IBM Cúram Social Program Management
Version 6.0.5

Cúram Custom Widget Development Guide

Note
Before using this information and the product it supports, read the information in "Notices" on page 85

Revised: March 2014

This edition applies to IBM Cúram Social Program Management v6.0.5 and to all subsequent releases unless otherwise indicated in new editions.

Licensed Materials - Property of IBM.

© Copyright IBM Corporation 2012, 2014.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

© Cúram Software Limited. 2011. All rights reserved.

Contents

© Copyright IBM Corp. 2012, 2014

Figures

© Copyright IBM Corp. 2012, 2014

Tables

© Copyright IBM Corp. 2012, 2014

Developing Custom Widgets

Use this information to develop custom widgets for UIM pages. A comprehensive set of widgets are provided, which are configured against the application's domain definitions by default. These configurations can be changed as required.

Introduction

Objective

The objective of this guide is to explain when it is appropriate to use a custom widget to present the content of a UIM page and to show how to develop such a widget and integrate it into the application.

The text within the images used throughout this guide are intentionally blurred because we are only concerned with the high level details of these widgets. Each number in an image maps to a specific detail in a widget. A list is given below each image to explain its details by referring those numbers.

The objective of this chapter is to explain briefly what widgets are, what can be achieved through the customization of widgets and how the rest of this guide is structured to aid the developer in the task of developing custom widgets.

Audience

This is a guide for client application developers who want to customize the presentation of Cúram application pages in ways that are not possible through UIM or through the reconfiguration of the set of widgets provided in the Cúram Client Development Environment (CDEJ).

Prerequisites

The developer should be proficient in Cúram client-side application development in Java™ and UIM. In addition, knowledge of HTML, JavaScript, CSS and other web application technologies is required to varying degrees depending on the nature of the widget being developed.

What's New?

UIM provides support for easy development of a consistent application user interface and can meet most presentation requirements. However, sometimes there is a requirement for richer functionality or a more sophisticated look than can be achieved with UIM alone. Cúram 6.0 introduces support for the customization of *widgets*. Widgets are the elements of the user interface used to present the values of the fields defined in UIM, such as simple text values, editable text fields, date selectors, bar charts and calendars. The new custom widget development features make it possible for developers to create their own widgets that supplement or replace those provided by the CDEJ. Here are just a few examples of the kinds of customizations that can now be performed:

- The configuration can be changed so that the basic text field widget is used for the input of all date values, instead of the date selector that is configured by default;

© Copyright IBM Corp. 2012, 2014

- The presentation of all e-mail address values can be customized so that, instead of being shown as simple text, they are shown as HTML `mailto:` links beside an e-mail icon;
- A photograph of a person stored in the application database can be displayed as the value of a field;
- The details of a person can be presented using a richer and more compact layout than possible with a UIM `CLUSTER`;
- Widgets can be reused within other widgets, so that the e-mail address widget can be reused within the widget that displays the details of a person and that details widget can, in turn, be combined with the widget that displays a photograph of a person to create a single widget that presents a more engaging summary of a person in a tab context panel.

Customizing Widgets

Customizing widgets is a process that involves customizing the HTML that is produced to represent the value of a field. A client application developer defines a Cúram application page using UIM, but the page is displayed in a user's web browser using HTML. Behind the scenes, the CDEJ translates the `CLUSTER` and `LIST` elements of the UIM page into HTML elements and then presentsor *renders* the labels and values of the `FIELD` elements within the structure provided by those HTML elements. Typically, the CDEJ renders a cluster or list using a HTML table and then places the labels and values of the fields into the cells of that table. The CDEJ renders the label of a field the same way for all fields, but renders the HTML for the value of a field in different ways depending on the type, the domain definition, of that field's value.

The processing of field values in a domain-specific manner has been available since Cúram 4.0. This support for custom data conversion and sorting is described in detail in the *Cúram Web Client Reference Manual*. Using the same configuration mechanism, the CDEJ now extends this domain-specific customization to the widgets used to produce the HTML for the values of fields. The CDEJ includes a default configuration that associates the provided *Cúram widgets* with all of the the domain definitions of the application. The CDEJ now also supports these key features:

- The customization of the default configuration by the application developer, providing the freedom to change what widget is used to render the value of each type of field;
- The development of new widgets by the application developer and their integration into the application through the customization of the default configuration. These *custom widgets* allow full control over the rendering of values for individual UIM `FIELD` elements.

Custom widgets are integrated into the application in a manner that preserves all of the time-saving and simplifying features of UIM development. However, developing custom widgets can be a complex process. Widget developers take on the responsibility for considerations such as styling, internationalization, cross-browser support and other concerns from which they are insulated when using UIM alone. There is a balance to be achieved between ease of development and maintenance on the one hand and user interface richness and flexibility on the other.

Cúram widgets and custom widgets differ only in where they are developed and configured, not how. Therefore, custom widgets are a powerful tool for application developers who need to meet challenging presentation requirements by

complementing or replacing the provided Cúram widgets. The development and configuration of such custom widgets is the subject of this guide.

Outline of this Guide

The next chapter, "Approaches to Customization" on page 4 guides the developer on the choice of approach to achieving the required customization of the user-interface while minimizing the development effort.

"How Widgets Work" on page 9, presents more detailed information about the components of a widget and their configuration.

"An E-Mail Address Widget" on page 13 introduces the fundamental principles of the widget development process and the subsequent widget configuration. The chapter shows how to create a simple widget that presents an e-mail address more appealingly in the context of a typical UIM page.

"The Sample Context Panel Widgets" on page 18 presents some samples of context panels used within the tabbed user interface. These sample context panels are constructed using several complex widgets that are supplied with data in XML form. The development and configuration of each of these widgets is covered in the following chapters. Each chapter introduces new concepts in widget development that build upon what has gone before until the complete context panels have been created and configured.

All of the widgets described to that point are used to present read-only values. "A Text Field Widget with No Auto-completion" on page 39 introduces a widget for editing values on a form page. Widgets used to edit values have some unique requirements that are not applicable to widgets that present read-only values. To edit a value, a widget must ensure that, once the user submits a form page containing the widget, the entered field value reaches its destination on the server interface and that any validation errors are handled correctly.

Often, the deployed Cúram application must comply with local regulatory requirements for the localization of text and the accessibility of the user-interface. While the details differ between jurisdictions, the general principles are common to all. "Internationalization and Localization" on page 44 and "Accessibility Concerns" on page 47 outline the main principles.

This is not a comprehensive reference manual for widget development. References to external sources of information, such as the published Javadoc of the CDEJ, will be used to draw the attention of the developer to additional information when necessary. The developer should also study the primary companion guide, the *Cúram Web Client Reference Manual*, before embarking on custom widget development. Several appendixes at the end of this guide supplement these other sources where they lack specific information related to widget development. Throughout this guide, the developer's attention will be referred to the relevant appendix as appropriate.

Conventions of this Guide

For clarity, the source code presented throughout this guide is abridged. Import statements are omitted and package names are not shown. "Source Code for the Sample Widgets" on page 78 provides the full, unabridged source code listings showing the import statements that identify the package names of the referenced classes and interfaces.

Similarly, the configuration files in the examples show only the domain configuration entry relating to the configuration of the widget just presented. The real configuration file within an application component will typically contain all of the configuration entries for all of the domain definitions to which customizations have been applied.

Limitations and Restrictions

The focus of this guide is on the development of custom widgets for inclusion into context panels within the tabbed user interface. Other uses of widgets are covered only briefly or not at all.

warning: No Implied Support

Only the custom widget functionality described in this document is supported. No other functionality, whether inferred by the reader through extrapolation or analysis of the Javadoc or other sources, is supported. Neither is support offered for use of custom widgets in contexts other than those contexts presented in this document.

Throughout this guide, other limitations or restrictions will be highlighted in the relevant contexts.

Approaches to Customization

Objective

To understand when UIM should be used to define all of the content of a page, when a custom widget is required to achieve a presentation requirement and what the scope of the custom widget should be.

Prerequisites

A basic knowledge of the capabilities of UIM and the structure of web pages rendered from UIM sources.

Identifying the Right Approach

UIM pages can define the content of an application page in terms of fields, action controls, clusters, lists and other elements. UIM provides enough control to present the page content in ways that meet most presentation requirements. Alternatively, instead of using multiple fields in clusters and lists in a UIM page, a single field can be used in the UIM to anchor a custom widget that produces most of the HTML content of the page. Between these two bounding approachesdoing it all with UIM or doing it all with a widgetthere are several intermediate approaches. Where a requirement for customized presentation is identified, the developer needs to assess the necessary extent of that customization and how best to meet the requirement to minimize the complexity and effort required.

While the development of custom widgets provides greater control over the presentation of the content than UIM, this control comes at the cost of greater complexity. Trying to do everything from one widget by producing large amounts of HTML content can lead to significant long-term maintenance overheads. This is particularly so if the appearance of the content needs to be kept consistent with content produced from standard elements of a UIM page or with content from Cúram widgets. For example, if a custom widget attempts to produce HTML output that looks the same as that produced for a standard UIM `CLUSTER`, that may

introduce a long-term requirement to repeatedly reverse engineer the potentially changing structure of that HTML. The HTML structure and CSS produced by the CDEJ is subject to change and it cannot be guaranteed that customizations that depend on this HTML structure or CSS styling will continue to work when the Cúram application is upgraded. Therefore, while a custom widget could present all of the page content, it is usually best to limit what the custom widget produces and to produce as much of the content as possible using UIM.

Attempt to meet the presentation requirement by selecting the first approach listed below capable of meeting the requirements. These approaches are listed in order of increasing complexity and are described more fully in the following sections.

- Use only UIM, though perhaps use it more creatively than is typical.
- Reconfigure the standard widgets to change the presentation of the field values.
- Develop and use one or more simple custom widgets and use them in combination with UIM.
- Develop and use one or more complex custom widgets instead of many UIM elements.
- Apply some combination of the above approaches.

Using Only UIM

Before deciding to develop a custom widget, the developer should first assess if the required presentation can be achieved using the layout and styling capabilities supported by UIM. If the presentation requirement can be achieved using only UIM, there will be no need to develop a custom widget and time and effort can be saved.

UIM allows `CLUSTER` and `LIST` elements to be nested within other `CLUSTER` elements. The number of columns in a cluster can be controlled, as can the display of the titles of clusters and lists and of the labels of their contained `FIELD` elements. This flexibility can exploited to achieve quite complex page layouts. See the *Cúram Web Client Reference Manual* for more details on these UIM elements.

