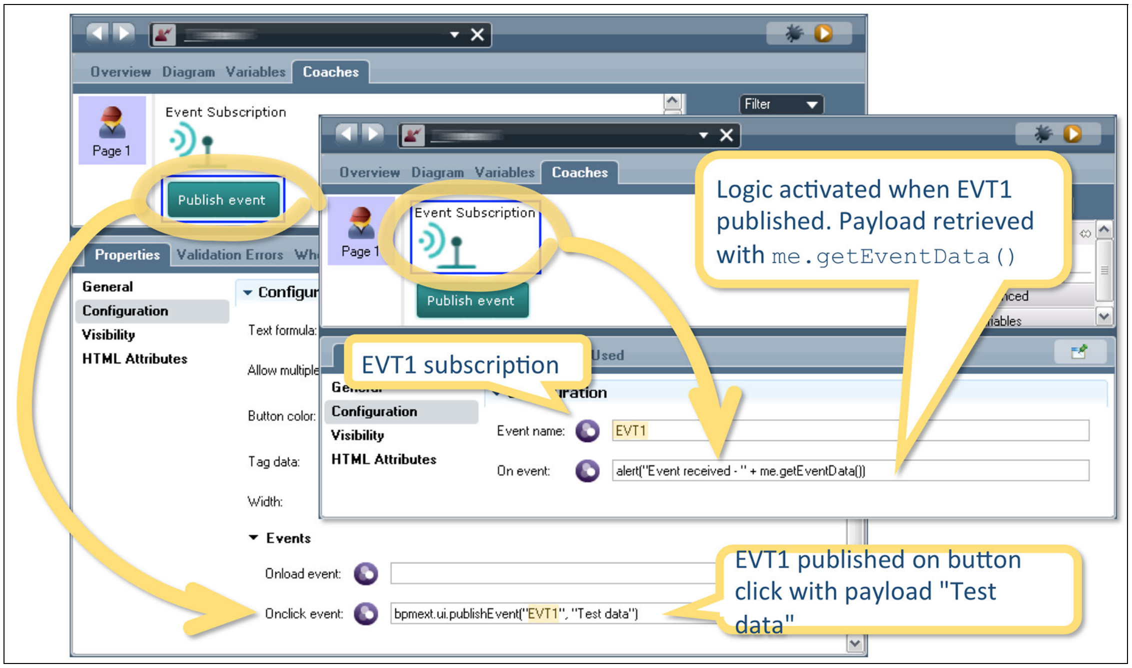
Sufficiently complex Coach UIs containing controls and subviews with interdependent behaviors can simplify and loosen those interrelationships through event publishing and subscribing. An author can publish an event using EmeriConVIEWS with the following call:

```
bpmext.ui.publishEvent("EVENT_NAME", "my payload");
```

The event name is used to notify subscribers. The type of the payload is only important if the publisher and the subscribers can agree on the structure of the payload data.

Event subscription control

The event subscription control can be placed at the Coach Page level or in a Coach View to listen for event matching the name to which it subscribes. When an event with a matching name is published, the *on event* event of the control is activated, as shown in Figure 7-71.



Overview Diagram Variables Coaches
Event Subscription
Page 1
Publish event
Properties Validation Errors
General
Configuration
Visibility
HTML Attributes
Logic activated when EVT1 published. Payload retrieved with me.getEventData()
EVT1 subscription
EVT1 published on button click with payload "Test data"
Text formula:
Allow multiple
Button color:
Tag data:
Width:
Events

Event name: EVT1

On event: alert("Event received - " + me.getEventData())

Onload event:

Onclick event: bpmext.ui.publishEvent("EVT1", "Test data")

Figure 7-71 Event Subscription control example

There can be any number of Event Subscription controls for the same event name in a Coach. The event subscription controls can be placed at the same level, or some on the Coach Page and some in Coach Views and subviews.

7.3 Responsive UI design for Coach Pages and Coach Views

Responsive Web Design is an important aspect of BPM Coaches targeted to several devices. A responsive UI addresses what happens when coaches that fit comfortably on a wide screen (where elements are large and often interacted with using a mouse pointer) are displayed on much smaller or narrower screens, such as mobile device screens, and where the pointer is replaced by fingers, as shown in Figure 7-72 on page 319.

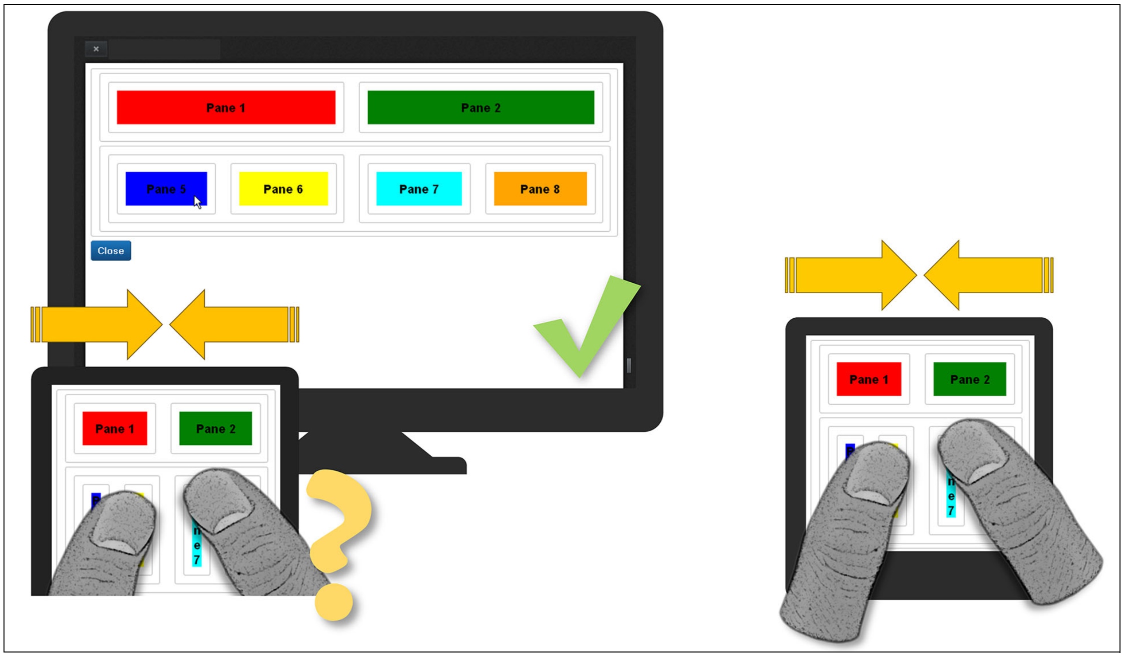


Figure 7-72 Usability challenge from wide screen to mobile screen coach UI

EmeriConVIEWS provides specific capabilities to detect media (screen) width and adjust the various controls on a Coach based on screen width change triggers.

Responsive Web Design components for EmeriConVIEWS

Designing web pages that properly rearrange themselves often requires the use of "canned" configurable layouts, and good knowledge of CSS to alter the styles of UI components to control size, visibility, and positioning (those being most common attributes).

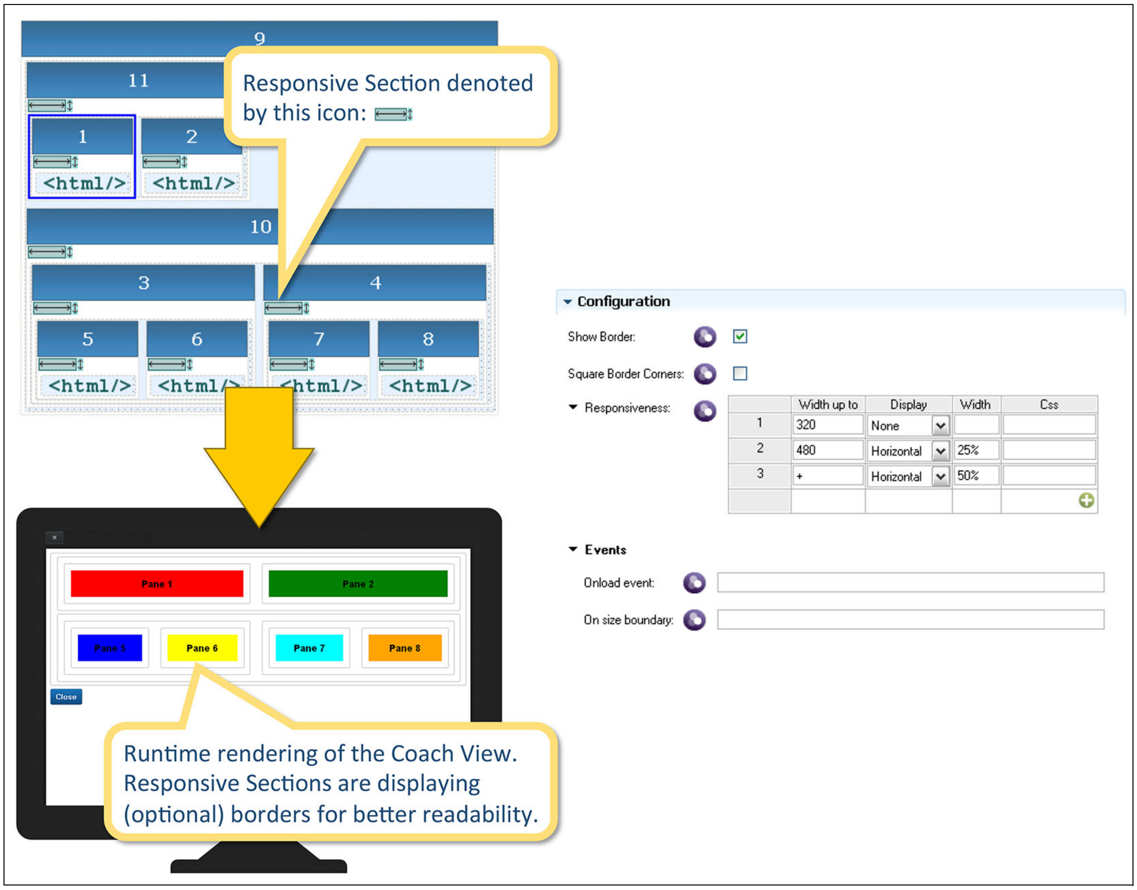
Instead of explicitly using CSS and media queries to design a responsive Coach Page or Coach View, EmeriConVIEWS provides a *Responsive Section* and a *Responsive Policy* control to fully implement responsive design.

The EmeriConVIEWS toolkit also includes a special device detection service for specialized cases when a completely different coach page should be displayed based on the device viewing the Human Service, or for when *GeoLocation* information needs to be used by the Human Service.

Responsive Section

The *Responsive Section* is the key enabler of responsive design in a Coach. Figure 7-73 shows a Coach Page initially organized to display the Coach on a wide screen.

For simplicity, plain HTML sections have been used in this example with different background colors to emphasize how the design of the Coach is reacting to width changes. In reality, each responsive section would contain one or more controls, according to the requirements of the particular Coach Page or View.


9
11
1
2
<html/>
10
3
4
5
6
7
8
Responsive Section denoted by this icon:
Pane 1
Pane 2
Pane 5
Pane 6
Pane 7
Pane 8
Close
Runtime rendering of the Coach View. Responsive Sections are displaying (optional) borders for better readability.

Configuration

Show Border: ✓

Square Border Corners: ☐

Responsiveness:

	Width up to	Display	Width	Css
1	320	None		
2	480	Horizontal	25%	
3	+	Horizontal	50%	

Events

Onload event:

On size boundary:

Figure 7-73 Usability challenge from wide screen to mobile screen coach UI

In Process Designer, authoring a Coach for responsiveness entails selecting a Responsive section, then specifying the width, display mode (and other styling if wanted) of the section at a particular width.

Figure 7-74 shows the Coach UI for three distinct form factor-related widths:

- Up to 320px
- Up to 480px
- Greater than 480px

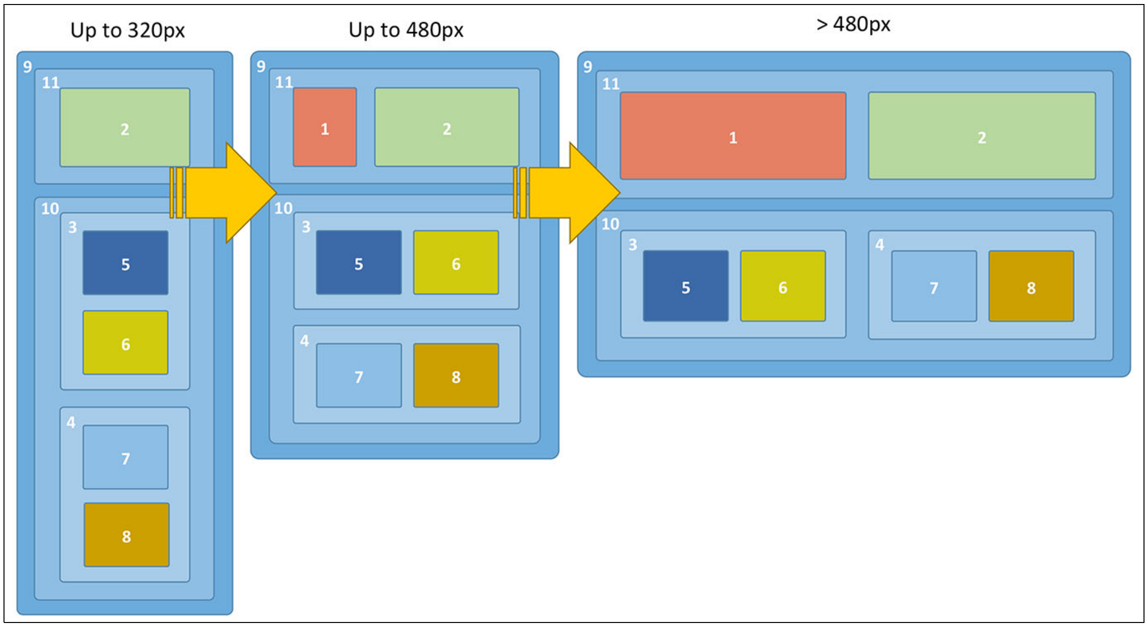
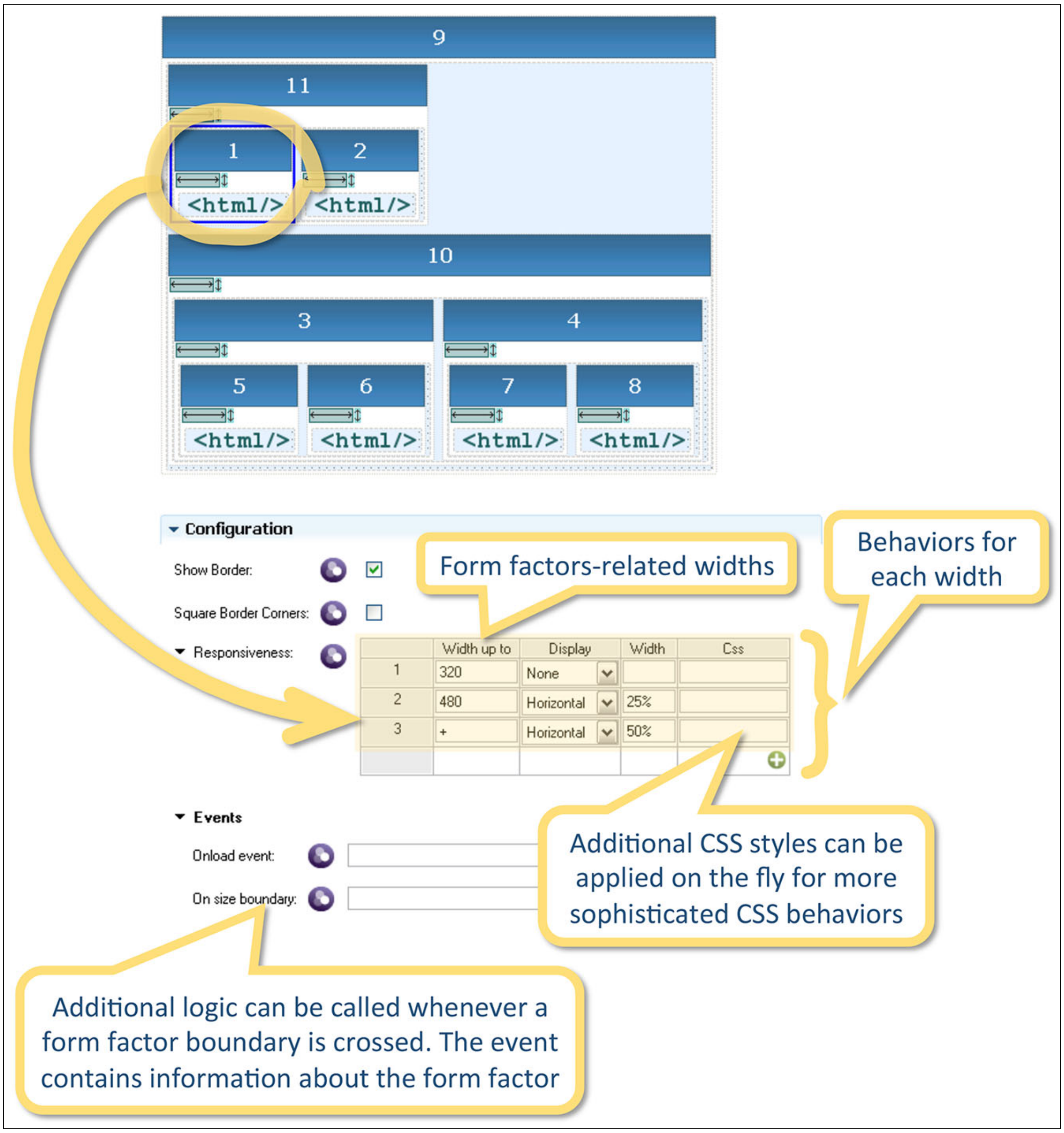