Many UIM elements also support a `STYLE` attribute that can be used to associate a custom CSS class with the HTML content generated in respect of those elements. The custom CSS class can define styles that control many aspects of the presentation. Fonts, background images, spacing, borders, colors and other aspects of the presentation can be customized easily. See the *Cúram Web Client Reference Manual* for more details on the use of the `STYLE` attribute and on the inclusion of custom CSS resources.

The developer may identify a UIM-only solution to the presentation requirement, but may need to apply this to many pages. Doing this one page at a time may not be desirable, particularly if later changes would also require that all of the pages be updated again. Using a UIM `VIEW` in a VIM file and including this view into many UIM files may meet this requirement.

If the requirement is to change the presentation of a field value in a significant way, rather than to change the page layout and/or make minor styling changes to the content, then this approach of using only UIM may not be sufficient. If the customization needs to be repeated across many pages in a way that cannot be accommodated by included views (VIM files), or in a way that imposes significant maintenance overheads, then this approach may also be insufficient. In those cases, a more advanced approach may be necessary, such as the reconfiguration of the standard widgets or the development of a new widget.

Reconfiguring Standard Widgets

Cúram provides a comprehensive set of widgets that are configured against the application's domain definitions by default. The application developer has the option to change (override) this configuration to meet the presentation requirements. Such reconfiguration can change the standard widget used for a particular type of data to be a different standard widget. Where custom widgets have been added to the application already, these custom widgets are also candidates for reuse through reconfiguration.

For example, the date selector widget is used for fields in the SVR_DATE domain (and its descendant domains). If the requirement is to change the date selector to a simple text field, possibly formulated as, "Remove the pop-up calendar icon," then a new date selector that acts like a text field is not required. This requirement can be met simply by associating the same widget used for the SVR_STRING domain (and many numeric domains) with the SVR_DATE domain. This configuration change, made in a configuration file in the application component, will cause all SVR_DATE values on all pages to be presented for editing with a simple text field.

The elements of a widget that are configured in this way are explained in the next chapter and the configuration process is covered in detail in "Configuring Renderers" on page 65. Also described in that appendix are the names and locations of the configuration files, including the default configuration file that shows what is configured as standard in the CDEJ.

If a reconfiguration of the widgets by changing the domain associations, perhaps in combination with the creative use of UIM, cannot meet the presentation requirement, it may be necessary to develop a new custom widget and configure it for use.

Developing Simple Custom Widgets

A widget controls how the value of a field is presented by adding the HTML mark-up to the value that is appropriate for that presentation. Reconfiguring the widgets associated with different domain definitions and restyling the HTML of existing widgets with custom CSS are not always sufficient to meet a presentation requirement. If the developer decides that the presentation requirement can only be satisfied by modifying the structure of the HTML produced for the value of a field in a manner that no existing widget can achieve, then the developer must write a new widget and configure it for use by the application.

"An E-Mail Address Widget" on page 13 explains how to develop a simple widget for viewing the value of a field; "A Text Field Widget with No Auto-completion" on page 39 explains how to develop a simple widget for editing the value of a field. Both chapters describe briefly how to configure these widgets and more information about the configuration of custom widget can be found in "Configuring Renderers" on page 65.

In the simple case, a widget will replace the HTML content produced for the value of a UIM `FIELD` within the context of a normal UIM `CLUSTER` or `LIST`. The value of the field will still be a single string, number or date, only styled more elaborately. The general layout of the page will not be affected. Where the presentation requirement has a wider scope and requires that the layout of significant parts of the page be changed, or that the value of a field contain many embedded values, such as in an XML document, a more complex widget will be required.

Developing Complex Custom Widgets

There is no clear dividing line between simple widgets and complex widgets. The more control over the presentation that the developer exerts through a custom widget, the more complex the implementation of that widget will become. Some indicators of increased complexity are:

- The value of the field may be more than a simple string or numeric value. For example, the value may be an XML document containing several separate values, such as the data for a bar chart.
- Multiple values may be presented to the user differently from the usual grid layout of a cluster or list. For example, a photograph of a person may be presented with the person's name below the image and with no field label to the side.
- A widget may present information by delegating significant parts of the presentation to the renderer plug-ins of other widgets. For example, in presenting a non-grid layout for the details of a person, the value of the single UIM field may be an XML document containing all of those details. A single widget is invoked by the CDEJ for that XML document value. That widget may then produce the non-grid layout in HTML and, in each position within this layout, delegate the rendering of the values within the XML document to other widgets. This is similar to the way the CDEJ delegates to widgets when rendering the contents of the cells in the grid layout presented by a UIM cluster.

While a UIM `FIELD` is always required to anchor a custom widget, a UIM page can contain little more than a single `FIELD` element and leave most of the rendering of the HTML page content to the associated custom widget. (The page title and other surrounding content are still rendered independently of the field.) The ability to place a UIM `FIELD` element directly within a `PAGE` element without any `CLUSTER` or `LIST` element, is a new feature of the CDEJ. While it allows a widget more control over the layout of the data, this approach should only be used if the presentation requirement is such that it cannot be achieved using only UIM, or using a combination of UIM and one or more simple widgets.

Even if a presentation requirement *can* be met using only UIM, the developer may prefer to use a custom widget to allow the customization to be applied automatically to many application pages, via the domain definition association, rather than repeat the UIM-only solution on every page that needs it. Where the use of VIM `VIEW` elements cannot achieve this, a complex custom widget may be necessary.

This guide presents the development of several complex widgets in later chapters. The developer should not assume that because much of the guide is concerned with the development of complex widgets that complex widgets are the preferred approach. On the contrary, much of this guide covers complex widgets because their very complexity requires more explanation. The developer should always opt for the simplest possible approach first and only resort to complex widget development when there are no simple alternatives.

Mixing Simple Custom Widgets with UIM

The complexity of a widget increases as it assumes more and more control over the layout of more and more data. If a presentation requirement cannot be met using only UIM, the developer may need to create a custom widget. However, the complexity can be reduced by developing only the widgets that are absolutely necessary and using UIM as much as possible to achieve the goal. The developer

should assess if a combination of UIM with several simple widgets could achieve the desired result, or if a full, single custom widget is the only solution.

The developer can use UIM clusters, lists and fields in various combinations to produce HTML output that is close to what is required. The developer may then associate simple custom widgets with individual fields, replacing the default HTML content for those fields with custom content. Further, the developer may replace the presentation of a cluster on the page with a presentation produce by a single custom widget, which still using UIM clusters elsewhere on the same page. The combination of default content for the main layout of the page with changes to the content for individual fields or individual clusters, is generally easier to achieve than using a single custom widget to produce all of the page content.

Constructing pages from several, simpler custom widgets reduces the complexity of the individual widgets. It also results in a number of simpler widgets that are much easier to reuse in other contexts. The developer may identify that some widgets could be developed in a way that makes them a component of the solutions to the differing requirements of several pages. In this case, the alternative approach of a single custom widget that can only satisfy the requirements of a single page, is likely to be more complex to develop and result in further development of other complex widgets for other pages with little reuse.

Responsibilities of the Widget Developer

This chapter has presented the approaches to the customization of widgets in increasing order of complexity. The widget developer, in eliminating a simpler approach and moving on to consider a more complex approach, takes on more responsibility for the proper operation of the resulting user interface. UIM insulates client application developers from most of these responsibilities, but this insulation is, to a significant extent, provided by the widgets that underlie the UIM fields. Therefore, the widget developer is responsible for ensuring that the custom widget continues to insulate the UIM developer from concerns such as the following:

- The Cúram user interfaces evolves with each new release. Widgets that attempt to emulate the output produced by standard elements of the Cúram user interface, such as clusters and lists, will need to evolve in step with Cúram to ensure that the consistency of presentation of the user interface is preserved. This is a long-term maintenance task that should be considered as part of the cost of development of any such custom widget.
- Rendering HTML to the application page is a low-level process. It offers considerable power and flexibility to customize the application. However, it also, by its nature, opens up the possibility of introducing unwanted side-effects that interfere with the presentation of other parts of the application page, or introducing security defects, such as vectors for cross-site scripting (XSS) attacks. The widget developer assumes the responsibility for ensuring that such defects are not introduced.
- Complex widgets with ambitious presentation requirements can be an expensive undertaking. Much of the development effort goes not into developing the widget source code, but into fine-tuning the styling of the HTML for that widget within the browser. Where there is a requirement for cross-browser support, either different versions of the same web browser, or different web browsers entirely, the time required to achieve a consistent look across all web browsers should not be underestimated.

- The CDEJ provides considerable assistance to the widget developer to aid with the internationalization of a widget. However, this assistance is only of value if the widget developer takes advantage of it to ensure that the widget can be properly localized after development.
- The widget developer may not have a free hand to implement all presentation requirements as specified. Most jurisdictions implement regulations and guidelines requiring that web applications be available and accessible to as many people as possible and, in particular, be inclusive of those with disabilities. The technical requirements may differ between jurisdictions and it is the responsibility of the widget developer to understand and comply with any such requirements.
- The perceived quality of the application can be diminished if a widget does not operate correctly or if it introduces inconsistencies or unwanted side-effects. As the complexity of a widget increases, so too does the effort required to test it in all of its aspects and to ensure that it enhances, not degrades, the application and the experience of the users. The widget developer should not underestimate the effort required to test a complex widget properly and the need to test it repeatedly as the application is further customized or upgraded.

This guide explains these concerns in more detail in the later chapters and appendixes and advises on how they can be addressed. By choosing the simplest approach possible to achieve a presentation requirementafter evaluating if the presentation requirement can be modified to permit a simpler approachthe widget developer can minimize the effort required to meet all of these added responsibilities.

How Widgets Work

Objective

To understand the components of a widget and the principles of their development and configuration.

Prerequisites

A basic knowledge of the capabilities of UIM and the basic principles of web application development in HTML.

Introduction

As described in the previous chapter, a developer defines a Cúram application page using UIM, but the page is displayed in a user's web browser using HTML. The label of a field is presented the same way for all fields, but the HTML that presents the value of each field differs depending on two factors: the *mode of operation* of the field and the type, the domain definition, of its value.

There are two modes of operation: the view mode and the edit mode. In the view mode, the user cannot modify the value of the field. The user may see the value presented as just text, or presented more elaborately as a bar chart or a rate table, depending on the type of the value. In the edit mode, the user can enter a new value or modify the existing value of a field. The user may see the value presented in a simple text input field, or a date selector or a check-box, again depending on the type of the value.

For each mode of operation and type of data, a specialized component is invoked by the CDEJ to render the HTML for a field's value. This HTML is included into

the full HTML page and the page is returned for presentation to the user by the web browser. Often, other resources, such as icons and JavaScript, are required to complete that presentation. These specialized rendering components together with their associated resources are called *widgets*. Thus, there is a date selector widget, a text field widget, a bar chart widget, and many other widgets. The CDEJ provides a comprehensive set of widgets for all modes of operation and types of data. These are detailed in the " *Domain Specific Controls* " chapter of the *Cúram Web Client Reference Manual* and further in this guide in "Configuring Renderers" on page 65.

When rendering a complete UIM page at run-time, the CDEJ automatically identifies the mode and type of each UIM `FIELD` and selects the appropriate widget to render the value. The mode of operation is determined by the presence or absence of a `TARGET` connection on that field. When that connection is present, the field is in the edit mode; when it is absent, the view mode. The type of a field is determined by the domain definition of the server interface property to which that field is connected. What widget is "appropriate" for any given combination of mode and type is defined by configuration. A configuration file associates widgets with named domain definitions. For each domain definition, the widget to be used for each mode is specified. The CDEJ will use a widget so configured whenever it needs to render the value of a field with a matching mode and domain definition.

The configuration used by the CDEJ to associate widgets with domain definitions is the same configuration used to associate custom converter and comparator plug-ins with domain definition. The development and configuration of these plug-ins are described in the " *Custom Data Conversion and Sorting* " chapter of the *Cúram Web Client Reference Manual*. Custom widget development involves the development and configuration of new types of plug-ins that are configured in the same way. The widget developer can define a configuration within the application that overrides the default configuration of the CDEJ to customize the associations between widgets and domain definitions and change how the values of fields are presented. In order to do this, the widget developer must first understand the relationship between widgets and domain-specific plug-ins.

Anatomy of a Widget

"Introduction" on page 1 and the introduction to this chapter described what a widget is and outlined how a widget is integrated into the application. However, the widget developer must be familiar with the anatomy of a widget in more detail before developing one.

To a user, a widget is just what is shown in the web browser. To a widget developer, a widget comprises all the resources involved in the generation and presentation of what a user sees. From this development perspective, a widget may be comprised of many artifacts that, together, realize a presentation requirement for a specific type of data in one mode of operation.

The common artifacts of a widget are as follows:

Renderer Plug-in

The main component of a widget is its *renderer plug-in*, the Java class that generates the HTML mark-up around the field value. The renderer plug-in class is the only artifact required for every widget. The CDEJ provides abstract base classes that all custom renderer plug-in classes *must* extend. There is a different base class for each mode of operation. Each renderer plug-in class has a `render` method that must be implemented to generate the HTML content using the W3C DOM Core API.

Custom renderer plug-in classes are placed into the `javasource` folder of the chosen client application component. The classes can be added to a Java package sub-folder, but the Java package name should not conflict with the name of the Cúram application packages. Throughout this guide, the package folder `sample` is used, but the use of that name is neither required nor recommended.

The presentation requirement of a widget can sometimes be realized with nothing more than a single renderer plug-in class. In this case, the terms *widget* and *renderer* may be synonymous to a developer. However, most widgets require additional resources, and sometimes additional renderer plug-in classes, so the term *widget* has a wider scope than *renderer*.

Domain Configuration

A configuration file associates domain definitions with the renderer plug-ins of widgets. One file named `DomainsConfig.xml` is permitted in each application component. The same configuration file is used for other types of plug-ins, such as those used to customize sorting and data validation described in the *Cúram Web Client Reference Manual*. The change to the domain configuration required to associate a custom widget's renderer plug-in class with a domain must be added to this file and the file must be created if it does not already exist. The configuration process is covered as required in the other chapters of this guide and in more detail in "Configuring Renderers" on page 65.

JavaScript

JavaScript can be incorporated in two ways by a widget. Both are controlled by the renderer plug-in class. The renderer plug-in can embed JavaScript code directly into the HTML using `script` tags, or it can request that the CDEJ add a link to the page to include a separate JavaScript resource. It is common for a renderer plug-in to do both: include a link to a JavaScript resource and then add scripts that invoke the functions defined in that resource. External JavaScript resources should be placed into the application component. They will be copied into the correct location during the build.

Images

Images can be included by embedding a HTML `img` tag with the appropriate value for its `src` attribute. For images such as icons, the image files can be placed into the application component. For images, such as photographs stored on the database, a special source URL is required. Examples of both approaches are presented in the later chapters of this guide.

CSS CSS can be used to separate the styling of the HTML produced by a renderer plug-in from the operation of that plug-in. Like JavaScript and image resources, CSS resources are not directly associated with a widget. They are just added to the application component. Unlike JavaScript and image resources, CSS resources are not requested explicitly by a renderer plug-in. The style rules defined within a CSS resource, and all other CSS resources in the application components, are automatically combined into a single new CSS resource during the build process. The specific CSS resource is not referenced anywhere in the HTML, but the rules will be applied nonetheless. See the *Cúram Web Client Reference Manual* for more details on the incorporation of custom CSS resources.

Localized Text Properties

Any text produced by a renderer plug-in other than the actual field value is usually required to be internationalized, i.e., to support localization into

different languages. Standard Java properties resources, as defined by the Java `ResourceBundle` API are supported for this purpose. The techniques for locating these resources and referencing their content are covered in "Internationalization and Localization" on page 44.

Widgets can use or depend on other artifacts, such as Java libraries, supporting Java classes, XSLT stylesheets, XML schemas, and many others. The use of such artifacts is dependent on the nature of the widget and what it must achieve. This guide does not describe the use of such artifacts or their integration into an application. A widget developer will not be supported in the resolution of any issues related to the use of artifacts, or types of artifact, not explicitly covered in the later chapters of this guide.

How Widgets Work

"Introduction" on page 1 described the modes of operation of widgets and outlined how widgets can be associated with domain definitions. The widget developer will benefit from a deeper understanding of this process and of its dynamics.

As explained in the previous chapter, widgets are selected and invoked automatically by the system depending on the type of data and mode of operation of a field. In UIM, each `FIELD` is associated with data using `SOURCE` and/or `TARGET` connections. The system identifies the type of the data based on the domain definition of the server interface property named on those connections. The domain definition for the `TARGET` connection is preferred over that of the `SOURCE` connection. The mode is determined by the presence or absence of the `TARGET` connection; if a `TARGET` connection is present, the edit mode is used; if only a `SOURCE` connection is present the view mode is used.

A configuration file associates the widgets' renderer plug-in classes with domain definitions, so that, for any given type of data and mode of operation, the same renderer plug-in class is invoked on every page to present that data with the appropriate HTML mark-up. A widget's renderer plug-in class can identify itself as either a *view-renderer* for the view mode or an *edit-renderer* for the edit mode, but not both, so a separate renderer plug-in class is required for each mode. The configuration allows one edit-renderer plug-in class and one view-renderer plug-in class to be associated with each domain definition. If the developer changes the configuration file so that a custom widget's renderer plug-in class is associated with a domain definition, then every time a field in that mode with a connection to data in that domain is presented on any page, the custom renderer plug-in class will be used. Thus, the developer can produce any desired custom HTML mark-up to present the data of any UIM `FIELD` and see that applied consistently across the application.

The same widget is often used for many different types of data in a given mode. For example, the application presents the majority of view-only data using a single widget that simply inserts the text representation of that data into the HTML without any HTML element mark-up. Only where the presentation is more specialized are specialized widgets applied.

The CDEJ invokes widgets in the course of transforming a UIM page to HTML. For widgets associated with UIM `FIELD` elements, this always happens at run-time. During the rendering of the page, the CDEJ constructs a `Field` object from the information defined in the UIM. Using this information, it consults the domain configuration to select the appropriate widget's renderer plug-in and then passes

the `Field` object to the renderer plug-in along with an empty DOM `DocumentFragment` object. Using the information provided by the `Field` object, the renderer plug-in uses the DOM Core API to create the DOM nodes representing the required HTML and field value and adds these nodes to the `DocumentFragment` object. When the renderer plug-in returns, the CDEJ will take the now populated `DocumentFragment` object, serialize it to a HTML text stream, and add this to the stream being returned to the web browser. By this method, *any* HTML content can be produced by the renderer plug-in class.

The developer can implement a widget such that multiple renderer classes are used together to achieve a presentation requirement. The CDEJ first invokes a single renderer plug-in class based on its association with a domain definition. That renderer class can then delegate the rendering of elements of its output to other renderer classes. The first renderer can create empty DOM `DocumentFragment` objects of its own and pass them on to the other renderers. These renderers will populate the fragments with HTML nodes and the first renderer can add the contents of those fragments to its own before returning control to the CDEJ. Combining renderer classes together into such a *rendering cascade* simplifies the individual renderer classes and maximizes the potential to reuse these classes in other combinations to realize new custom widgets. Examples of this process will be presented in later chapters of this guide.

The configuration file, identified in the previous section, that associates renderer plug-in classes with domain definitions is subject to the same type of component-order-based merging as most other configuration files in the Cúram client application. In simple terms, the CDEJ default domain configuration is loaded first. Then the domain configurations defined (if at all) in each of the application's components are loaded in order from the lowest priority component to the highest priority component. Each configuration can replace elements of the the configuration that has been loaded before, so the last configuration is the one that has the most control. The actual configuration process is a little more complex than this simplified explanation and is explained in full in "Configuring Renderers" on page 65. Crucially, the configuration defined in the application is given more weight than that defined in the CDEJ, so it is possible for the developer to customize anything. However, there are limits on what customizations are supported within the Cúram application and that are described at the relevant points in this guide.

When a custom widget controls most of the page content, it is often the case that much of the output of the widget relates to laying out other page content in the correct manner. The view-renderer and edit-renderer plug-in types that are associated with domain definitions are used to renderer fields that are bound to data. However, page layout is often unrelated to any data. Another type of plug-in, the *component-renderer*, can be used to perform these layout operations. These plug-ins are associated with *styles*, not domain definitions, and can be invoked by the domain-specific renderers when necessary. Styles and component-renderer plug-ins are covered in "Tying Widgets Together in a Cascade" on page 32.

An E-Mail Address Widget

Objective

To learn how to write a simple widget to present some data more appealingly in the context of a simple UIM page.

Prerequisites

A knowledge of UIM and Java development.

Introduction

The presentation requirements of many pages can be satisfied with simple UIM pages containing fields that are laid out using clusters and lists. However, the presentation of the data within a cluster or list might benefit by presenting it in a more aesthetically pleasing way. This chapter will show how the an e-mail address can be enhanced instead of presenting it as plain text. A link will be added to allow the user to click the address and open their e-mail software and also an icon will be added.