Figure 7-74 Responsive Design diagram for coach UI with three form factors

Translating this into the Process Designer authoring experience, for Section 1 (represented in red in Figure 7-74) for example, the author would select Section 1, then use the Responsive configuration options to specify styling at each distinct width, as shown in Figure 7-75 on page 322.


9
11
1
2
<html/>
<html/>
10
3
4
5
6
7
8
<html/>
<html/>
<html/>
<html/>
Form factors-related widths
Behaviors for each width
Additional CSS styles can be applied on the fly for more sophisticated CSS behaviors
Additional logic can be called whenever a form factor boundary is crossed. The event contains information about the form factor

Configuration

Show Border: ✓

Square Border Corners:

Responsiveness:

	Width up to	Display	Width	Css
1	320	None		
2	480	Horizontal	25%	
3	+	Horizontal	50%	

Events

Onload event:

On size boundary:

Figure 7-75 Authoring experience for Responsive Design in Process Designer

By repeating the procedure for each Responsive Section based on the three different form factors, the author enables the Coach UI with full responsive design capability, as shown in a web browser at run time, as shown in Figure 7-76 on page 323.

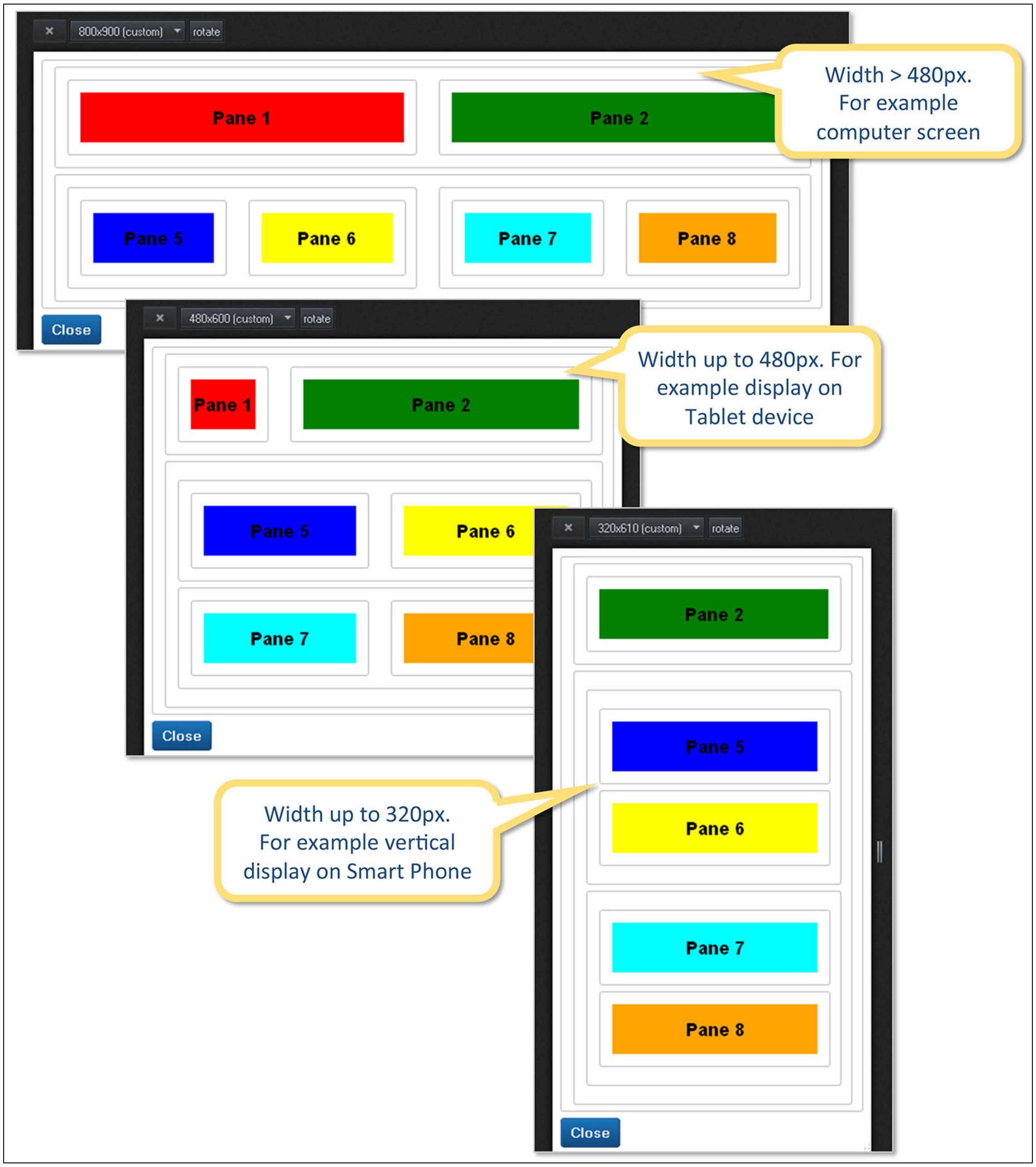


Figure 7-76 Responsive Coach UIs with EmeriConVIEWS

The responsive design authoring experience with EmeriConVIEWS is significantly streamlined compared to conventional approaches, but the example presented in Figure 7-76 on page 323 is deceptively simple.

A well thought-out, useful responsive UI requires planning upfront to determine how the UI will adapt at each width. This planning has little to do with Process Designer and entails plain UI responsiveness behavior planning. Meaning the form factors and the wanted arrangements for each form factor should be planned out before specifying responsive design parameters in Process Designer.

Figure 7-77 illustrates the preparatory planning that was performed before working in Process Designer to author the Coach.

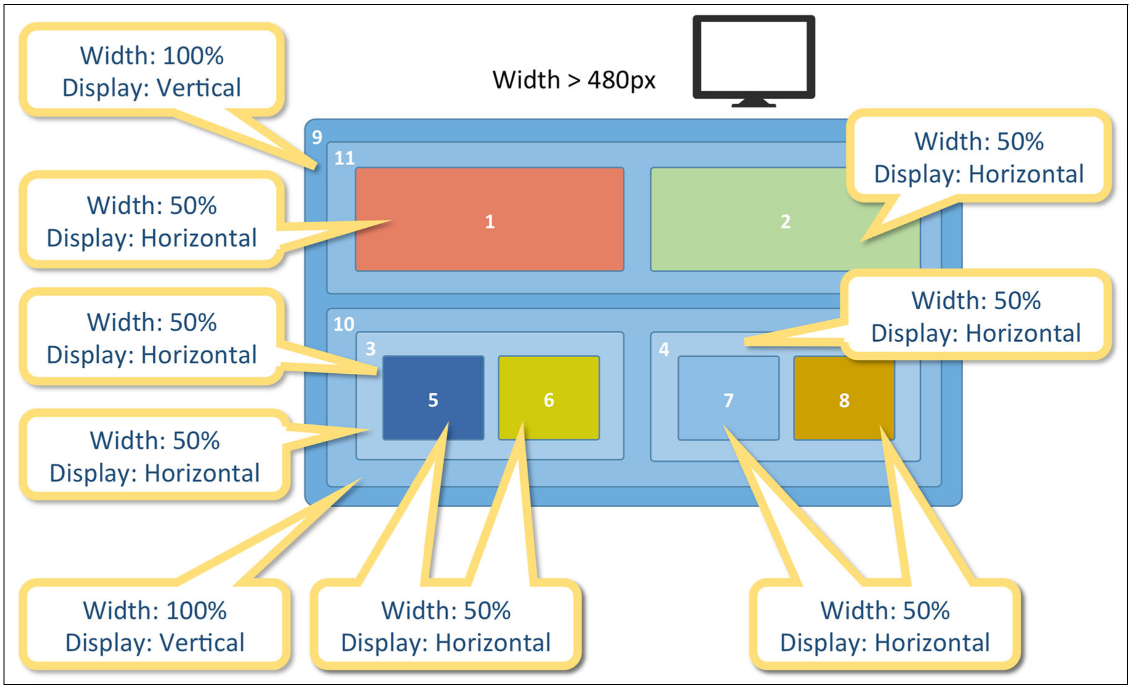

Width > 480px
Width: 100% Display: Vertical
Width: 50% Display: Horizontal
Width: 50% Display: Horizontal
Width: 50% Display: Horizontal
Width: 100% Display: Vertical
Width: 50% Display: Horizontal
Width: 50% Display: Horizontal
Width: 50% Display: Horizontal
Width: 50% Display: Horizontal
9
11
1
2
10
3
4
5
6
7
8

Figure 7-77 Planning for wide screen form factor

Figure 7-78 on page 325 shows the planning for a medium screen form factor.

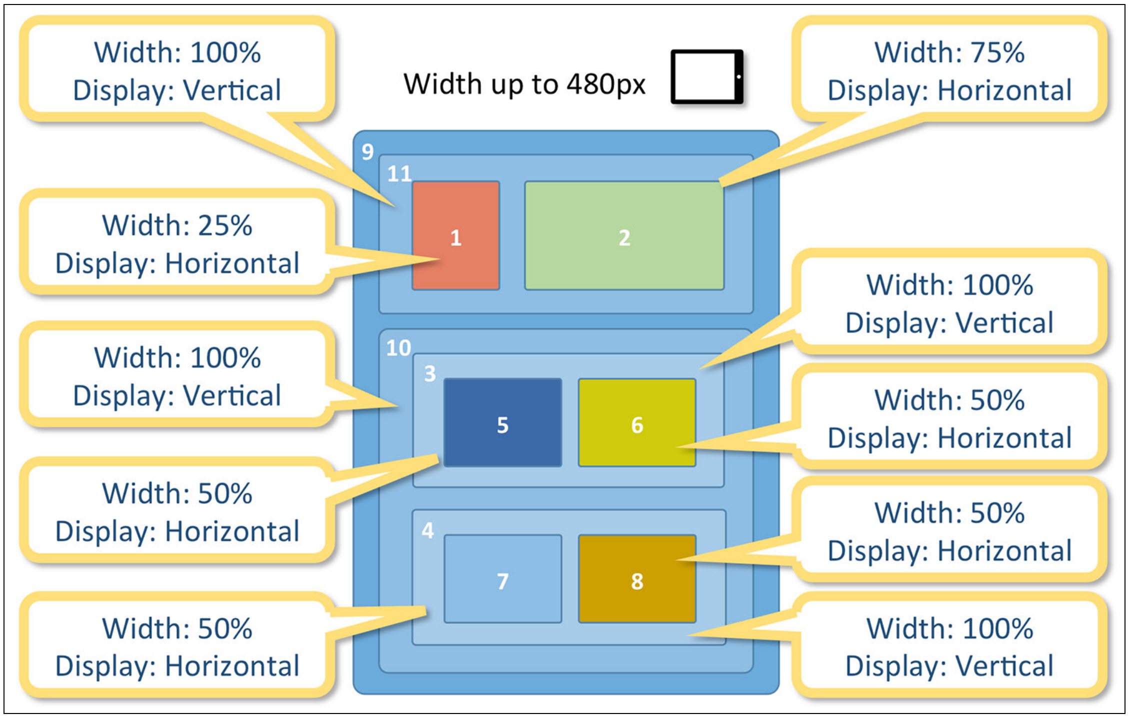


Figure 7-78 Planning for medium screen form factor

Figure 7-79 on page 326 shows the planning for a small screen form factor.

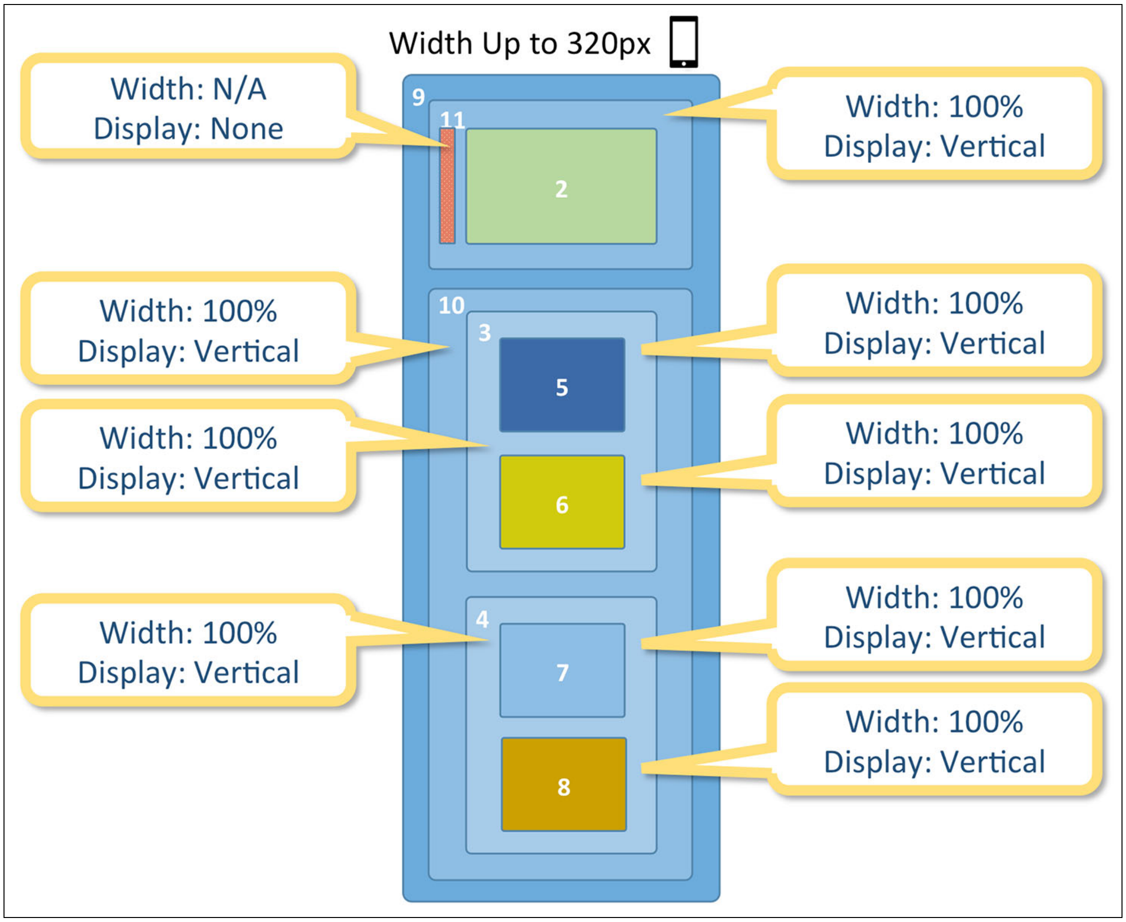


Figure 7-79 Planning for small screen form factor

When the UI planning is complete to a level of detail that is similar to what is shown in Figure 7-77 on page 324, Figure 7-78 on page 325, and Figure 7-79 the responsive implementation becomes very straightforward by using the configuration options of the Responsive Section, as shown in Figure 7-75 on page 322.