Defining the HTML

By default, string values are presented in the Cúram application, such as e-mail addresses, without any HTML mark-up. The string value is simply added to the HTML page in the appropriate location. The e-mail address widget must produce HTML in the following form for an e-mail address such as `info@example.com`:

```
<span class="email-container">
  <a href="mailto:info@example.com">
    <img src="../Images/email_icon.png"/>
    info@example.com
  </a>
</span>
```

Figure 1. HTML Output of the E-Mail Address Widget

The HTML above is formatted for clarity, but it will be generated without any indentation or line breaks, as these are not necessary for the browser to present the e-mail address properly and only increase the size of the page.

A `span` element specifying a custom CSS class name contains a hyperlink defined by the `a` (anchor) element. The anchor element's `href` attribute prefixes the e-mail address with `mailto:`, as most browsers will react to that value by opening the system's default e-mail application and creating a new message with that address in the **To:** field. The anchor element contains an `img` element for the e-mail icon and the e-mail address text that will be displayed for the user to click.

```
.email-container img {
  vertical-align: middle;
}
```

Figure 2. Custom CSS for the E-Mail Address Widget

The CSS vertical-align style applies only to the `img` element. It ensures that the e-mail address text shown to the user lines up with the centerline of the text, rather than the baseline. This looks more appealing. The same styling goal could be achieved if the `class` attribute were placed on the `img` element instead of the `span` element. However, placing the `email-container` class name on the `span` element allows further customization of the other elements using different CSS selectors without the need to change the HTML structure generated by the widget, which would involve changing and rebuilding the Java source code.

The *Cúram Web Client Reference Manual* provides more details on adding custom CSS resources to the application.

Defining the Renderer Class

The Cúram Renderer API defines the `DomainRenderer` interface that is used when writing renderer plug-in classes, such as for the e-mail address widget. A plug-in class has a `render` method that is provided with details of the field to be rendered and the method must retrieve the data bound to that field and add the HTML mark-up to that data.

The developer must not implement the `DomainRenderer` interface directly. Instead, the OOTB application provides abstract base classes that the developer must use as the base of any custom renderer plug-in class. The e-mail address widget produces a read-only value, so it will be presented using a view-renderer plug-in based on the `AbstractViewRenderer` class. The developer should place the `EMailAddressViewRenderer.java` source file in the `sample` package sub-folder of the `javasource` folder of the client application component.

```
public class EMailAddressViewRenderer
       extends AbstractViewRenderer {

  public void render(
        Field field, DocumentFragment fragment,
        RendererContext context, RendererContract contract)
        throws ClientException, DataAccessException,
               PlugInException {
    // Create the HTML here....
  }
}
```

Figure 3. Declaration of the `EMailAddressViewRenderer` Class

A renderer plug-in class uses the *W3C DOM Level 3 Core* API to create the HTML content. This API is a standard component of the Java Runtime Environment for Java 5 and above. It is documented in the Javadoc supplied for the corresponding JDK. For further information about this API, refer to that documentation.

The first argument to the `render` method is a `Field` object that represents the details of the UIM `FIELD` element to be rendered and the data bound to it by its connections.

The second argument is a DOM `DocumentFragment` node. The goal of the render method is to append DOM nodes representing the data and its HTML mark-up to this fragment. The system will automatically serialize these nodes to HTML in string form and include this in the HTML stream for the page that is returned to the web browser.

The third argument is a `RendererContext` object. This object provides access to the context in which a renderer is invoked. It includes facilities to delegate rendering to other renderers, to resolve the data identified by the paths associated with a `Field` object, to include JavaScript resources in the page that can be shared with other renderers, and other facilities that are elaborated upon in the API documentation.

Use of the `RendererContract` argument to the render method is not supported except in the limited manner described later in this guide.

See the Cúram Javadoc for full details on each of these arguments and their interface types.

Accessing the Data

The `Field` object has a `Binding` property that defines the source path and target path that identify the data that is bound to the field. These paths combine the server interface name and the property name into a single value. The context provides a `DataAccessor` object that can be invoked to resolve paths to their values. For a view-renderer, only the source path is provided. The target path is only provided for edit-renderers (presented in "A Text Field Widget with No Auto-completion" on page 39). Paths can represent values other than server interface properties. The developer should not be concerned about where the data comes from, only that it can be retrieved when required. More information about the available paths and their forms is provided in "Accessing Data with Paths" on page 68. The code to retrieve the e-mail address string value is shown below.

```
String emailAddress = context.getDataAccessor()
       .get(field.getBinding().getSourcePath());
```

Figure 4. Getting the E-Mail Address Value

The source path is retrieved from the field's binding and passed to the `get` method of the data accessor retrieved from the context. The source path will never be null for a view-renderer plug-in. The `get` method will return the value of the (in this case) server interface property. The value will be formatted to a string representation appropriate for the active user. This formatting is performed using the `format` method of the `DomainConverter` plug-in associated with the domain of the server interface property. While the formatting of an e-mail address value is trivial (the value is simply returned as is), other values, such as dates and date-times must be formatted using the active user's locale, time zone and date format. Regardless of the type of the underlying data, this will all be handled automatically by the converter plug-ins. The returned string will be suitable for inclusion in the HTML response without any further formatting. See the *Cúram Web Client Reference Manual* for more information on converter plug-ins and their `format` methods.

Generating the HTML Content

With the e-mail address retrieved, it must now be marked up with the required HTML. The DOM API, while a little verbose, makes this process easy and reduces the chances of producing invalid output. The use of the DOM API means that opening and closing tags for the elements will be created as needed and the attribute values and body content will be escaped automatically.

All content created using the DOM API must be created in the context of the owning DOM `Document`. Each node has a property that identifies this `Document` object, so it can be retrieved from the document fragment. Elements and other nodes can be created using the factory methods of the Document object. The nodes can be appended to each other, and ultimately to the provided document fragment, to create the correct HTML structure. This is shown below (see "Source Code for the E-Mail Address Widget" on page 78 for the complete source code of this renderer).

```
Document doc = fragment.getOwnerDocument();

    Element span = doc.createElement("span");
    span.setAttribute("class", "email-container");
    fragment.appendChild(span);

    Element anchor = doc.createElement("a");
    anchor.setAttribute("href", "mailto:" + emailAddress);
    span.appendChild(anchor);

    Element img = doc.createElement("img");
    img.setAttribute("src", "../Images/email_icon.png");
    anchor.appendChild(img);

    anchor.appendChild(doc.createTextNode(emailAddress));
```

Figure 5. Marking Up the E-Mail Address Value

The first line gets the owner document that is used throughout the rest of the method to create new nodes. The `span` element is then created and added to the document fragment. The other elements and nodes are created and added in turn. When the render method returns, the system takes the newly populated document fragment and incorporates its contents into the HTML page in the appropriate location.

The URI of Cúram application pages includes the locale code as the first part of the resource path, for example, `en/Person_homePage.do`. This path is relative to the application's *context root*, which corresponds to the `WebContent` folder in the development environment. When icons or other resources are referenced, the `../` path prefix is needed for relative URIs so move from the locale-specific folder in the page's URI, back to the context root folder. More details about the inclusion of custom image resources can be found in the *Cúram Web Client Reference Manual*.

Configuring the Widget

To configure the e-mail address widget, the data must be in a domain that is specific to e-mail addresses. Here, the SAMPLE_EMAIL_ADDR domain is assumed. The `DomainsConfig.xml` file should be added to the client application component, or the existing file should be modified if it already exists, to associate the view-renderer plug-in class with that domain.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dc:domains

  <dc:domain name="SAMPLE_EMAIL_ADDR">
    <dc:plug-in name="view-renderer"
                class="sample.EMailAddressViewRenderer"/>
  </dc:domain>

</dc:domains>
```

Figure 6. Configuring the E-Mail Address Widget

Applying the above configuration, the view-renderer of the custom widget will now be invoked anywhere a UIM `FIELD` element has a source connection to a server interface property in the SAMPLE_EMAIL_ADDR domain. If the UIM `FIELD` has a target connection, the edit-renderer will be used instead. As no edit renderer is defined in this configuration, the edit-renderer of the parent or other ancestor domain, will be inherited and used. Typically, this will be the `TextEditRenderer` associated by default with the SVR_STRING domain.

More information about configuring renderers and other plug-ins is provided in “Configuring Renderers” on page 65.

The Sample Context Panel Widgets

Objective

To introduce the functionality of the sample context panel widgets that will be developed throughout the following chapters of this guide.

Prerequisites

An understanding of the basic process of developing custom widgets, as presented in the previous chapters.

Introduction

The previous chapter presented the main steps required to develop a simple custom widget and the artifacts required for its operation. Simple custom widgets, such as the e-mail address widget, are often sufficient to meet presentation requirements. They can also be used in the context of more complex widgets. In this chapter, two such complex widgets will be introduced. The following chapters will develop these sample widgets in full to demonstrate all of the main concepts in advanced custom widget development.

The two sample widgets are used to present information in context panels. To avoid overloading the developer with information, the main parts of these context panel widgets will be developed first in isolation. Each part will be a widget in its own right and will be configured for use on its own before the next part is introduced. When the parts are essentially complete, they will be combined using new renderer classes that delegate the rendering of these parts to form the full sample widgets. Later chapters will then show how issues such as text localization, locale-specific data formatting and accessibility compliance can be addressed.

The Sample Widgets

The first sample widget is a context panel providing details about a person shown in “The Sample Widgets.” The widget has two parts: the first part presents a photograph of the person above their name, an icon provides a hyperlink from the photograph to the home page of that person; the second part displays details about that person using text with elaborate styling and icons. This development of this context panel will show how these two parts can be created and used independently and how they can also be combined together into a single widget. In the cases of both of these parts, the content and layout requirements cannot be met using ordinary UIM pages.

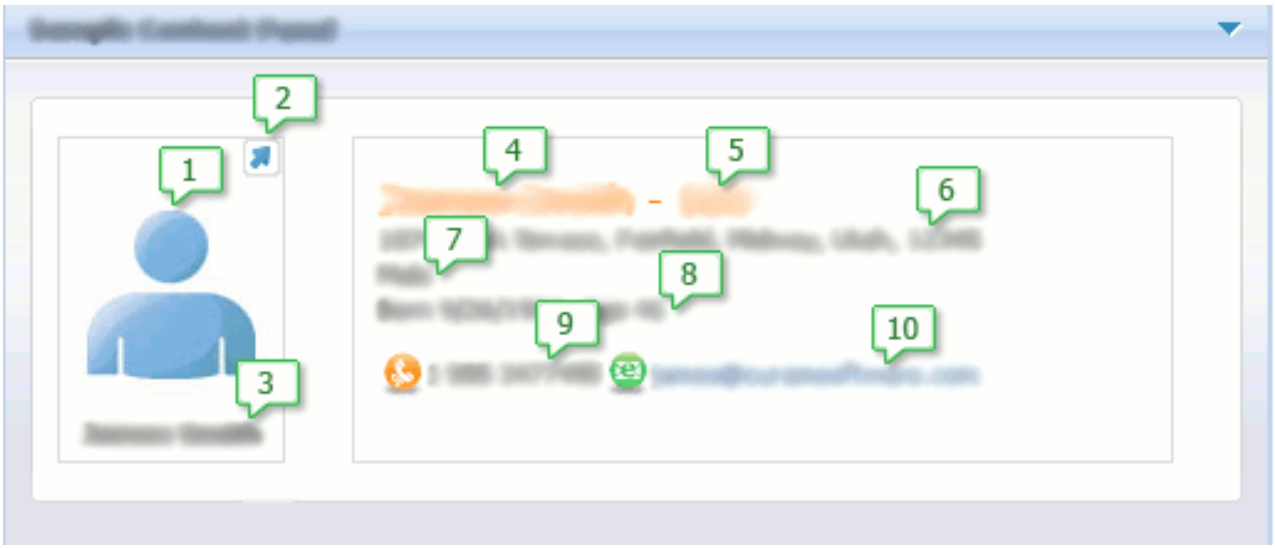

Figure 7. Context Panel for a Person

1. Photograph
2. Icon Links to Person's Home Page
3. Name
4. Name
5. ID
6. Address
7. Gender
8. Date Of Birth
9. Telephone Number
10. Email

The photograph widget will introduce XML-based data sources and the use of the `FileDownload` servlet to deliver images to the web browser. The details widget will, at first, demonstrate a more complex example of a widget backed by XML data. Later, the details widget will be used to show how the e-mail address widget developed in "An E-Mail Address Widget" on page 13 can be reused through delegation to present the e-mail address value, how text can be localized and how locale-specific formatting can be applied to the date of birth value.

The second sample widget, a person list widget shown in "The Sample Widgets" on page 18, is another context panel widget. This widget displays a list of people using their photographs and, when each photograph is clicked, some details about that person are shown in a pop-up box. The photograph widget and details widget developed for the first sample are reused to create this new context panel widget. This time, however, the person's name is presented in a different way in the details panel and their ID number is omitted. This kind of reuse is more complex than the reuse of a simple e-mail address renderer.

Figure 8. Context Panel for a List of Case Participants

1. Photograph
2. Icon Links to Person's Home Page
3. Name
4. Name
5. Address
6. Gender
7. Date Of Birth
8. Telephone Number
9. Email

There are two fundamentally different ways to access data: as single values and as lists of values. The person list widget must handle a list of values that stored in an XML document. Widgets that are developed to handle a single value can, with a little care, be reused in the context of widgets that present lists of values. The reuse of the photograph and details widgets in a list context will demonstrate further complex rendering techniques.

A Photograph Widget

Objective

To show how to develop a widget that displays the photograph of a person in a context panel and to show to to access XML data.

Prerequisites

Familiarity with Java development and with the construction of web page content using CSS and HTML.

Introduction

Figure 9. Context Panel Showing a Photograph of a Person

1. Photograph
2. Link to the Associated Details Page
3. Name

The photograph widget displays a photograph of a person in the current context with their name and a link to an associated details page. “An E-Mail Address Widget” on page 13 described how to access a single source value (the e-mail address) and generate HTML markup to provide a more aesthetically pleasing representation of an e-mail address. The same principals apply here, except that multiple source values are required for the photo widget. The person's name is displayed as text and their unique identifier is required to retrieve their photograph as well as being needed as a parameter to link to the associated details page. This chapter will show how multiple source values can be combined and accessed by the widget.

This chapter will also show how to access a photograph. Photographs are typically stored in the database along with other details of the person. Photographs, like any other images, can be delivered to the web browser by using a HTML `img` element and setting its `src` attribute to the URI of the resource that can supply the image data. For images such as icons, the URI points to a static image file within the web client application. For photographs, the URI points to the Cúram `FileDownload` servlet and includes the necessary parameters to instruct that servlet to retrieve the image data from the database and return it to the web browser.

Defining the HTML

As shown by the screen-shot, the photograph widget displays a link, a photograph and the person's name one under the other. It is recommended that all widgets have a single root node with a specific CSS class. This makes the “boundaries” of the widget obvious. It is also the basis of making associated CSS rules as specific as possible to this widget. The “root” class is then used when when defining CSS rules for all content within the widget. In this case, the root `div` element has been given the `photo-container` class name. There are three child `div` elements

containing the link, the photo and the person name. Each of these has also been given a CSS class so that their contents can be individually styled. The `img` elements show how both a static and a dynamic image resource can be accessed. The dynamic image resource uses the Cúram `FileDownload` servlet. The use of this feature and the value of the `img` element's `src` attribute will be described in this chapter.

```
<div class="photo-container">
  <div class="details-link">
    <a href="Person_homePage.do?id=101>
      <img src="../Images/details-link-icon.png">
    </a>
  </div>
  <div class="photo">
    <img src="../servlet/FileDownload?
              pageID=Sample_photo&id=101">
  </div>
  <div class="description">
    James Smith
  </div>
</div>
```

Figure 10. HTML Output of the Photo Widget

The HTML above is formatted for clarity, but it will be generated without any indentation or line breaks, as these are not necessary for the browser to present the e-mail address properly and only increase the size of the page.

Based on the screen-shot, the visual requirements of the widget can be summarized as:

- The widget has a border.
- The link is right-aligned in the widget.
- The photograph and person name are center-aligned in the widget.

The class names applied in the HTML allow these requirements to be implemented in CSS as follows:

```
.photo-container {
  border: 1px solid #DADADA;
  width: 90px;
  height: 130px;
}

.photo-container .details-link {
  text-align: right;
}

.photo-container .photo {
  text-align: center;
}

.photo-container .description {
  text-align: center;
  font-weight: bold;
}
```

Figure 11. Custom CSS for the Photo Widget

The class name of the root `div` element is used when defining all CSS rules to ensure they are specific to this widget. The `photo-container` class applies a border and fixed width to the widget. The fixed width means an image with a max size of 88 pixels can be accommodated, allowing for the border. If the image width is less

than this maximum value, ensure it is an even number. Since the image is centrally aligned this ensures there is even spacing on each side of the image. The remaining CSS classes make use of the `text-align` CSS style to align the contents within each child `div` element. This is possible because the contents of each `div` element are "inline" elements i.e. an anchor element, an image element and plain text. Finally, there is an additional style on the `description` element to set it's font.

Defining Data in XML Form

The previous chapters described how simple data can be accessed by a renderer and marked up with HTML for presentation. For complex widgets, simple values like that are usually not sufficient. It is often preferable for the value to be an XML document that contains all of the data required for the widget in a structured form. In the case of this photograph widget, the concern role ID of the person and the name of the person are required to present the photograph correctly. As the widget is associated with a UIM FIELD element that can only specify one SOURCE connection to the required data, both the ID and the name must be passed back in a single server interface property. The Cúram application provides support classes that make it simple to access data expressed as an XML document, so an XML document containing the values is the preferred form when combining data into a single server interface property.

Below is a sample of an XML document that represents all of the information required to present the photograph of a person. The `id` element defines the concern role ID value passed to the `FileDownload` servlet using the *id* parameter shown in the example in the previous section. The `name` element defines the name of the person to be shown below the photograph. To make best use of the support classes provided with the Cúram application, the values should be given in the body of the elements, rather than as attributes of a single element. The XML document is constructed in a server facade and returned in a single (string-based) property.

```
<photo>
  <id>101</id>
  <name>James Smith</name>
</photo>
```

Figure 12. An XML Document Describing a Photograph

Defining the Renderer Class

The skeleton renderer class for the photograph widget is shown below. The class extends the same base class as the e-mail address widget, as it too is a view renderer. The class should be created in the `component/sample/javasource/sample` folder.

```
public class PhotoViewRenderer
       extends AbstractViewRenderer {

  public void render(
      Field field, DocumentFragment fragment,
      RendererContext context, RendererContract contract)
      throws ClientException, DataAccessException {
    // Add the HTML to the "fragment" object here....
  }
}
```

Figure 13. The Renderer Class for the Photograph Widget

Accessing Data in XML Form

"An E-Mail Address Widget" on page 13 described how to access a single source value using a `Field` object, its `Binding` property and a source path. For the photograph widget, the source value is no longer a simple string, instead it is an XML document. The approach used for the e-mail address widget needs to be extended to allow values embedded in the XML document to be retrieved individually. Support is provided for accessing data in an XML by *extending* the source path. The code to retrieve the person's name and unique identifier from the XML document is shown below.

```
String personID = context.getDataAccessor()
      .get(component.getBinding()
          .getSourcePath().extendPath("photo/id"));
    String personName = context.getDataAccessor()
        .get(component.getBinding()
            .getSourcePath().extendPath("photo/name"));
```

Figure 14. Getting the Person Name and ID Values

The source path is retrieved from the field's binding in the same way as the e-mail address widget in the previous chapter. However, the source path is not passed directly to the `get` method of the data accessor retrieved from the *`context`* . Doing this would simply return the entire XML document as a string. Instead the source path is first extended using the `extendPath` method. The path extensions are `photo/id` and `photo/name`. They correspond directly to the tree structure of the XML document. For example, the `photo/id` path means the data accessor will retrieve the body content of the `id` element which is a child of the `photo` element. In the sample XML above, this is the value "101". Those familiar with XPATH will recognize the format of these paths. However, while the extended paths used here are similar, they are *not* XPATH. Creating simple XML documents where each value is represent in the body content of an element will mean the path formats shown in this section will be all that is required to use in a widget. However, the "Extending Paths for XML Data Access" on page 73 appendix describes XML data access through path extension in full detail.

Generating the HTML Content

With the data for the photograph widget retrieved, it must now be marked up with the required HTML.