Responsive policies

Specifying responsive behavior in Responsive Sections at the Coach View level assumes that the particular reusable View is always going to be used, positioned, or grouped with other controls in a similar way, in whatever context it is used.

This assumption is not necessarily correct, especially when implementing more complex Coaches containing many reusable Coach Views arranged in different ways. In other words, the resizing, displaying, and additional styling of a Coach View on one Coach Page might need to work very differently on another Coach Page, hence the need to specify responsive behavior at the Coach Page level, independently of Responsive Section specifications at the Coach View level.

A separate EmeriConVIEWS Responsive Policy can be added to a Coach Page and configured for each wanted form factor, as shown in Figure 7-80.

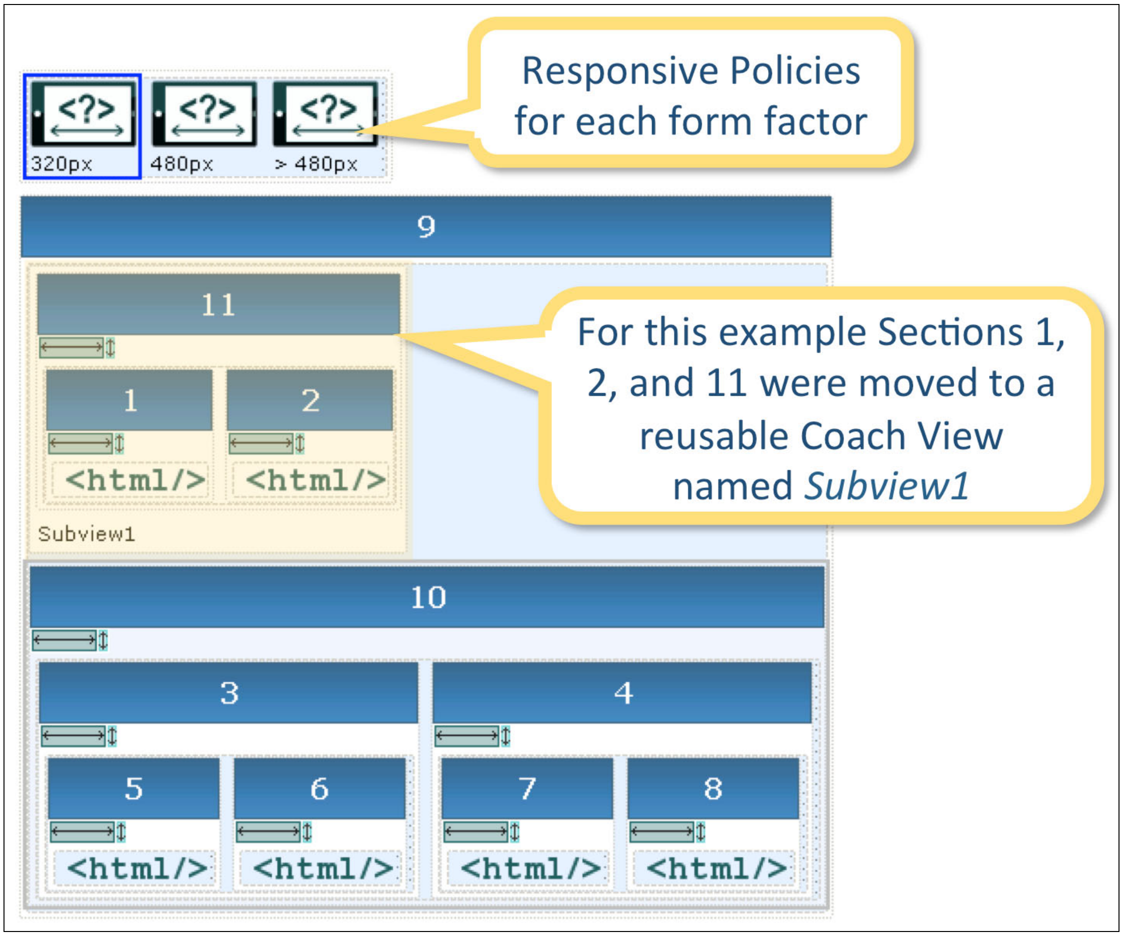
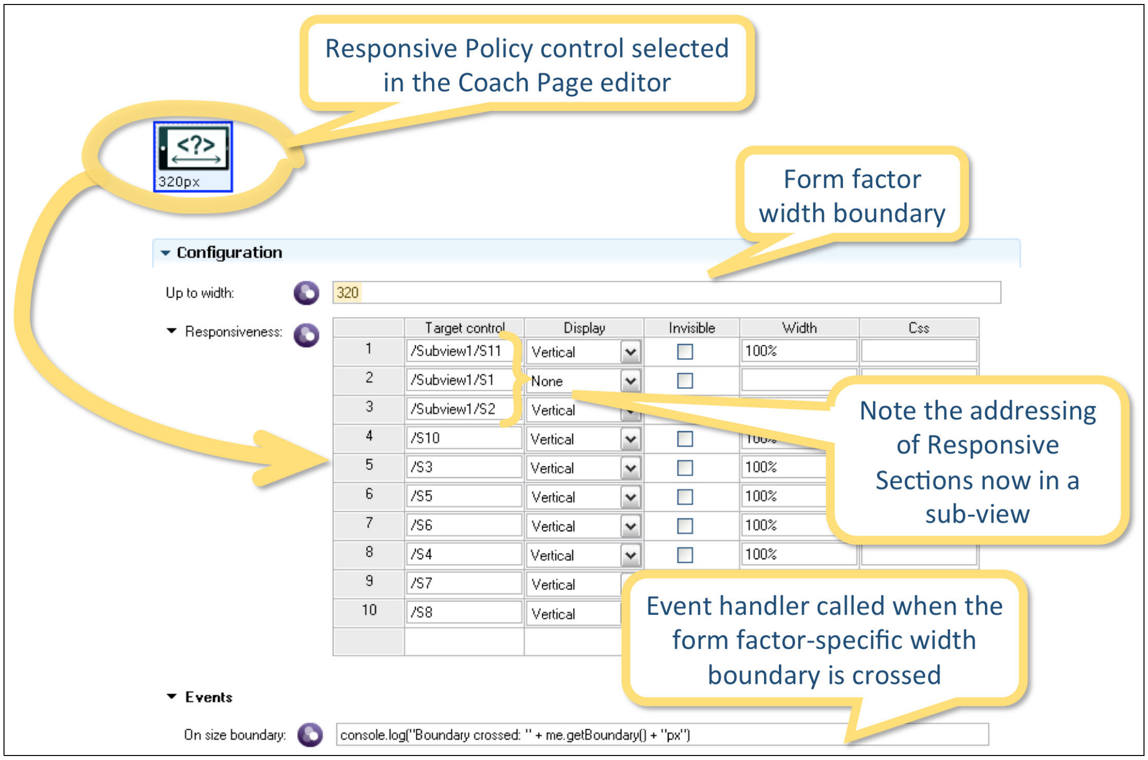

Responsive Policies for each form factor
320px
480px
> 480px
9
11
1
2
<html/>
Subview1
For this example Sections 1, 2, and 11 were moved to a reusable Coach View named *Subview1*
10
3
4
5
6
7
8

Figure 7-80 Coach Page-level Responsive Policies

The Coach Page author then selects a Responsive Policy control and adds wanted behaviors for controls based on the form factor that is specified on the Responsive Policy, as shown in Figure 7-81.


Responsive Policy control selected in the Coach Page editor
<?>
320px
Form factor width boundary
Note the addressing of Responsive Sections now in a sub-view
Event handler called when the form factor-specific width boundary is crossed

Configuration

Up to width: 320

Responsiveness:

	Target control	Display	Invisible	Width	Css
1	/Subview1/S11	Vertical	☐	100%	
2	/Subview1/S1	None	☐		
3	/Subview1/S2	Vertical			
4	/S10	Vertical	☐	100%	
5	/S3	Vertical	☐	100%	
6	/S5	Vertical	☐	100%	
7	/S6	Vertical	☐	100%	
8	/S4	Vertical	☐	100%	
9	/S7	Vertical			
10	/S8	Vertical			

Events

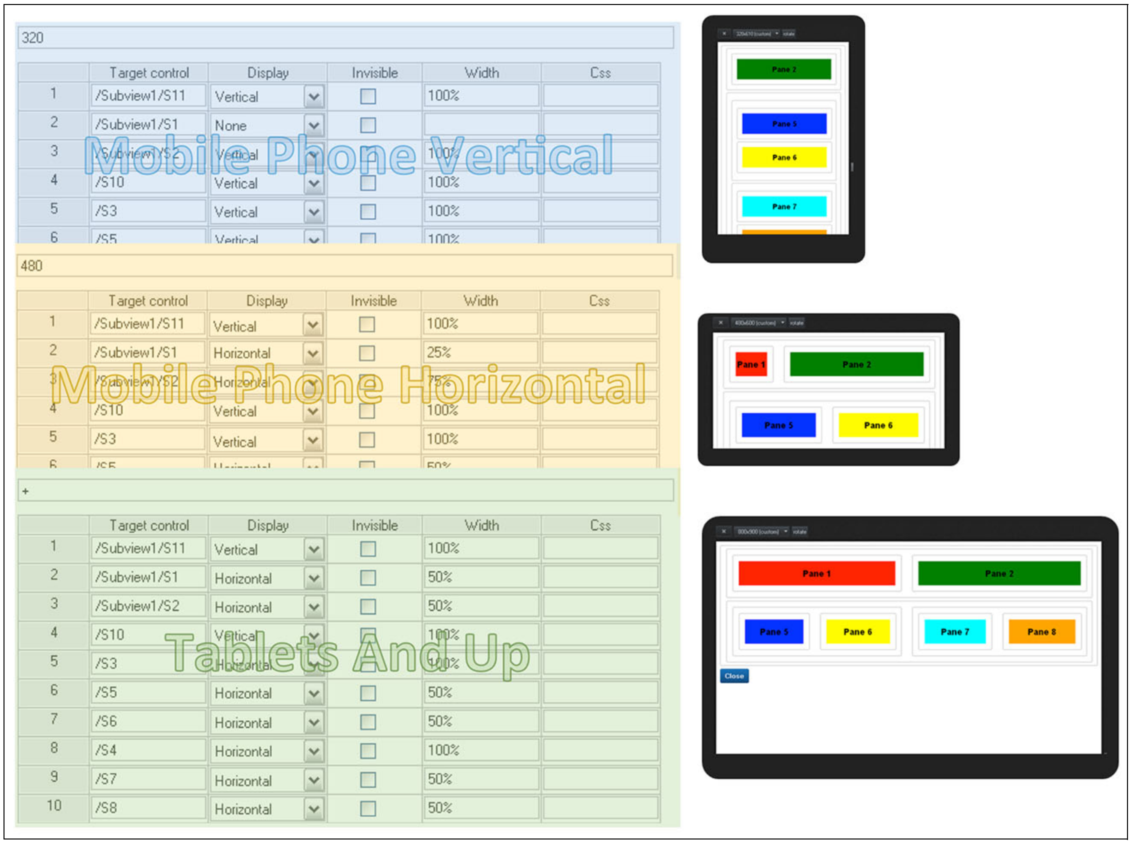
On size boundary:

```
console.log("Boundary crossed: " + me.getBoundary() + "px")
```

Figure 7-81 Responsive Policy configuration for a specific width

Responsive Policies provide a convenient way to group the behavior of all responsive elements on a Coach Page (and in the Coach View potentially contained by the page).

Figure 7-82 on page 329 shows each Responsive Policy configuration against its matching device or form factor.



320

	Target control	Display	Invisible	Width	Css
1	/Subview1/S11	Vertical	☐	100%	
2	/Subview1/S1	None	☐		
3	/Subview1/S2	Vertical	☐	100%	
4	/S10	Vertical	☐	100%	
5	/S3	Vertical	☐	100%	
6	/S5	Vertical	☐	100%	

480

	Target control	Display	Invisible	Width	Css
1	/Subview1/S11	Vertical	☐	100%	
2	/Subview1/S1	Horizontal	☐	25%	
3	/Subview1/S2	Horizontal	☐	75%	
4	/S10	Vertical	☐	100%	
5	/S3	Vertical	☐	100%	
6	/S5	Horizontal	☐	50%	

+

	Target control	Display	Invisible	Width	Css
1	/Subview1/S11	Vertical	☐	100%	
2	/Subview1/S1	Horizontal	☐	50%	
3	/Subview1/S2	Horizontal	☐	50%	
4	/S10	Vertical	☐	100%	
5	/S3	Horizontal	☐	100%	
6	/S5	Horizontal	☐	50%	
7	/S6	Horizontal	☐	50%	
8	/S4	Horizontal	☐	100%	
9	/S7	Horizontal	☐	50%	
10	/S8	Horizontal	☐	50%	

Figure 7-82 Configuration summary for three Responsive Policies

By design, Responsive Policy behaviors always override configured Responsive Section behaviors. This means that even if Responsive Section S1 contains a configured *Display: None* behavior for width 320px for example, a configuration entry for S1 in a 320px Responsive Policy specifying *Display: Vertical* always wins.

Lastly, note that Responsive Policies are meant to be used *in lieu of* Responsive Section-based configuration when Coach Views are used on a Coach Page. Because Responsive Section-based configuration may be deemed more convenient (because it is performed “in-place”), EmeriConVIEWS supports both modes of Responsive Policy and Responsive Section-based configurations.

Good practices

Following are some good practices:

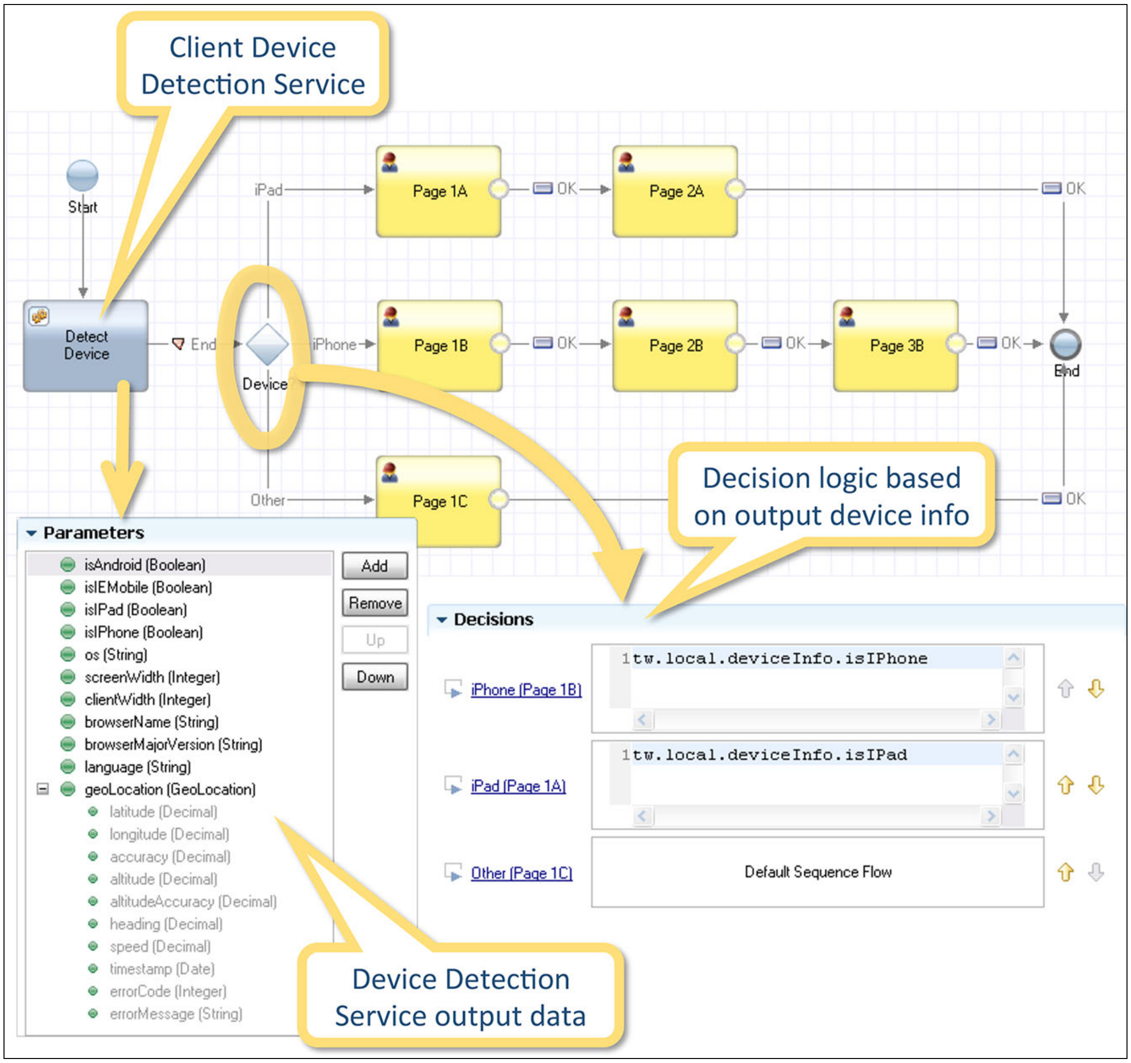
- Specifying responsive behavior directly on Responsive Sections is most appropriate (and arguably more convenient) for sections that are at the Coach Page level
- When using Responsive Sections in a Coach View, it is usually best to express responsive behavior through Responsive Policy controls at the Coach Page level instead of in the configuration of the Responsive Section
- Responsive Policies control should be added to a Coach Page, not a Coach View. In the rare case that responsive behavior should be specified at a Coach View level, the Responsiveness configuration options of a Responsive Section should be used instead
- Time-saving tip: To avoid reentering data for each new Responsive Policy (for a new form factor), just copy and paste an existing configured Responsive Policy control in the same Process Application or Toolkit. Doing so also replicates the configuration into the newly pasted control. The author then just needs to change the styling in the already existing entries in the configuration
- Always plan the responsive behavior of a Coach Page before starting to author it (as shown in Figure 7-77 on page 324, Figure 7-78 on page 325, and Figure 7-79 on page 326). The configuration capabilities of the Responsive Section and Responsive Policy controls are specifically designed to simplify expressing responsive behavior while following the planned approach.

Device detection

The last Responsive Design feature of EmeriConVIEWS is the Device Detection service.

In certain (albeit more rare) cases, designing a responsive Human Service entails not only rearranging content or changing its style, but also (or instead) presenting a significantly different UI that is more adapted to the device on which it runs. In such cases, what should be displayed as one Coach Page on one device may instead (for example) need to be displayed as several Coach Pages on another.

The service Figure 7-83 on page 331 provides such an example.



Parameters

- isAndroid (Boolean)
- isIEMobile (Boolean)
- isIPad (Boolean)
- isIPhone (Boolean)
- os (String)
- screenWidth (Integer)
- clientWidth (Integer)
- browserName (String)
- browserMajorVersion (String)
- language (String)
- geoLocation (GeoLocation)
  - latitude (Decimal)
  - longitude (Decimal)
  - accuracy (Decimal)
  - altitude (Decimal)
  - altitudeAccuracy (Decimal)
  - heading (Decimal)
  - speed (Decimal)
  - timestamp (Date)
  - errorCode (Integer)
  - errorMessage (String)

Decisions

iPhone (Page 1B)	`tw.local.deviceInfo.isIPhone`
iPad (Page 1A)	`tw.local.deviceInfo.isIPad`
Other (Page 1C)	Default Sequence Flow

Figure 7-83 Using the Device Detection Service in a Human Service

7.4 Technical considerations specifically addressed

EmeriConVIEWS addresses specific areas of usability, maintenance, and general capability by extending the IBM BPM Coach NG framework. The extension of the framework uses published and supported mechanisms to enhance the following areas:

- **Addressing/referencing Coach Views**: Referencing Coach NG Views or controls entails a level of technical complexity (using context.parentView and context.getSubView) that is significantly simplified by the EmeriConVIEWS addressing scheme. The simplified addressing scheme is then used in consistent ways through all EmeriConVIEWS aspects to provide the author a more intuitive experience for a fundamental aspect of Coach UIs
- **Focusing on business instead of technical aspects**: The Coach NG programming model requires a strong programming background with at least additional knowledge of the Dojo library, the HTML DOM, HTML-specific JavaScript, and sometimes Ajax. EmeriConVIEWS normalizes the experience with a higher-level and consistent approach and programming model that is more accessible to non-UI development experts, and arguably more natural to UI developers. The mix of simplified addressing, event handling, and control methods allow sophisticated client-side interactions to be expressed relatively simply, and focusing business requirements instead of technical questions
- **Triggering logic and Ajax services through events**: Coach NG does not allow simple actions such as selecting an item from a select control, tabbing off a text field, or selecting a check box, to immediately trigger logic or an Ajax service. At least not without developing some sort of technical glue. EmeriConVIEWS provides very accessible and intuitive mechanisms to achieve such interactions without necessitating the added technical development effort to support them
- **Resilience of logic to aesthetic UI changes**: Coach NG Sections such as Horizontal, Vertical, and Tabs sections are often (rightly) used to make a UI more appealing and more visually organized. Moving Coach Views in and out of sections however can cause code breakage in Coach Views because the current client-side control addressing scheme considers aesthetic elements of a UI part of the view tree. Thus, moving a Coach View out of or in to a section effectively changes its address. The EmeriConVIEWS addressing scheme provides a more resilient approach to view references that is not impacted by changes that are merely visual
- **Preserving Coach View encapsulation principles**: It is often tempting and sometimes necessary to access the inside details of views (for example access to the contained Dojo, jQuery, Bootstrap component of the view or its HTML DOM) to implement certain behaviors. Having to work with the insides

of a view exposes the solution to breakage from internal view changes. To prevent these types of leakages, EmeriConVIEWS treats views or controls as fully encapsulated components with events, methods, and properties, containing enough built-in capabilities without needing to access internal and more technical details

- **Providing CSS Styling at the Coach Page level**: With Coach NG, CSS can only be specified at the Coach View level. More often than not however, the Coach Page should dictate the broad styling that should be applied to some, most, or all of the Coach Views that it contains. While it is possible to use a workaround with Coach NG, EmeriConVIEWS directly supports specifying CSS at the Coach Page level (with the Style control), and provides the ability to broadly switch CSS themes dynamically. Coupled with the Device Sensor control, CSS themes can be switched dynamically based on the device on which the Coach runs
- **Limiting artifact proliferation**: Certain technical aspects of Coach NG induce the proliferation of artifacts such as Business Object definitions and additional views (because accessing the full functionality of Coach NG is challenging), mostly for technical reasons. This leads to BPM solutions that are not as easy to understand, maintain, or change. By contrast, EmeriConVIEWS helps limit the creation of UI-related artifacts to those that have business meaning. The resulting solution is usually significantly cleaner than for a base Coach NG implementation with comparable functionality
- **Full function both at Coach Page and View level**: Authoring capabilities with Coach NG are very limited at the Coach Page level. This usually means that a Coach View needs to be created for each Coach Page in order to create very functional UIs. This tends to proliferate artificial solution components (as mentioned above). EmeriConVIEWS provides a virtually identical authoring experience whether at the Coach Page or the Coach View level. This means that reusable Coach Views are only created when reuse is justified
- **Using controls without data bindings**: It is often very convenient to use controls that are not bound to data on a Coach Page or Coach View. Coach NG controls are not easily usable without data bindings whereas EmeriConVIEWS controls work equally well with and without bound data. This allows the author to use business-relevant-only Business Objects without having to create artificial Business Objects to hold extra data that exists only to back controls that have no real business data meaning
- **Optimizing control content and UI behavior for mobile devices**: The logic of EmeriConVIEWS controls is fully contained in separate minified script files for improved caching and execution time. The ability to create sophisticated logic that is completely contained on the client side (and therefore limits server interactions to the absolute minimum required) further protects user experience from potentially unpredictable mobile device connectivity.

EmeriConVIEWS also provides specific mobile features to support displaying coaches on completely different devices where not only the style (for example, layout of UI elements) is different, but the behavior and overall capability of the UI is different also

7.5 Mixing EmeriConVIEWS with other toolkits

EmeriConVIEWS can be used in conjunction with, or as an extension to, other Coach UI toolkits. The following sections address how to EmeriConVIEWS-enable regular Coach NG Coach Views and how other UI toolkits can interact EmeriConVIEWS.

7.5.1 Making EmeriConVIEWS views out of regular Coach NG views

Any non-compound Coach View (for example a special text control, a tree control, a chart) can be made an EmeriConVIEWS control with fairly minimal effort. The level of EmeriConVIEWS enablement depends on the expected usage for the Coach View but typically involves:

- Enabling the control with the EmeriConVIEWS simplified addressing scheme
- Adding meaningful methods to the control (optional but often useful)
- Adding event support (optional but often useful)
- Adding formula support (optional)

Figure 7-84 on page 335 augments a Gauge Coach View from the BPM Samples Wiki named *Horizontal Linear Gauge Green CV* (available at the following BPM Samples Wiki link) with all four aspects above:

`http://bpmwiki.blueworkslive.com/display/samples/CV+-+Gauges`

The Gauge Coach View starts out.

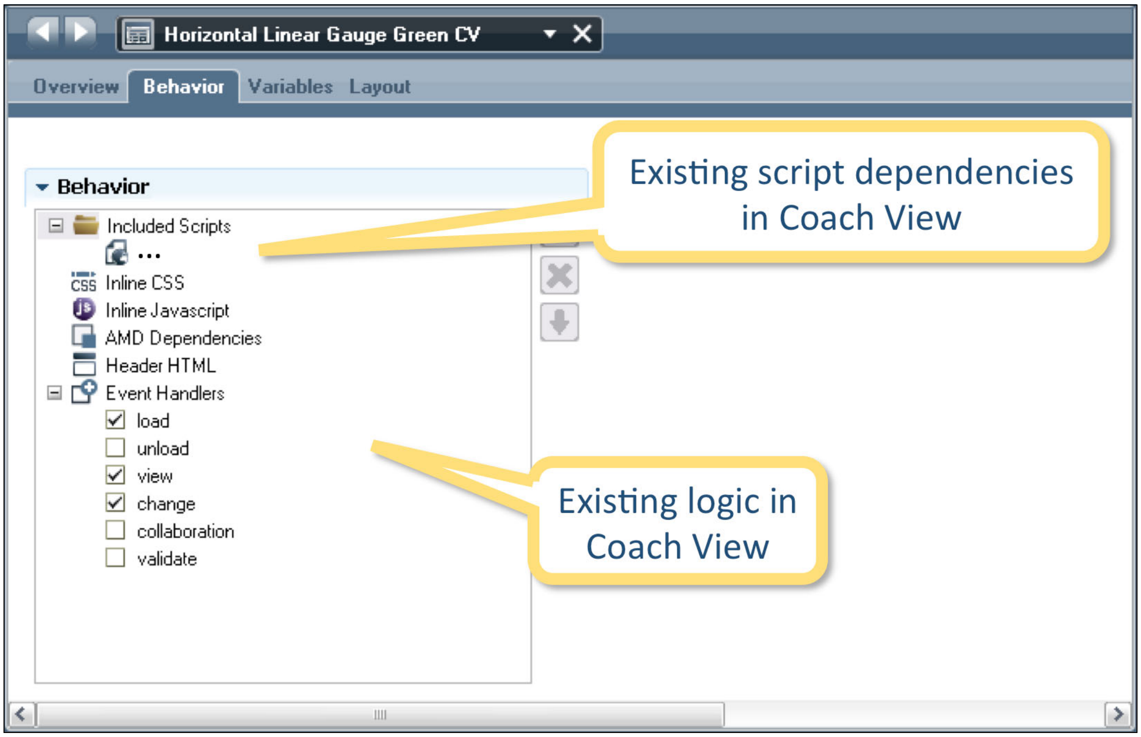


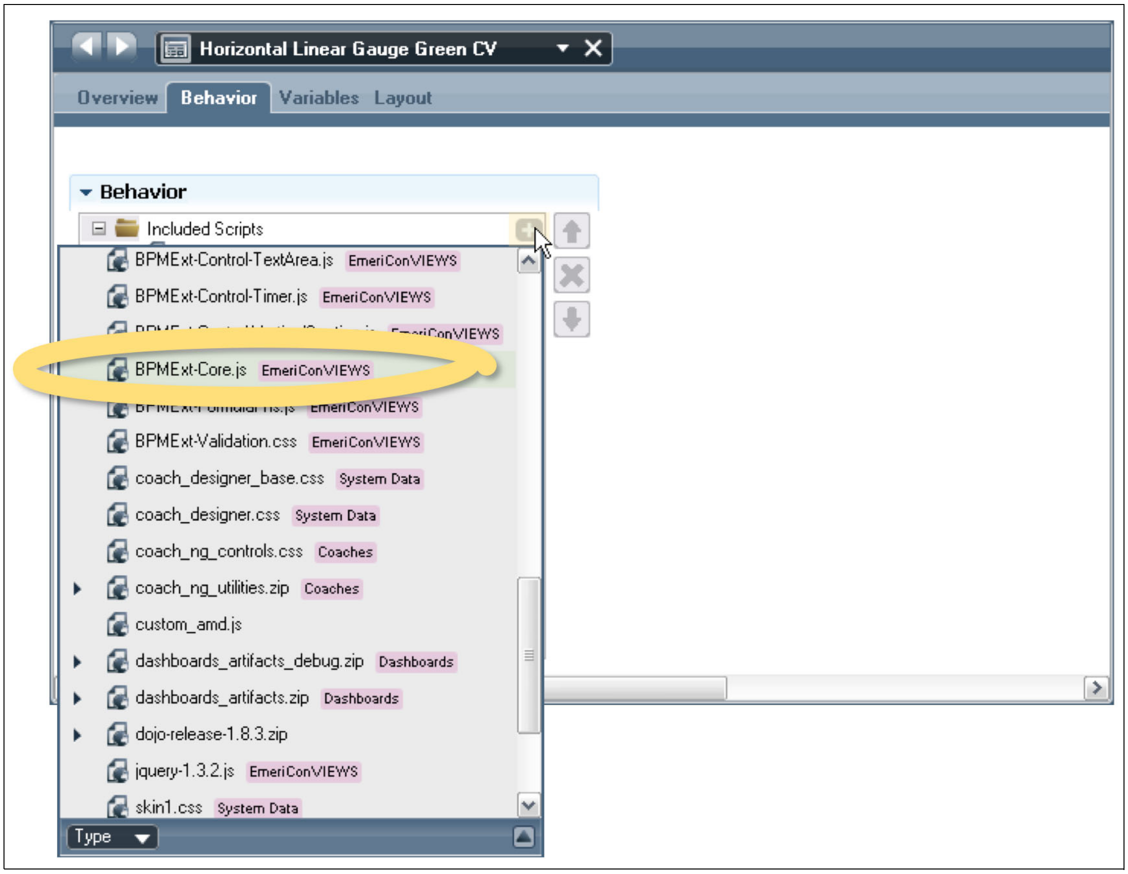
Figure 7-84 Initial Coach View – Before augmenting with EmeriConVIEWS