```
Document doc = fragment.getOwnerDocument();

    Element rootDiv = doc.createElement("div");
    rootDiv.setAttribute("class", "photo-container");
    fragment.appendChild(rootDiv);

    Element linkDiv = doc.createElement("div");
    linkDiv.setAttribute("class", "details-link");
    rootDiv.appendChild(linkDiv);

    Element anchor = doc.createElement("a");
    anchor.setAttribute("href", "Person_homePage.do?"
                              + "id=" + personID);
    linkDiv.appendChild(anchor);

    Element anchorImg = doc.createElement("img");
    anchorImg.setAttribute("src", "../Images/arrow_icon.png");
    anchor.appendChild(anchorImg);

    Element photoDiv = doc.createElement("div");
    photoDiv.setAttribute("class", "photo");
    rootDiv.appendChild(photoDiv);

    Element photo = doc.createElement("img");
    photo.setAttribute("src",
      "../servlet/FileDownload?"
      + "pageID=Sample_photo"
      + "&id=" + personID);
    photoDiv.appendChild(photo);

    Element descDiv = doc.createElement("div");
    descDiv.setAttribute("class", "description");
    descDiv.appendChild(doc.createTextNode(personName));
    rootDiv.appendChild(descDiv);
```

Figure 15. Marking Up the Photograph Data

The same techniques used to construct the e-mail address widget using the DOM API in the previous chapter, also apply here. The URI used to link to the details page, a static image and the `FileDownload` servlet are described below.

Linking to a UIM Page

The URI of Cúram application pages includes the locale code as the first part of the resource path, for example, `en/Person_homePage.do`. This path is relative to the application's *context root*, which corresponds to the `WebContent` folder in the development environment. All UIM pages are therefore considered to be in a locale "folder". When linking from one UIM page to another, it is always in the same locale (or "folder"). Therefore the locale should not be specified in the URI when generating a link. For example, in the sample code shown above, note that the `href` to link to the `Person_home` UIM page was generated *without* the locale specific folder specified:

```
anchor.setAttribute("href", "Person_homePage.do?"
                              + "id=" + personID);
```

Figure 16. Linking to a UIM Page

Linking to a Static Image

Linking to a static image was described when creating the e-mail address widget in the previous chapter, but is worth repeating here. Static images are stored in the folder `Images` which is located directly under the application's context root. Because a UIM page is in a locale specific folder, when icons or other resources are referenced the `../` path prefix is needed for relative URIs. This is to move from the

locale-specific folder in the page's URI, back to the context root folder as shown in this excerpt from the sample code:

```
anchorImg.setAttribute("src", "../Images/arrow-icon.png");
```

Figure 17. Linking to a Static Image

Linking to the `FileDownload` Servlet

The `FileDownload` servlet is used to download an image resource from the Cúram database. The path to the file download servlet is `servlet/FileDownload` which is relative to the application's context root. The `../` path prefix is also needed to move from the locale-specific folder as shown in this excerpt from the sample code:

```
photo.setAttribute("src",
      "../servlet/FileDownload?"
      + "pageID=Sample_photo"
      + "&id=" + personID);
```

Figure 18. Linking to the `FileDownload` Servlet

The `FileDownload` servlet has to be configured to use the parameters shown in the URI above to download the correct photograph. This is described in detail in later in this chapter.

Configuring the Widget

To configure the photograph widget, the data must be in a domain that is specific to photographs. Here, the SAMPLE_PHOTO_XML domain is assumed. The `DomainsConfig.xml` file should be added to the client application component, or the existing file should be modified if it already exists, to associate the view-renderer plug-in class with that domain. To access data in XML form and use the path extension feature described earlier a "marshal" plug-in *must* also be configured *exactly* as shown below. Failure to do so will mean that individual values cannot be retrieved from the XML document as shown earlier.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dc:domains
  <dc:domain name="SAMPLE_PHOTO_XML">
    <dc:plug-in
      name="view-renderer"
      class="sample.PhotoViewRenderer"
      />
    <dc:plug-in
      name="marshal"
      class="curam.util.client.domain.marshal.SimpleXPathMarshal"
      />
  </dc:domain>

</dc:domains>
```

Figure 19. Configuring the E-Mail Address Widget

Applying the above configuration, the view-renderer of the custom widget will now be invoked anywhere a UIM `FIELD` element has a source connection to a server interface property in the SAMPLE_PHOTO_XML domain. If the UIM `FIELD` has a target connection, the edit-renderer will be used instead. As no edit renderer is defined in this configuration, the edit-renderer of the parent or other ancestor domain, will be inherited and used. Typically, this will be the `TextEditRenderer` associated by default with the SVR_STRING domain. However, this type of widget is displaying a subset of the information the Cúram application captures about a

person. An editable version of this widget would not be expected. Instead the information would edited through the standard Cúram screens associated with a person, for example if the person's name required updating.

More information about configuring renderers and other plug-ins is provided in "Configuring Renderers" on page 65.

Configuring the FileDownload Servlet

The *Cúram Web Client Reference Manual* provides full information on the configuration of the FileDownload servlet for the use of the FILE_DOWNLOAD WIDGET in a UIM page. For this photograph widget, the same configuration is used, but instead of letting the UIM WIDGET element generate HTML anchor tag that downloads the photograph when clicked, the photograph widget will create a HTML image tag using the same URI that displays the image within the page. The example below is representative of the FileDownload configuration that is required in curam-config.xml:

```
<APP_CONFIG>

  <FILE_DOWNLOAD_CONFIG>
    <FILE_DOWNLOAD PAGE_ID="Sample_photo"
        CLASS="sample.interfaces.SamplePkg.Sample_readImage_TH">
      <INPUT PAGE_PARAM="id" PROPERTY="key$concernRoleID"/>
      <FILE_NAME PROPERTY="key$concernRoleID"/>
      <FILE_DATA PROPERTY="result$concernRoleImageBlob"/>
    </FILE_DOWNLOAD>
  </FILE_DOWNLOAD_CONFIG>

</APP_CONFIG>
```

Figure 20. Example FileDownload Configuration for a Photograph

Each file download configuration is uniquely represented by the PAGE_ID of the FILE_DOWNLOAD element. The PAGE_ID is used when a file download is initiated directly from a UIM page by using the FILE_DOWNLOAD WIDGET. However, as the file download link is being generated by a custom widget, the only requirement is that the PAGE_ID value is unique, it does not have to correspond to an existing UIM page. The widget will use this value when generating the URI to the FileDownload servlet. The remaining configuration elements and attributes define the server facade to invoke and it's inputs and outputs. Consult the *Cúram Web Client Reference Manual* for information on the configuration of the FileDownload servlet

```
<img src="../servlet/FileDownload?pageID=Sample_photo&id=101"/>
```

Figure 21. Example of the HTML to Show an In-line Image

The HTML for the image element should look like the example above. The src attribute path is made up of a number of parts. The fixed path to Cúram's file download servlet is: ../servlet/FileDownload. The pageID request parameter is mandatory and must correspond to the PAGE_ID of the FILE_DOWNLOAD configuration element. The id request parameter corresponds to the INPUT configuration element. With this URI, the FileDownload servlet reads the configuration, sets the input fields of the server facade, invokes the facade and retrieves it's output fields which contain the file name and binary file data.

A Details Widget Demonstrating Widget Re-use

Objective

To show how to develop a widget that presents the details of a Person using formatting not possible on a plain UIM page. To show how to re-use the e-mail address widget described earlier.

Prerequisites

The previous chapters in this document.

Introduction

The presentation requirements of many pages can be satisfied with simple UIM pages containing fields that are laid out using clusters and lists. However, the presentation of this details widget requires additional processing such as displaying the person's name and reference number in a different font, refer to "The Sample Widgets" on page 18. Also, the e-mail address is presented in the same form as shown in "An E-Mail Address Widget" on page 13. This widget will be re-used within the details widget.

Defining the HTML

In the details widget, there are a number of lines of plain text displaying the the person's address, date of birth and so on. The person's name, reference number and contact details have specific presentation requirements and which means they need to be distinguished in the HTML so that specific CSS rules can be applied to them. The following HTML structure for the details widget achieves this:

```
<div class="person-details-container">
  <div class="header-info">James Smith - 24684</div>
  <div>1074, Park Terrace, Fairfield,
  Midway, Utah, 12345</div>
  <div>Male</div>
  <div>Born 9/26/1964, Age 46</div>
  <div class="contact-info">
    <img src="../Images/phone_icon.png">1 555 3477455
    <span class="email-container">
      <a href="mailto:info@example.com">
        <img src="../Images/email_icon.png"/>
        info@example.com
      </a>
    </span>
  </div>
</div>
```

Figure 22. HTML Output of the Details Widget

The HTML above is formatted for clarity, but it will be generated without any indentation or line breaks, as these are not necessary for the browser to present the e-mail address properly and only increase the size of the page.

It is good practice to give a widget a single root node with a specific CSS class. It is the basis of making CSS rules as specific as possible to this widget. The "root" class is used when when defining CSS rules for all content within the widget. The root `div` element has been given the `person-details-container` class name. Each line of text in the details panel is represented by a `div` element. Additionally, two `div` elements have CSS class names so that specific CSS rules can be applied to them. Note that the HTML representing the e-mail address is identical to that

described in "An E-Mail Address Widget" on page 13.

```
.person-details-container .header-info {
  color: #FB7803;
  font-size: 140%;
}
.person-details-container .contact-info img {
  vertical-align: middle;
}
```

Figure 23. Custom CSS for the Details Widget

The `header-info` and `contact-info` classes allow the specific presentation requirements (e.g. changing the font) to be implemented. Note that the CSS rules are made as specific as possible by using the `person-details-container` class name in every rule.

Defining Data in XML Form

The photograph widget required an XML document to provide all of the data required by the renderer class. The details widget also requires an XML document for the same reasons. The general structure of the documents is the same: a root element containing one child element for each value, where each value is the body content of the child element.

```
<details>
  <name>James Smith</name>
  <reference>24684</reference>
  <address>1074, Park Terrace, Fairfield,
  Midway, Utah, 12345</address>
  <gender>Male</gender>
  <dob>9/26/1964</dob>
  <age>46</age>
  <phone>1 555 3477455</phone>
  <e-mail>james@ie.ibm.com</e-mail>
</details>
```

Figure 24. An XML Document Describing a Person

The XML above is formatted for clarity, the indentation or line breaks are not required.

Defining the Renderer Class

The skeleton renderer class for the details widget is shown below. The class extends the same base class as the e-mail address widget and the photograph widget, as it too is a view renderer. The class should be created in the `component/sample/javasource/sample` folder.

```
public class PersonDetailsViewRenderer
       extends AbstractViewRenderer {

  public void render(
      Field field, DocumentFragment fragment,
      RendererContext context, RendererContract contract)
      throws ClientException, DataAccessException {
    // Add the HTML to the "fragment" object here....
  }
}
```

Figure 25. The Renderer Class for the Details Widget

Accessing Data in XML Form

Data from the XML document is accessed in the same way as the photograph widget described in the previous chapter. The source path is extended to extract an individual value. For example, `/details/name` retrieves the person's name.

```
String name = context.getDataAccessor().get(
        field.getBinding().getSourcePath()
                          .extendPath("/details/name"));
    String reference = context.getDataAccessor().get(
        field.getBinding().getSourcePath()
                          .extendPath("/details/reference"));
```

Figure 26. Getting the Person name and Reference Number

All values in the XML document can be accessed using the same technique except for the e-mail address value. The e-mail address widget described in "An E-Mail Address Widget" on page 13 will be re-used to output the e-mail address. As shown in that chapter, the e-mail address widget uses a `Field` object, its `Binding` property and a source path to access the e-mail address value. The next section will explain how to invoke that renderer.