Augmenting a Coach View with EmeriConVIEWS simplified addressing

Enabling the control with the EmeriConVIEWS simplified addressing scheme entails:

- Adding the EmeriConVIEWS dependency to the toolkit or process application
- Adding the EmeriConVIEWS core JS file to the control's included scripts
- Adding registration and deregistration function calls to the control

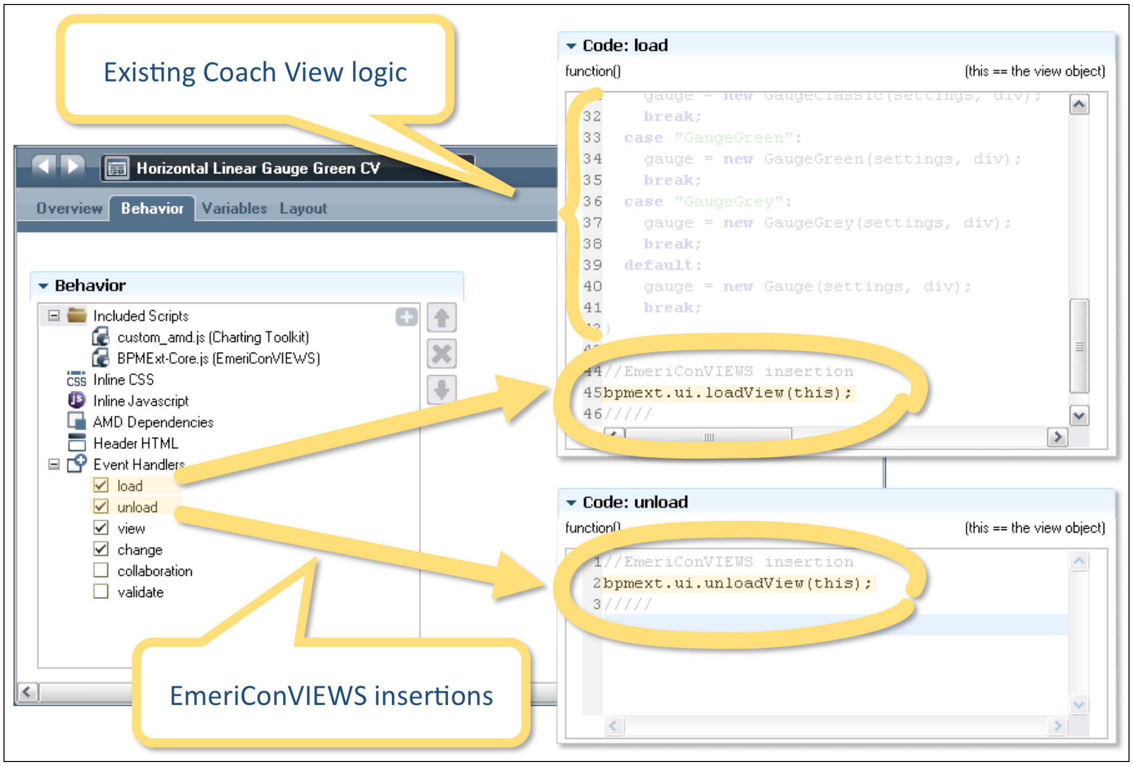
The last two prerequisites are illustrated in Figure 7-85 on page 336. First, the `BPMExt-Core.js` file in added to the included scripts for the Coach View.



BPMExt-Control-TextArea.js EmeriConVIEWS
BPMExt-Control-Timer.js EmeriConVIEWS
BPMExt-Core.js EmeriConVIEWS
BPMExt-Validation.css EmeriConVIEWS
coach_designer_base.css System Data
coach_designer.css System Data
coach_ng_controls.css Coaches
coach_ng_utilities.zip Coaches
custom_amd.js
dashboards_artifacts_debug.zip Dashboards
dashboards_artifacts.zip Dashboards
dojo-release-1.8.3.zip
jquery-1.3.2.js EmeriConVIEWS
skin1.css System Data

Figure 7-85 Adding the Core EmeriConVIEWS script to Included Scripts of Coach View

Then the two registration and deregistration function calls are added to the load and unload Coach View events respectively, as shown in Figure 7-86 on page 337.



```
        gauge = new GaugeClassic(settings, div);
        break;
    case "GaugeGreen":
        gauge = new GaugeGreen(settings, div);
        break;
    case "GaugeGrey":
        gauge = new GaugeGrey(settings, div);
        break;
    default:
        gauge = new Gauge(settings, div);
        break;
}

//EmeriConVIEWS insertion
bpmext.ui.loadView(this);
/////
```

```
//EmeriConVIEWS insertion
bpmext.ui.unloadView(this);
/////
```

Figure 7-86 Adding EmeriConVIEWS registration logic to Coach View

`bpmext.ui.loadView()` adds the Coach View to the EmeriConVIEWS simplified addressing scheme. It also augments the Coach View with methods common to all controls such as:

- `isVisible()/setVisible()`, `isEnabled()/setEnabled()` programmatically get and set visibility or enabled status
- `isValid()/setValid()` get and set client-side validation status

Conversely, `bpmext.ui.unloadView()` removes the Coach View from the EmeriConVIEWS addressing scheme.

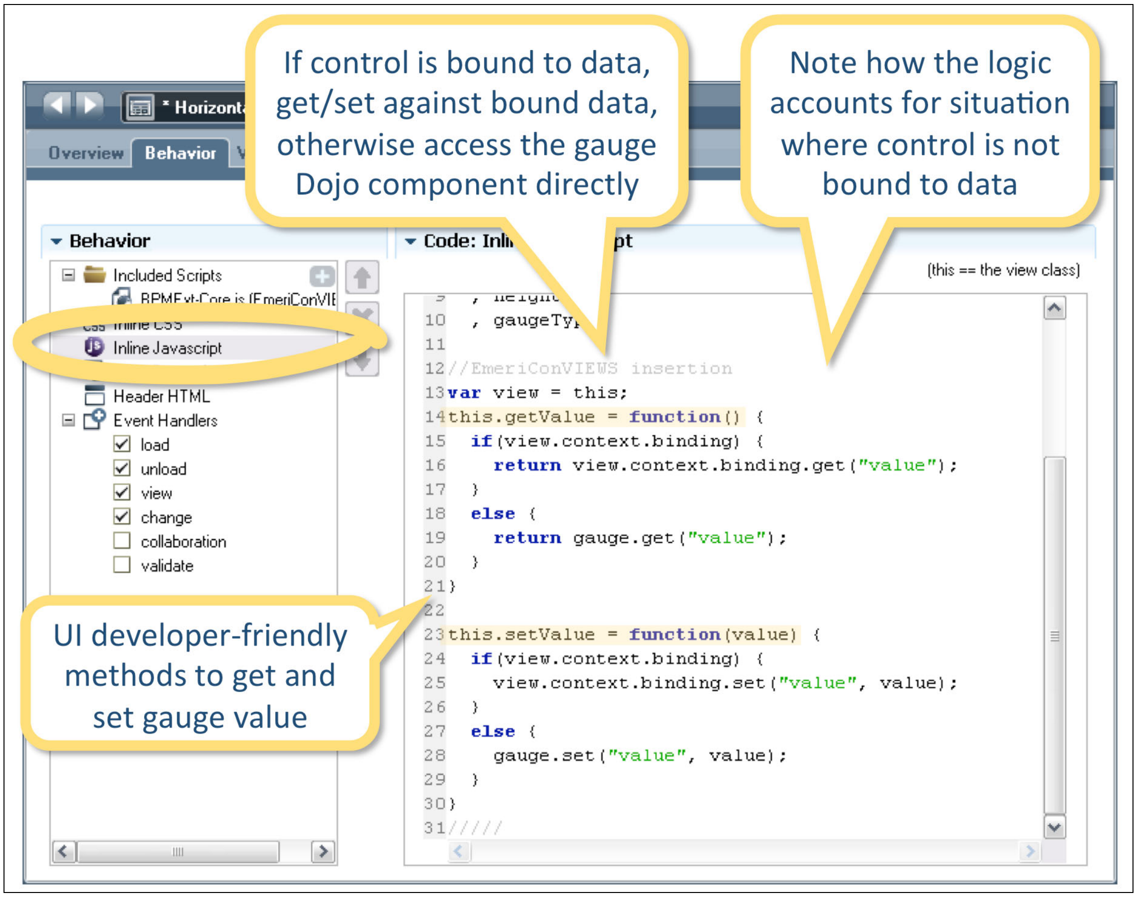
Note that for Coach Views that act as containers (such as sections), the `bpmext.ui.loadContainer()/bpmext.ui.unloadContainer()` functions should be used instead.

The Gauge control is now ready to be addressed through the simplified EmeriConVIEWS addressing scheme.

Adding methods to the Coach View

The next step is to add meaningful functions to the Coach View to make it act more like a control with useful methods.

For example, a simplified way to get and set the value of the gauge could be provided, as shown in Figure 7-87.



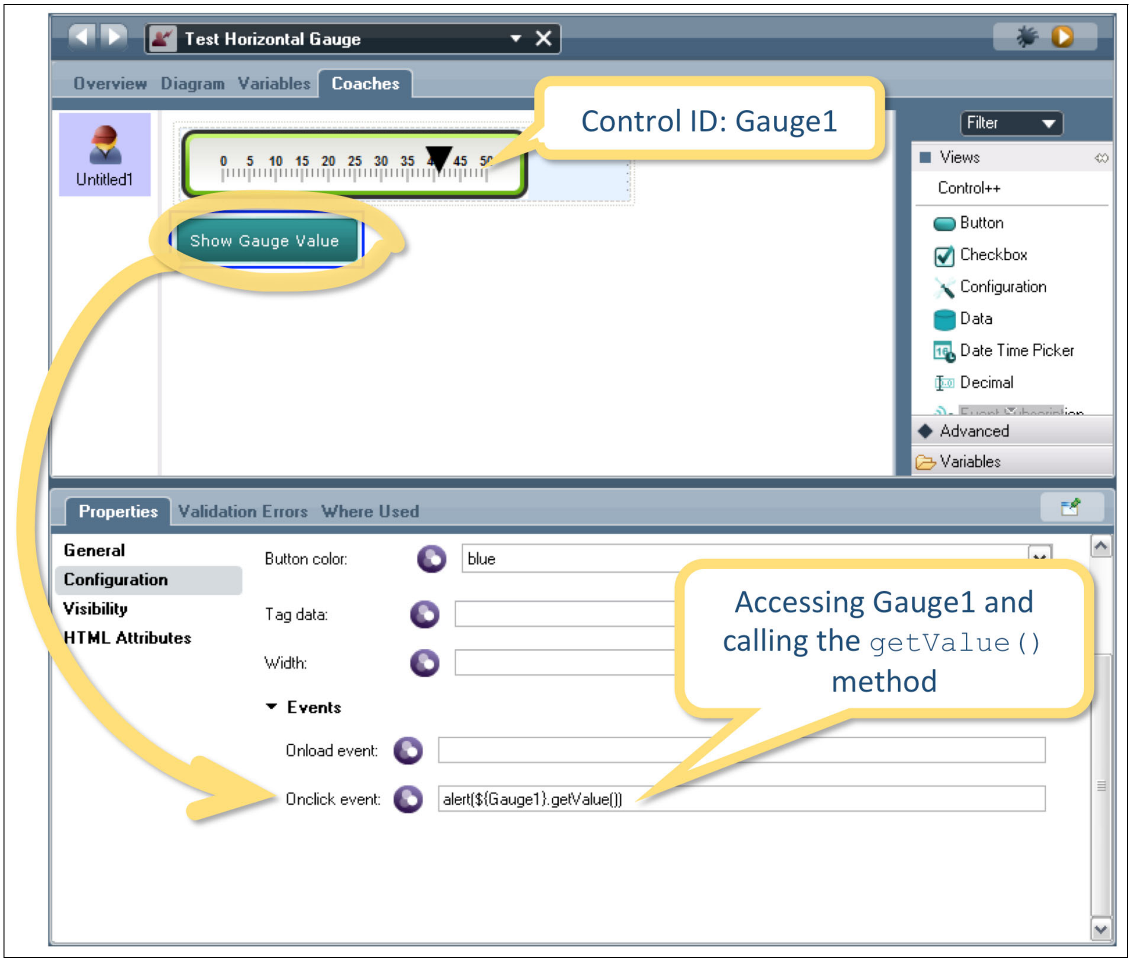
If control is bound to data, get/set against bound data, otherwise access the gauge Dojo component directly
Note how the logic accounts for situation where control is not bound to data
UI developer-friendly methods to get and set gauge value
* Horizont
Overview
Behavior
Behavior
Included Scripts
Inline Javascript
Header HTML
Event Handlers
load
unload
view
change
collaboration
validate
Code: Inli
pt
(this == the view class)

```
  , gaugeTy

//EmeriConVIEWS insertion
var view = this;
this.getValue = function() {
  if(view.context.binding) {
    return view.context.binding.get("value");
  }
  else {
    return gauge.get("value");
  }
}

this.setValue = function(value) {
  if(view.context.binding) {
    view.context.binding.set("value", value);
  }
  else {
    gauge.set("value", value);
  }
}
/////
```

Figure 7-87 Adding UI developer-friendly methods: Coach View

Figure 7-87 is only an example of possible methods. Other control methods could be created to dynamically alter the aspect of the control, manage validation, change the range of allowed values, and so on.

With the added get/setValue methods, the following example logic can now be added. This is demonstrated in Figure 7-88 on page 339.



Button color: blue
Tag data:
Width:
Onload event:
Onclick event: alert(${Gauge1}.getValue())

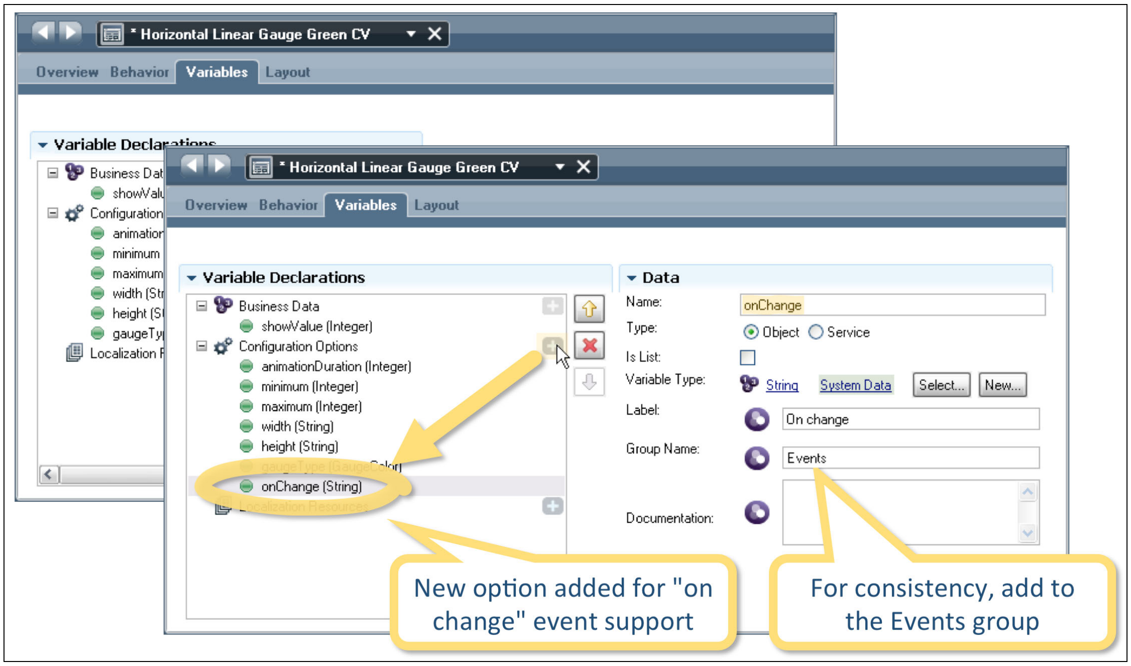
Figure 7-88 Showing Gauge Coach View value on button click

Adding event support to the Coach View

Useful controls also need easily exploitable event support, for example, to react to user interactions. The following example shows how to add “on change” event support for the Gauge control. Creating support for an event such as “on change”, the developer must decide what will cause the event to be fired and if the event firing will occur only if the control is bound to data or in all cases. In this example, “on change” will be fired whether the control is bound to data or not. The event will be fired if one of the following actions occurs:

- A user moves the slider
- The data bound to the control (if the control is bound) is updated
- The `setValue()` method is called

The first step is to add a configuration option that will either contain the inline logic to be invoked when the event is fired, or the name of a function to be called as shown in Figure 7-89 (see "Control event-handling" on page 241).

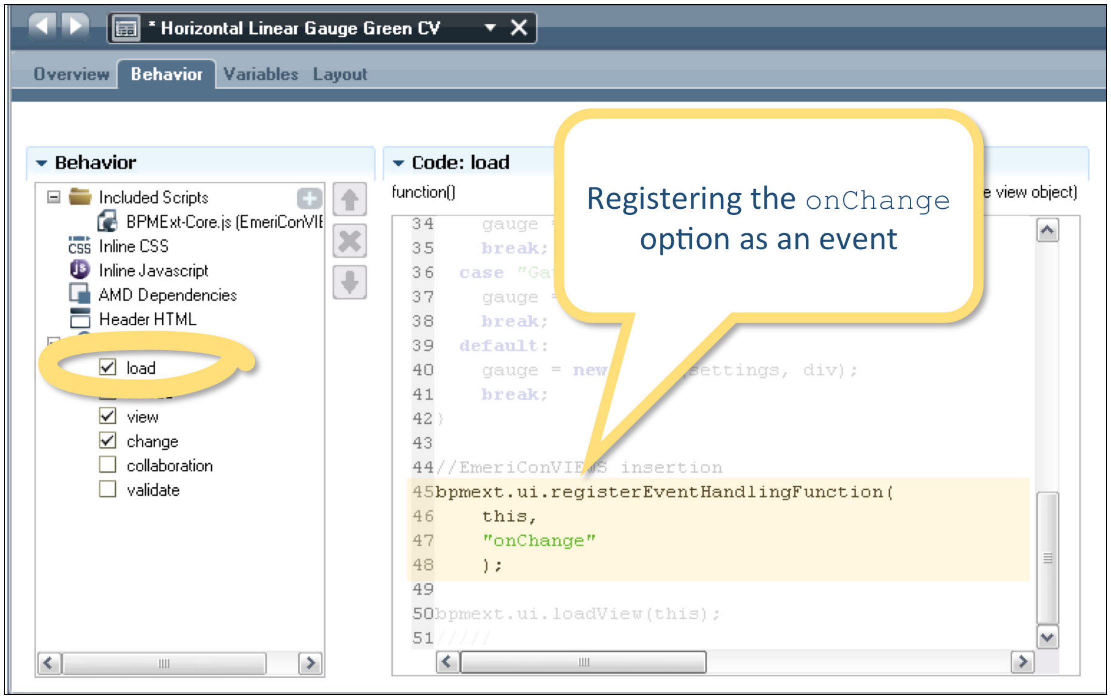


Name: onChange
Type: [✓ Object] Service
Is List: ☐
Variable Type: String System Data
Label: On change
Group Name: Events
Documentation:

Figure 7-89 Adding a Gauge Coach View configuration option for "on change" event support

The name of the option is onChange only because it is more meaningful than a non-descript name. The name of the option is used further in the registration and event firing mechanism as shown in Figure 7-90 on page 341.

The next step is to register the event so the event can be fired later.



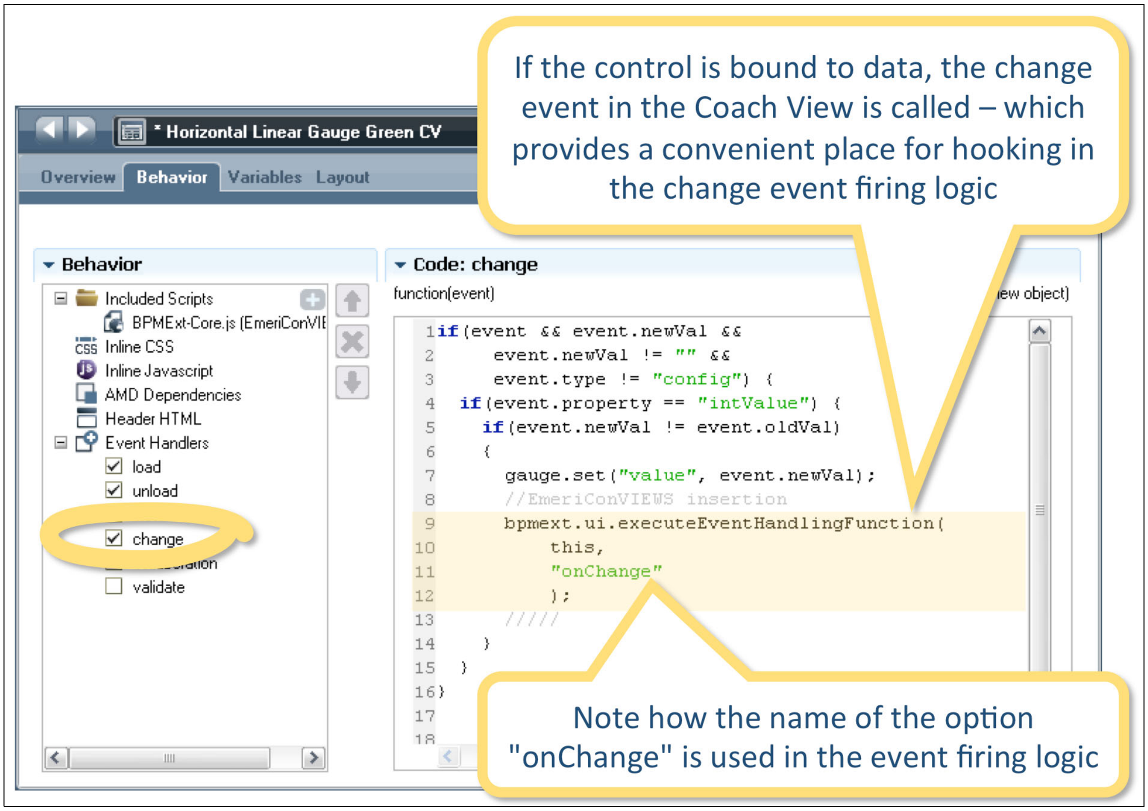
```
    gauge =
    break;
  case "Ga
    gauge =
    break;
  default:
    gauge = new         settings, div);
    break;
}

//EmeriConVIEWS insertion
bpmext.ui.registerEventHandlingFunction(
    this,
    "onChange"
    );

bpmext.ui.loadView(this);
/////
```

Figure 7-90 Registering the Gauge Coach View onChange option as an event

At the stage of event registration, the inline logic specified for the event is preprocessed so that any control reference in the form `${control-id}` is translated into a real JavaScript reference expression and the inline logic is turned into a real JavaScript function.

The last step is to insert the logic that fires the on change event in appropriate locations in the Coach View code. The work required for this step can vary significantly from one view to the next and depends upon whether the existing Coach View provides straightforward hooks for event handling or not.

Handling *on change* events for Coach Views that are bound to data is straightforward with Coach NG Views because a change event handler already exists, as shown in Figure 7-91 on page 342.

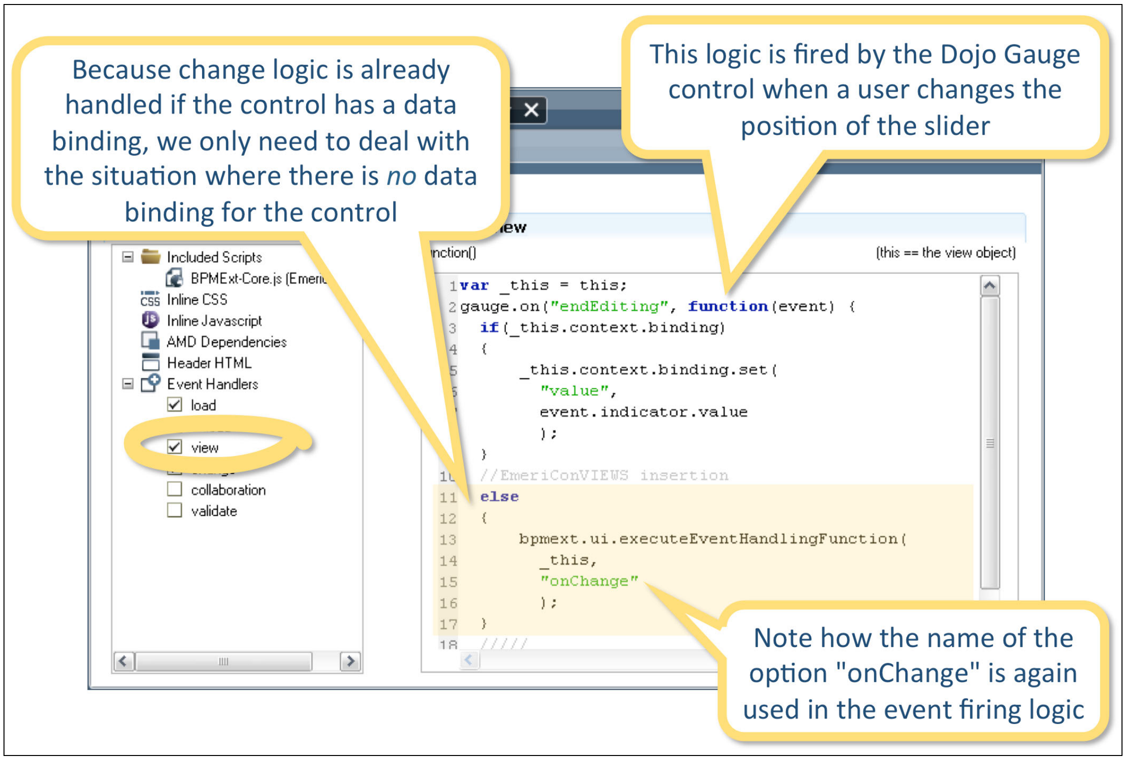

* Horizontal Linear Gauge Green CV
Overview
Behavior
Variables
Layout
Behavior
Included Scripts
BPMExt-Core.js (EmeriConVIE
Inline CSS
Inline Javascript
AMD Dependencies
Header HTML
Event Handlers
load
unload
change
validate
Code: change
function(event)
ew object)
If the control is bound to data, the change event in the Coach View is called – which provides a convenient place for hooking in the change event firing logic
Note how the name of the option "onChange" is used in the event firing logic

```
if(event && event.newVal &&
    event.newVal != "" &&
    event.type != "config") {
  if(event.property == "intValue") {
    if(event.newVal != event.oldVal)
    {
      gauge.set("value", event.newVal);
      //EmeriConVIEWS insertion
      bpmext.ui.executeEventHandlingFunction(
          this,
          "onChange"
      );
      /////
    }
  }
}
```

Figure 7-91 Adding "on change" event firing logic to Gauge Coach View through existing change event handler

The approach in Figure 7-91 works well for Gauge that has a data binding; however, a different approach must be used to detect changes in the Coach View if it has no data binding.

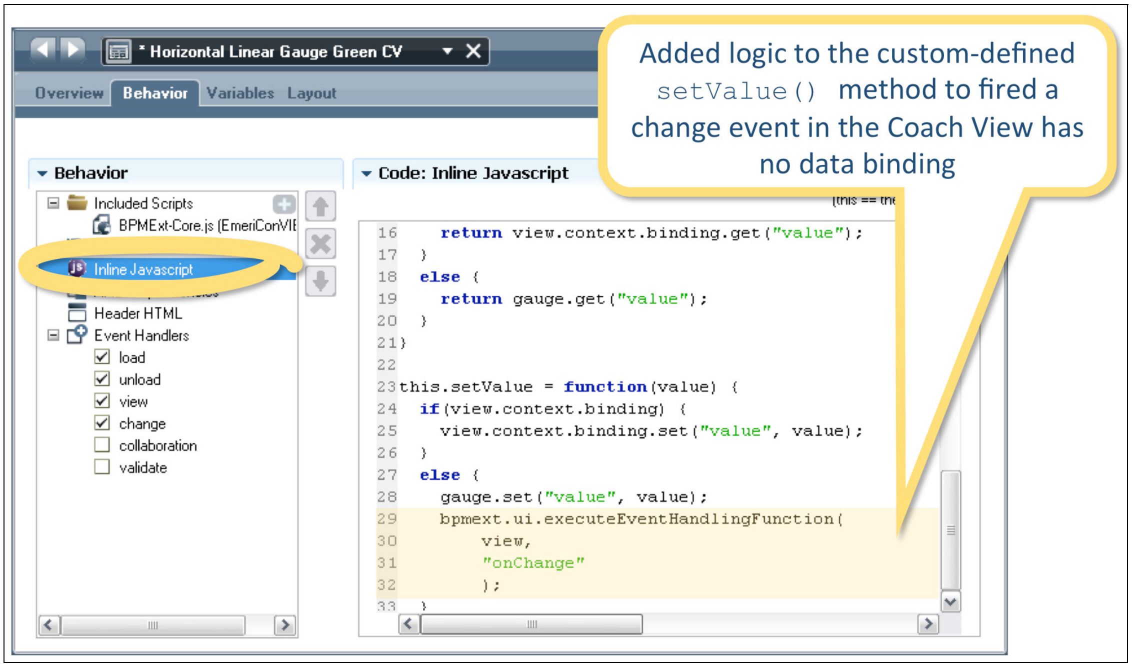
If the Coach View is used without a data binding, the Coach View change event will not be fired. In this example, the Gauge Coach View uses the Horizontal Gauge Dojo component, which provides change detection hook with the *endEditing* Dojo control event, as shown in Figure 7-92 on page 343.