Generating the HTML Content

The same technique, described in previous chapters, of using the DOM API to generate HTML content can be used to output the HTML show earlier in this chapter. The only new concept comes at the point when the HTML for the e-mail address is to be output. The e-mail address widget will be re-used within the details widget to output the HTML required for an e-mail address.

The `render` method of a widget is usually invoked by directly by the Cúram infrastructure. The parameters provided to the `render` method are set based on what was specified in UIM. For example, the source path of the `Field` object's `Binding` is set based on `CONNECT` and `SOURCE` element's used within a `FIELD` element. To invoke one widget from another it becomes the developer's responsibility to ensure the appropriate widget is invoked and the correct parameters are supplied to it. The code required to do this is as follows:

```
FieldBuilder fb =
      ComponentBuilderFactory.createFieldBuilder();
    fb.setDomain(
      context.getDomain("SAMPLE_EMAIL"));
    fb.setSourcePath(
      field.getBinding().getSourcePath()
                        .extendPath("/details/e-mail"));
    DocumentFragment emailFragment = doc.createDocumentFragment();
    context.render(fb.getComponent(), emailFragment,
                   contract.createSubcontract());
    div.appendChild(emailFragment);
```

Figure 27. Invoking the E-Mail Address Widget from the Details Widget

The steps to invoke the e-mail address widget are:

1. Create a `Field` component.

   A `FieldBuilder` is required to create a `Field`. The `ComponentBuilderFactory` can be used to create a `FieldBuilder` as shown above. See "Overview of the Renderer Component Model" on page 50 for full details.
2. Set the domain of the `Field`.

   The underlying domain definition of a `Field` is used to select the appropriate widget. "An E-Mail Address Widget" on page 13 showed how the e-mail

address widget was associated with the SAMPLE_EMAIL domain definition. This domain definition is set on the `Field` as shown above.

3. Set the source path of the `Field`.

   "An E-Mail Address Widget" on page 13 chapter showed how the e-mail address widget used it's source path to access the value of the e-mail address. This is normally set based on the `CONNECT` in UIM. In this case the source path for the widget has to be specified "manually". The details widget has to tell the e-mail address widget where to get its data from. As shown earlier the e-mail address is embedded in the XML document supplied to the details widget. The path extension technique to access XML data, that has been described in previous chapters, can be used to specify the source path for the e-mail address widget.

   The `setSourcePath` method of the `FieldBuilder` is used to set the source path as shown in the following excerpt from the example above. The source path is the same as used to access other values from the XML document. The difference is that instead of retrieving the value directly in the details widget, it is set as the source path of the e-mail address widget.

   ```
   fb.setSourcePath(
       field.getBinding().getSourcePath()
                         .extendPath("/details/e-mail"));
   ```

   This demonstrates the benefits of the path system to access data. In "An E-Mail Address Widget" on page 13, the e-mail address was retrieved directly from a server interface property. In this chapter the e-mail address is retrieved from an XML document. However the e-mail address widget is identical in both cases. It retrieves its data using a source path and is abstracted from what source path actually resolves to "behind the scenes".

4. Create a `DocumentFragment` for the widget content

   As shown in previous chapters, he DOM API has been used to create HTML elements and add them to a `DocumentFragment`, supplied as the `fragment` parameter to the `render` method. The `DocumentFragment` is usually supplied by the Cúram infrastructure. In this case the fragment has to be created using the `createDocumentFragment` as shown above.

5. Invoke the e-mail address widget

   The e-mail address widget is invoked by calling `context.render`. The first parameter to the method is a `Field`. The `FieldBuilder` was used to set the domain and source path and the `Field` is retrieved by calling the `getComponent` method. The second parameter is the `DocumentFragment` created earlier. The widget will add it's HTML content to this fragment. The final parameter is reserved and should always be set to `contract.createSubcontract()`.

6. Append HTML generated from e-mail address widget

   After the e-mail address widget has been invoked, the `DocumentFragment` will contain its HTML content. This fragment can be added to the appropriate place in the details widget. In the HTML described earlier the HTML should be added as a child of the `div` element with the `contact-info` CSS class.

The first three steps above build up a "component model", in this case a single `Field`. The remaining steps then *render* the model as HTML. The "Overview of the Renderer Component Model" on page 50 appendix provides more details on the classes and APIs which can be used to build a "component model".

Configuring the Widget

To configure the details widget, the data must be in a domain that is specific to person details. Here, the SAMPLE_DTLS_XML domain is assumed. The

DomainsConfig.xml file should be added to the client application component, or the existing file should be modified if it already exists, to associate the view-renderer plug-in class with that domain. To access data in XML form and use the path extension feature described earlier a "marshal" plug-in *must* also be configured *exactly* as shown below. Failure to do so will mean that individual values cannot be retrieved from the XML document as shown earlier.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dc:domains
  <dc:domain name="SAMPLE_DTLS_XML">
    <dc:plug-in
      name="view-renderer"
      class="sample.PersonDetailsViewRenderer"
      />
    <dc:plug-in
      name="marshal"
      class="curam.util.client.domain.marshal.SimpleXPathMarshal"
      />
  </dc:domain>

</dc:domains>
```

Figure 28. Configuring the Person Details Widget

Applying the above configuration, the view-renderer of the custom widget will now be invoked anywhere a UIM FIELD element has a source connection to a server interface property in the SAMPLE_EMAIL_ADDR domain. If the UIM FIELD has a target connection, the edit-renderer will be used instead. As no edit renderer is defined in this configuration, the edit-renderer of the parent or other ancestor domain, will be inherited and used. Typically, this will be the TextEditRenderer associated by default with the SVR_STRING domain. However, this type of widget is displaying a subset of the information the application captures about a person. An editable version of this widget would not be expected. Instead the information would edited through the standard Cúram screens associated with a person, for example if the person's name required updating.

More information about configuring renderers and other plug-ins is provided in "Configuring Renderers" on page 65.

Tying Widgets Together in a Cascade

Objective

To expand on the concepts of widget re-use and delegation. To show how to build generic widgets using the Component and Container interfaces.

Prerequisites

The previous chapters in this document.

Introduction

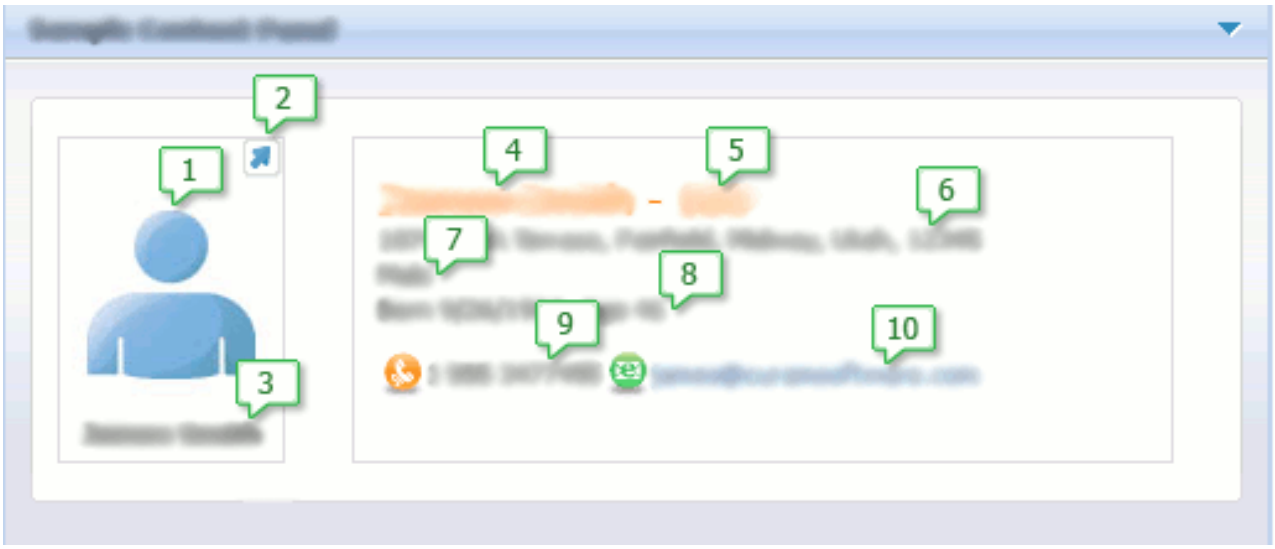

Figure 29. Context Panel Showing the Photograph and Details of a Person

1. Photograph
2. Icon Links to Person's Home Page
3. Name
4. Name
5. ID
6. Address
7. Gender
8. Date Of Birth
9. Telephone Number
10. Email

This chapter will expand on the re-use of widgets to produce the “Person Context Panel Widget”. As shown above, this is a combination of the photograph widget and details widget positioned side-by-side. The previous chapter introduced widget re-use by showing how the details widget could delegate to the e-mail address widget to generate part of it's HTML content. Using the exact same technique, the “Person Context Panel Widget” could combine the output of the photograph and details widgets and display them side-by-side to produce the content shown above. However, there is an opportunity to provide a further layer of abstraction by introducing a generic widget for displaying content side-by-side in a horizontal layout. The generic requirement could be phrased as: “To combine the output of multiple widgets in a horizontal layout”.

The previous chapter introduced the concepts of building a "component model" and delegating to another widget to render it as HTML. The details widget was responsible for building the component model, which consisted of a single `Field`. The model was then passed to the e-mail address widget to generate HTML. In the same way the “Person Context Panel Widget” will be responsible for building the component model. In this case the component model will be represented as a collection of `Field` 's; one for the photograph, the other for the person's details. The “Person Context Panel Widget” will pass the component model to a new widget,

the “Horizontal Layout Widget”. This widget in turn will delegate to photograph and details widgets introduced in previous chapters and combine their output. The advantage of this abstraction is the “Horizontal Layout Widget” could also be used to fulfill separate requirements such as combine the display of multiple details widgets or multiple photograph widgets in a horizontal layout. For example, consider the requirement to display the photographs of a family side-by-side.

In summary, by the end of this chapter the “Person Context Panel Widget” will delegate to the “Horizontal Layout Widget”, which in turn will delegate to the widgets introduced in earlier chapters. This is what is referred to as a “cascade”.

Defining Data in XML Form

The XML document for the “Person Context Panel Widget” widget is a combination of the XML documents used by the photograph and details widgets described in previous chapters, but combined in a new root element. This will allow each of those renderers to be re-used.