Because change logic is already handled if the control has a data binding, we only need to deal with the situation where there is *no* data binding for the control
This logic is fired by the Dojo Gauge control when a user changes the position of the slider
Note how the name of the option "onChange" is again used in the event firing logic
Included Scripts
BPMExt-Core.js (Emeri
Inline CSS
Inline Javascript
AMD Dependencies
Header HTML
Event Handlers
load
view
collaboration
validate
(this == the view object)

```
var _this = this;
gauge.on("endEditing", function(event) {
  if(_this.context.binding)
  {
      _this.context.binding.set(
        "value",
        event.indicator.value
      );
  }
  //EmeriConVIEWS insertion
  else
  {
      bpmext.ui.executeEventHandlingFunction(
        _this,
        "onChange"
      );
  }
  /////
```

Figure 7-92 Adding "on change" event firing logic to Gauge Coach View through Dojo gauge event handler

Lastly, setting the value of the slider through the setValue() method that was created previously only automatically fires a change event if the Gauge Coach View has a data binding. If not, specific logic can be added to the setValue() method to fire an event, as shown in Figure 7-93 on page 344.


* Horizontal Linear Gauge Green CV
Overview
Behavior
Variables
Layout
Behavior
Included Scripts
BPMExt-Core.js (EmeriConVIE
Inline Javascript
Header HTML
Event Handlers
load
unload
view
change
collaboration
validate
Code: Inline Javascript
Added logic to the custom-defined setValue() method to fired a change event in the Coach View has no data binding

```
    return view.context.binding.get("value");
  }
  else {
    return gauge.get("value");
  }
}

this.setValue = function(value) {
  if(view.context.binding) {
    view.context.binding.set("value", value);
  }
  else {
    gauge.set("value", value);
    bpmext.ui.executeEventHandlingFunction(
        view,
        "onChange"
        );
  }
```

Figure 7-93 Adding "on change" event firing logic to Gauge Coach View through setValue method

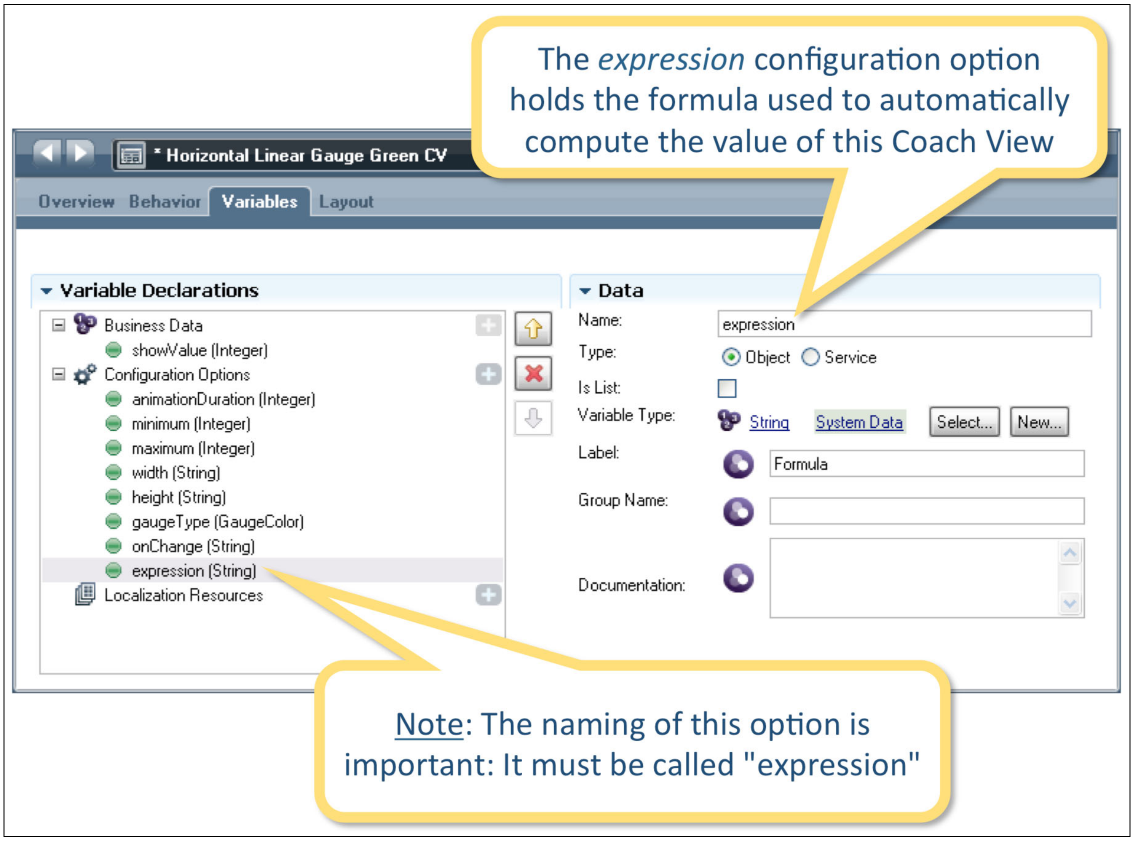
Adding formula support to the Coach View

It is convenient to enable certain controls with formula support. Formula support for a control can mean that:

- ► A control can be referenced in formulas
- ► A control can use a formula to set its value automatically

This section shows how to support both scenarios.

For a Coach View to use a formula to automatically set its value based on the computation of that formula, a special expression configuration option must be created to hold the formula for the view, as shown in Figure 7-94 on page 345.


The *expression* configuration option holds the formula used to automatically compute the value of this Coach View
* Horizontal Linear Gauge Green CV
Overview
Behavior
Variables
Layout
Variable Declarations
Data
Select...
New...
Note: The naming of this option is important: It must be called "expression"

Business Data
- showValue (Integer)

Configuration Options
- animationDuration (Integer)
- minimum (Integer)
- maximum (Integer)
- width (String)
- height (String)
- gaugeType (GaugeColor)
- onChange (String)
- expression (String)

Localization Resources

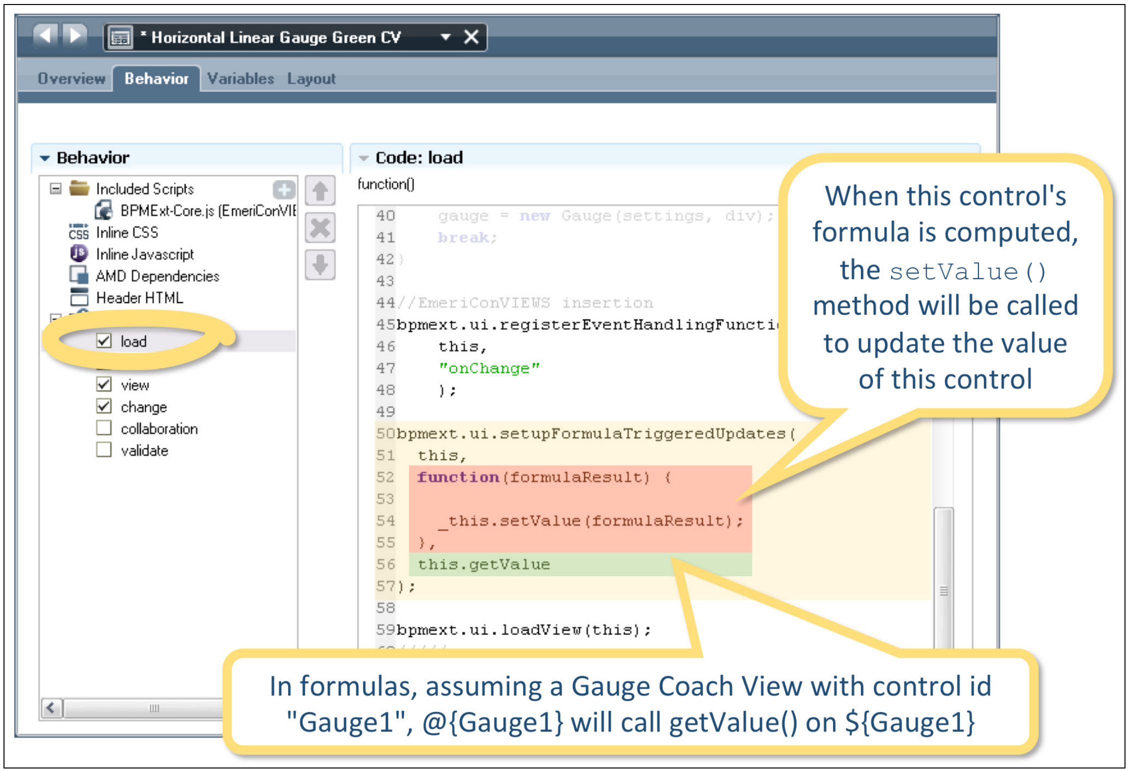
Name:	expression
Type:	[✓ Object] Service
Is List:	☐
Variable Type:	String System Data
Label:	Formula
Group Name:	
Documentation:	

Figure 7-94 Adding a Gauge Coach View configuration "expression" option for formula support

Similarly to event handling, a Coach View is registered for formula support through a special EmeriConVIEWS function:

```
bpmext.ui.setupFormulaTriggeredUpdates(
coachViewRef, //Reference to the Coach View
onFormulaComputedFn, //Callback to invoke on formula result
defaultValueFn //Function ref to map to @{ctrl-ref} in formulas)
```

Figure 7-95 on page 346 shows how setupFormulaTriggeredUpdates is used in the Gauge Coach View.


* Horizontal Linear Gauge Green CV
Overview
Behavior
Variables
Layout
Behavior
Included Scripts
BPMExt-Core.js (EmeriConVIE
Inline CSS
Inline Javascript
AMD Dependencies
Header HTML
load
view
change
collaboration
validate
Code: load
When this control's formula is computed, the setValue() method will be called to update the value of this control
In formulas, assuming a Gauge Coach View with control id "Gauge1", @{Gauge1} will call getValue() on ${Gauge1}

```
function()
    gauge = new Gauge(settings, div);
    break;
}

//EmeriConVIEWS insertion
bpmext.ui.registerEventHandlingFuncti
    this,
    "onChange"
    );

bpmext.ui.setupFormulaTriggeredUpdates(
  this,
  function(formulaResult) {

    _this.setValue(formulaResult);
  },
  this.getValue
);

bpmext.ui.loadView(this);
```

Figure 7-95   Setting up formula support for the Gauge Coach View

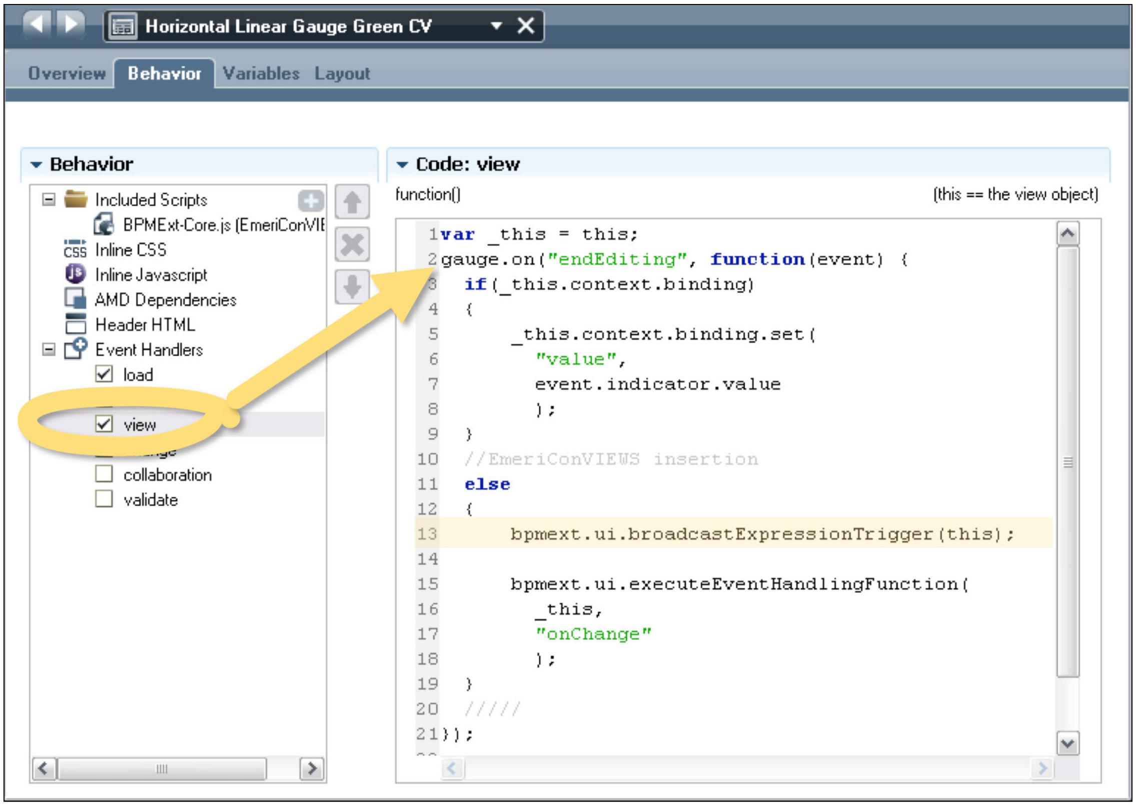
The formula support registration through setupFormulaTriggeredUpdates() is sufficient to let the Gauge Coach view participate in the EmeriConVIEWS formula framework, however, it is currently unable to push value updates to other formula-aware controls that may be using the value of this Coach View in their formula.

Enabling this last aspect is simple in this case since the locations for potential changes in this Coach View were already identified with the on change event handler. In every location where the on change event is fired, the broadcast to the formula framework will need to be fired also. The function that is used to publish a value change to the formula framework is:

```
bpmext.ui.broadcastExpressionTrigger(coachViewRef)
```

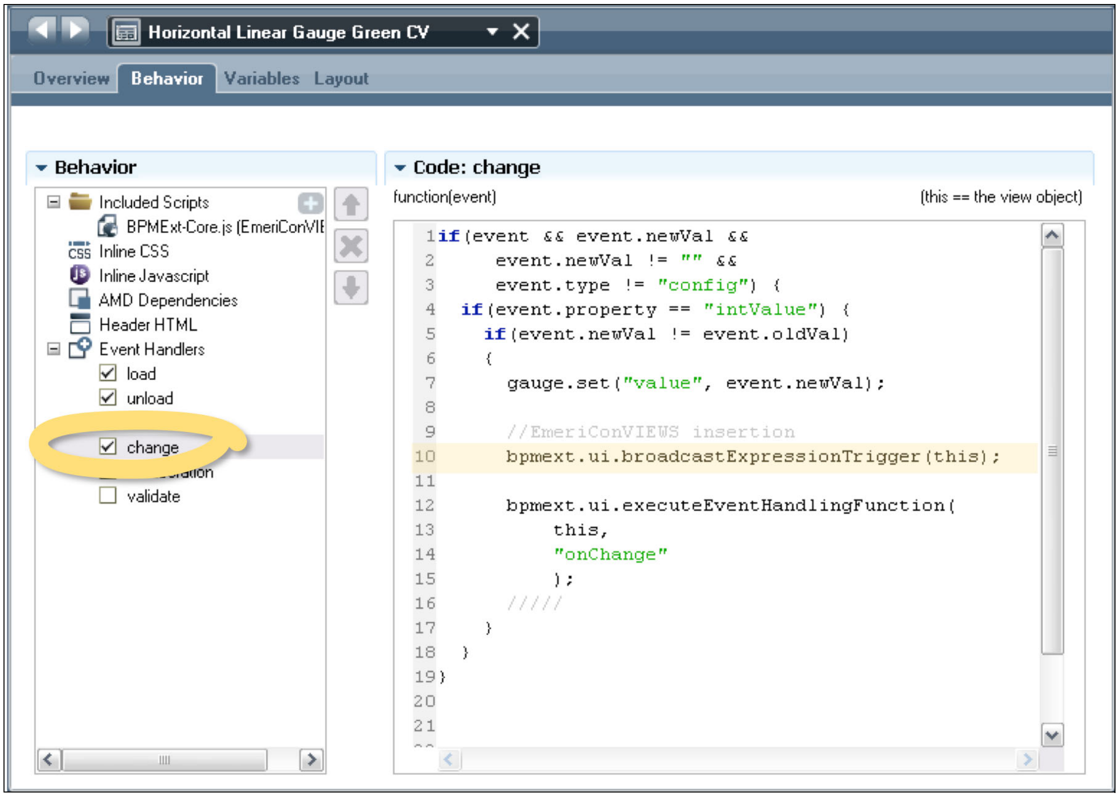
Performance: Notification of value changes through broadcastExpressionTrigger to the formula framework is performed very efficiently. Even with hundreds of controls on a form, controls only receive formula-based notifications from the other controls in which they are interested. This provides end-users with instantaneous response experience and avoids unnecessary chatter even in a Coach that is densely populated and relies heavily on formulas.

The next three illustrations that are shown in Figure 7-96, Figure 7-97 on page 348, and Figure 7-98 on page 349 shows where the broadcastExpressionTrigger call is made in the Gauge Coach View.



```
var _this = this;
gauge.on("endEditing", function(event) {
  if(_this.context.binding)
  {
      _this.context.binding.set(
        "value",
        event.indicator.value
        );
  }
  //EmeriConVIEWS insertion
  else
  {
      bpmext.ui.broadcastExpressionTrigger(this);

      bpmext.ui.executeEventHandlingFunction(
        _this,
        "onChange"
        );
  }
  /////
});
```

Figure 7-96   Value change broadcast for the Gauge Coach View (1)


Horizontal Linear Gauge Green CV

Overview | Behavior | Variables | Layout

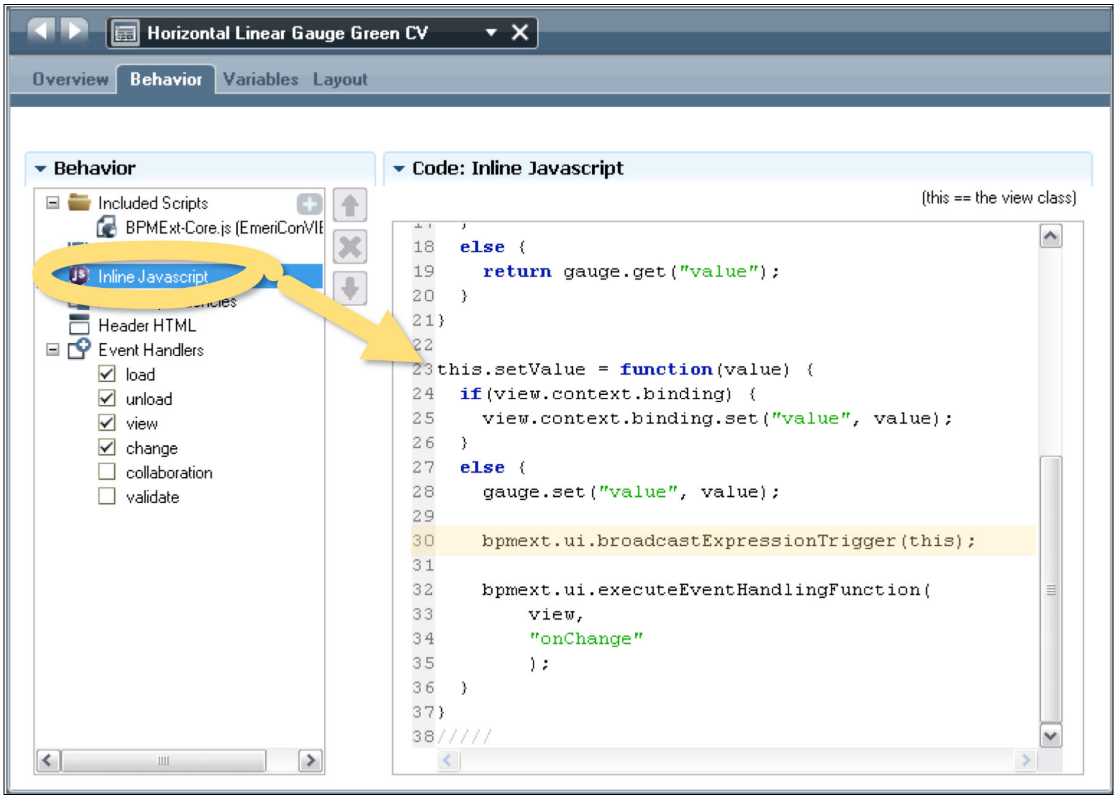
Behavior
- Included Scripts
  - BPMExt-Core.js (EmeriConVIE
- Inline CSS
- Inline Javascript
- AMD Dependencies
- Header HTML
- Event Handlers
  - [✓] load
  - [✓] unload
  - [✓] change
  - ration
  - [ ] validate

Code: change

function(event) (this == the view object)

```
if(event && event.newVal &&
    event.newVal != "" &&
    event.type != "config") {
  if(event.property == "intValue") {
    if(event.newVal != event.oldVal)
    {
      gauge.set("value", event.newVal);

      //EmeriConVIEWS insertion
      bpmext.ui.broadcastExpressionTrigger(this);

      bpmext.ui.executeEventHandlingFunction(
          this,
          "onChange"
          );
      /////
    }
  }
}
```

Figure 7-97 Value change broadcast for the Gauge Coach View (2)



- [✓] load
- [✓] unload
- [✓] view
- [✓] change
- [ ] collaboration
- [ ] validate

```
  else {
    return gauge.get("value");
  }
}

this.setValue = function(value) {
  if(view.context.binding) {
    view.context.binding.set("value", value);
  }
  else {
    gauge.set("value", value);

    bpmext.ui.broadcastExpressionTrigger(this);

    bpmext.ui.executeEventHandlingFunction(
        view,
        "onChange"
        );
  }
}
/////
```

Figure 7-98 Value change broadcast for the Gauge Coach View (3)

7.5.2 Using Coach subviews with EmeriConVIEWS

Unlike non-compound Coach Views, any subview that directly or indirectly contains at least one EmeriConVIEWS control is automatically added to the EmeriConVIEWS simplified addressing scheme. So there is no need to call bpmext.ui.loadView/unloadView explicitly for such a view.

The author of a Coach subview (meaning a Coach View that contains other Coach Views) *does not need to do anything more to start using the subview.* However, it may add convenience and usability to outfit the subview with author-friendly methods and events in a way that is similar to other EmeriConVIEWS controls.

The section entitled *"Adding logic to reusable Coach Views" on page 253* provides an overview of how a method can be added to a Coach subview, and how the added method can be used.

7.5.3 Interacting with non-EmeriConVIEWS controls

EmeriConVIEWS are fundamentally Coach NG Views and can be seamlessly mixed with non-EmeriConVIEWS controls without any need for coding adjustments or reconfiguration.

However, although there no added complexity to mixing controls, non-EmeriConVIEWS controls must be interacted on their own terms and may not provide the referencing, encapsulation, and event-handling benefits provided by EmeriConVIEWS controls.

7.5.4 EmeriConVIEWS-enabling other UI toolkits

As shown in *7.5.1, "Making EmeriConVIEWS views out of regular Coach NG views" on page 334*, a UI toolkit can be EmeriConVIEWS-enabled by adding the EmeriConVIEWS toolkit (assuming it has already been imported in the Process Center Server environment) as a toolkit dependency, as shown in Figure 7-99.

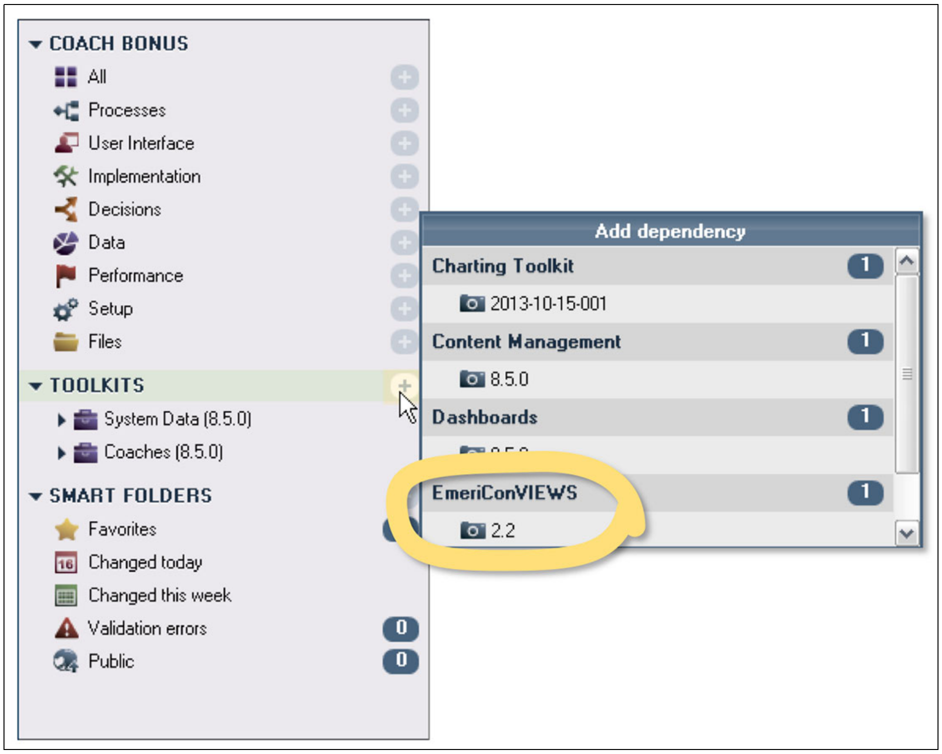


Figure 7-99 Adding the EmeriConVIEWS toolkit dependency

Once the dependency is added, the Toolkit (or Process Application) can use EmeriConVIEWS controls and create its own controls with the referencing, event handling, formula support made available by following the guidelines provided in 7.5.1, “Making EmeriConVIEWS views out of regular Coach NG views” on page 334 section.

7.6 Conclusion

EmeriConVIEWS provides a consistent BPM Coach UI authoring experience and normalizes key aspects, such as view referencing, event handling, formulas, the creation of new controls, working at the Coach Page level or inside a Coach View, or creating responsive Coach UI behaviors.

Instead of bringing together potentially disparate approaches in a BPM solution with different toolkits working in different ways, EmeriConVIEWS offers the BPM author a familiar authoring experience, similar to other well-known UI-authoring environments, where the author has the choice to focus the UI development effort on solving business problems in a highly consistent manner. EmeriConVIEWS does not in any way preclude lower-level web UI programming using HTML5, Dojo, jQuery, Ajax, HTML, CSS, the Coach NG API – but it does not mandate it.

EmeriConVIEWS is also fully compliant with existing Coach NG capabilities and approaches (such as server-side validation, client-side visibility controls, stay on page) and simply provides additional convenience and sometimes performance benefits to the base Coach NG framework.

In addition to the core aspects of the toolkit, EmeriConVIEWS also contains a rich set of Coach Views/controls and capabilities for enhanced form entry fields, electronic signature, charting and reporting, data analysis, responsive design for Coach UIs on mobile devices, and high-performance tabular data.

7.7 FAQs

This section provides a sampling of topics and questions frequently asked by IBM BPM customers and EmeriCon business partners regarding EmeriConVIEWS.

How does EmeriConVIEWS deal with BPM upgrades?

EmeriConVIEWS has been in existence since the introduction of the Coach NG framework with BPM 8.0. Although many new Coach View types (meaning controls) have been added over time, the core framework itself has remained stable. EmeriCon makes updates to EmeriConVIEWS when a new BPM product version introduces the need for adjustments or opportunities for feature enhancements.

EmeriCon typically produces EmeriConVIEWS toolkit updates within two weeks of an official IBM BPM product release.

Why extend IBM BPM stock controls instead of creating new ones?

EmeriConVIEWS contains both IBM extended Coach Views and completely new ones.

All Coach Views from the Coaches toolkit are indeed extended in EmeriConVIEWS. This allows EmeriCon to take advantage of IBM updates and fixes, incorporate them into the extended stock controls and make those IBM updates available to users of EmeriConVIEWS.

The intent of this approach is to make the latest IBM stock controls, available in their latest state of fixes or updates, with the added benefit of the EmeriConVIEWS usability enhancements.

What is the minimum browser requirement for EmeriConVIEWS?

EmeriConVIEWS works on all major modern web browsers, including Chrome, Chrome on Android, Firefox, Internet Explorer 9+, and Safari on the iPad or the iPhone.

Only a subset of EmeriConVIEWS controls work on Internet Explorer 8. The performance of the JavaScript engine under Internet Explorer 8 also provides a less responsive end-user experience than Internet Explorer 9. For these reasons, broadly using EmeriConVIEWS for Coach UIs under Internet Explorer 8 is discouraged.

Can EmeriConVIEWS be used with other UI toolkits?

Yes, either side by side, or to extend other toolkits with EmeriConVIEWS capabilities. EmeriConVIEWS is always dependent on the most current Coaches and System Data toolkits for the BPM product under which it is loaded. To reduce the potential for compatibility issues, other toolkits should do the same.

The approach to use EmeriConVIEWS with other Coach UI toolkits is explained in *7.5, “Mixing EmeriConVIEWS with other toolkits” on page 334*.

How to obtain EmeriConVIEWS

To request a copy of the EmeriConVIEWS toolkit, you can contact EmeriCon at emericon.sales@emericon.com, or see the following website:

```
http://www.emericon.com/emericonviews
```

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *Business Process Management Deployment Guide Using IBM Business Process Manager V8.5*, SG24-8175
- *IBM Business Process Manager Version 8.0 Production Topologies*, SG24-8135
- *IBM Business Process Manager V8.0 Performance and Tuning Best Practices*, REDP-4935

You can search for, view, download, or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

This website is also relevant as a further information source:

- The IBM Business Process Manager V8.5 information center can be found here:

  http://pic.dhe.ibm.com/infocenter/dmndhelp/v8r5m0/index.jsp?topic=%2Fcom.ibm.wbpm.main.doc%2Fic-homepage-bpm.html

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

A

Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

ftp://www.redbooks.ibm.com/redbooks/SG248210

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG248210.

Using the Web material

The additional Web material that accompanies this book includes the following files:

File name	*Description*
Chapter_2_Sample - V1.twx	Zipped Code Samples used in Chapter 2, “Assembling user interfaces with Coach Views” on page 11.

Downloading and extracting the Web material

Create a subdirectory (folder) on your workstation, and extract the contents of the Web material .zip file into this folder.

IBM

Redbooks

Leveraging the IBM BPM Coach Framework in Your Organization

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

Leveraging the IBM BPM Coach Framework in Your Organization

Understand how to develop and assemble user interfaces with Coach Views

Investigate three IBM Business Partner offerings from:

Apex Process Consultants, BP3, and EmeriCon

The IBM Coach Framework is a key element of the IBM Business Process Manager (BPM) product suite. With the Coach Framework, process authors can create and maintain custom web-based user interfaces that are embedded within their business process solutions. This ability to create and maintain custom user interfaces is a key factor in the successful deployment of business process solutions. Coaches have proven to be an extremely powerful element of IBM BPM solutions, and with the release of IBM BPM version 8.0 they were rejuvenated to incorporate the recent advances in browser-based user interfaces.

This IBM Redbooks publication focuses on the capabilities that Coach Framework delivers with IBM BPM version 8.5, but much of what is shared in these pages continues to be of value as IBM evolves coaches in the future. This book has been produced to help you fully benefit from the power of the Coach Framework.

SG24-8210-00 ISBN 0738439363