```
<person>
  <photo>
    <name>James Smith</name>
    <id>24684</id>
  </photo>

  <details>
    <name>James Smith</name>
    <reference>24684</reference>
    <address>1074, Park Terrace, Fairfield,
    Midway, Utah, 12345</address>
    <gender>Male</gender>
    <dob>9/26/1964</dob>
    <age>46</age>
    <phone>1 555 3477455</phone>
    <e-mail>james@ie.ibm.com</e-mail>
  </details>
</person>
```

Figure 30. An XML Document Describing a Person

Defining the HTML

The HTML of the “Person Context Panel Widget” is the output of the photograph and details widgets combined by placing them in the cells of a HTML table to lay them out horizontally.

```
<table class="sample-container">
  <tbody>
    <tr>
      <td>
        <!-- HTML of photograph widget goes here -->
      </td>
      <td>
        <!-- HTML of details widget goes here -->
      </td>
    </tr>
  </tbody>
</table>
```

Figure 31. HTML Output of the Person Context Panel Widget

The CSS class `sample-container` is unused in this example, but it is still a good practice to always provide a CSS class on the root element of a widget to allow for customization of the contents within it. For example, the root element of the

photograph widget has a CSS class of photo-container. If necessary, the photograph widget could be customized specifically when it is contained within the table shown above as follows:

```
.sample-container .photo-container {
/* customization of photograph widget styles */
}
```

Defining the Renderer Classes

Two classes are required; one for the "Person Context Panel Widget", the other for the "Horizontal Layout Widget". The skeleton renderer class for the "Person Context Panel Widget" is shown below. The class extends the same base class as the previous widgets, as it too is a view renderer. The class should be created in the component/sample/javasource/sample folder.

```
public class PersonContextPanelViewRenderer
       extends AbstractViewRenderer {

  public void render(
      Field field, DocumentFragment fragment,
      RendererContext context, RendererContract contract)
      throws ClientException, DataAccessException {
    // Add the HTML to the "fragment" object here....
  }
}
```

Figure 32. The Renderer Class for the "Person Context Panel Widget"

The skeleton renderer class for the generic "Horizontal Layout Widget" is shown below. The widgets described up to now in this guide have been "view renderer's" based on the AbstractViewRenderer class. The component model provided to each widget was a single Field (the first parameter of its render method). As described in the introduction above, "Horizontal Layout Widget" requires a collection of Field 's. This requires the use of a new base class and in turn, a different signature for the render method. Instead of a Field, a Component is provided to the render method. With the use of a new base class, this renderer class is known as a "component renderer" instead of a "view renderer". The class should be created in the component/sample/javasource/sample folder.

```
public class HorizontalLayoutRenderer
       extends AbstractComponentRenderer {

  public void render(
      Component component, DocumentFragment fragment,
      RendererContext context, RendererContract contract)
      throws ClientException, DataAccessException {
    // Add the HTML to the "fragment" object here....
  }
}
```

Figure 33. The Renderer Class for the "Horizontal Layout Widget"

Generating the HTML Content

Person Context Panel Widget

The role of the "Person Context Panel Widget" is to build the component model and delegate to the "Horizontal Layout Widget" to render the HTML from the model. The component model is a collection of Field 's. As described in the previous section, the render method of the "Horizontal Layout Widget" expects a Component as it's first parameter. The out-of-the-box Cúram application provides a subclass of Component called Container, which is specifically for creating collections

of Component 's or Field 's.

```
ContainerBuilder cb
      = ComponentBuilderFactory.createContainerBuilder();
    cb.setStyle(context.getStyle("horizontal-layout"));

    FieldBuilder fb
      = ComponentBuilderFactory.createFieldBuilder();
    fb.copy(component);
    fb.setDomain(context.getDomain("SAMPLE_PHOTO_XML"));
    fb.setSourcePath(
      component.getBinding().getSourcePath()
               .extendPath("person"));
    cb.add(fb.getComponent());

    fb.setDomain(context.getDomain("SAMPLE_DTLS_XML"));
    fb.setSourcePath(
      component.getBinding().getSourcePath()
               .extendPath("person"));

    cb.add(fb.getComponent());
    DocumentFragment content
      = fragment.getOwnerDocument().createDocumentFragment();
    context.render(cb.getComponent(), content,
                   contract.createSubcontract());
    fragment.appendChild(content);
```

Figure 34. Building the component model and invoking the "Horizontal Layout Widget"

The steps to build the model and invoke the "Horizontal Layout Widget" are:

1. Create a Container component.

   A ContainerBuilder is required to create a Container. The ComponentBuilderFactory can be used to create a ContainerBuilder as shown above. See "Overview of the Renderer Component Model" on page 50 for full details.

2. Set the "style" of the Container.

   The "Horizontal Layout Widget" is a component renderer which is associated with a "style". The "Horizontal Layout Widget" has been associated with the horizontal-layout style. This must be set using the setStyle method as shown above. The style corresponds to a particular renderer implementation class. Configuration of this "style" is described later in this chapter and more detail on the component model and configuring renderers can be found in the appendices (note it is *not* a CSS style that is being referred).

3. Create a Field representing the photograph and add it to the container.

   As shown in the previous chapter, a Field is created using a FieldBuilder. Setting the domain definition to SAMPLE_PHOTO_XML ensures the photograph widget will be invoked. The next step is to set it's source path. The photograph XML is now embedded in an XML document with a root element called person which is supplied to the "Person Context Panel Widget". "A Photograph Widget" on page 20 showed how data for the photo widget was accessed in the XML document using paths such as photo/name. The full path to get the same data is now /person/photo/name. The photograph widget cannot be changed. Instead the source path is extended as shown above to account for the root person element. When the photograph widget executes, the paths will be combined to ensure the full path corresponding to the combined document is used. The Field is created using the getComponent method and added to the Container

4. Create a Field representing the person details and add it to the container.

In the same way as the previous point, a `Field` is created. Its domain definition is set to `SAMPLE_DTLS_XML` to associate it with the details widget. The source path is extended in the same to account for the root `person` element. The `Field` is created using the `getComponent` method and added to the `Container`.

5. Create a `DocumentFragment` for the widget content

   As shown in previous chapters, he DOM API has been used to create HTML elements and add them to a `DocumentFragment`, supplied as the `fragment` parameter to the `render` method. The `DocumentFragment` is usually supplied by the Cúram infrastructure. In this case the fragment has to be created using the `createDocumentFragment` as shown above.

6. Invoke the horizontal layout widget

   The e-mail address widget is invoked by calling `context.render`. The first parameter to the method is a `Field`. The `FieldBuilder` was used to set the domain and source path and the `Field` is retrieved by calling the `getComponent` method. The second parameter is the `DocumentFragment` created earlier. The widget will add it's HTML content to this fragment. The final parameter is reserved and should always be set to `contract.createSubcontract()`.

7. Append HTML generated from horizontal layout widget

   After the e-mail address widget has been invoked, the `DocumentFragment` will contain its HTML content. This fragment can be added to the appropriate place in the details widget. In the HTML described earlier the HTML should be added as a child of the `div` element with the `contact-info` CSS class.

The next section shows how the "Horizontal Layout Widget" renders the component model has HTML.

Horizontal Layout Widget

The component model supplied to the "Horizontal Layout Widget" is a collection of components. The role of this widget is to iterate over that collection, delegating to the widget associated with each component and combining the output into the HTML shown earlier.

```
Document doc = fragment.getOwnerDocument();
    Element table = doc.createElement("table");
    table.setAttribute("class", "sample-container");
    fragment.appendChild(table);

    Element tableBody = doc.createElement("tbody");
    table.appendChild(tableBody);

    Element tableRow = doc.createElement("tr");
    tableBody.appendChild(tableRow);

    Container container = (Container) component;
    for (Component child : container.getComponents()) {
      Element tableCell = doc.createElement("td");
      tableRow.appendChild(tableCell);
      DocumentFragment cellContent
        = doc.createDocumentFragment();
      context.render(child, cellContent,
                     contract.createSubcontract());
      tableCell.appendChild(cellContent);
    }
```

Figure 35. Generating a HTML table and delegating to other widgets

As in all previous examples, the DOM API is used to generate HTML elements. As shown in the previous section, the component model is represented by a `Container`, the `render` method signature requires a `Component`. As former is a

sub-class of the latter, a cast is required to a `Container`. A for loop is used to iterate over each item in the collection using the `getComponents` method. Each iteration of the for loop will:

1. Create a table cell and add it to the table row.
2. Create a `DocumentFragment` used when delegating to another widget.
3. Invoke another widget by calling `context.render` passing the current component in the collection and the fragment (the third parameter is unused and must always be set as shown above).
4. Appends the output from the widget to the table cell.

The requirement of this widget was described in the introduction as: "To combine the output of multiple widgets in a horizontal layout". This widget achieves the horizontal layout requirement by generating a HTML table. However, note that it is completely abstracted from the underlying details of the components it is outputting. It is simply iterating over a collection of components and delegating to their associated widgets. In this particular example the components represent a photograph and person details panel. However, without any modification, the widget could display multiple photographs side-by-side if the component model supplied to it was constructed accordingly.

Configuring the Widgets

Person Context Panel Widget

The configuration of this widget is identical to all previous examples. It has to be associated with a domain definition, SAMPLE_PERSON_XML is used. To allow access to values embedded in XML documents a "marshal" plug-in *must* also be configured *exactly* as shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dc:domains
  <dc:domain name="SAMPLE_PERSON_XML">
    <dc:plug-in
      name="view-renderer"
      class="sample.PersonContextPanelViewRenderer"
      />
    <dc:plug-in
      name="marshal"
      class="curam.util.client.domain.marshal.SimpleXPathMarshal"
      />
  </dc:domain>

</dc:domains>
```

Figure 36. Configuring the Person Context Panel Widget

Horizontal Layout Widget

As described earlier, this widget is a component renderer which is not associated with a domain definition, instead it is associated with a "style". A separate configuration file is used for component renderers. The `StylesConfig.xml` file should be added to the client application component, or the existing file should be modified if it already exists, to associate the component-renderer plug-in class with the `horizontal-layout` style as shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<sc:styles
  <sc:style name="horizontal-layout">
    <sc:plug-in name="component-renderer"
                class="sample.HorizontalLayoutRenderer"/>
  </sc:style>
</sc:styles>
```

Figure 37. Configuring the Horizontal Layout Widget

The `horizontal-layout` style is what the "Person Context Panel Widget" used when delegating to the "Horizontal Layout widget" i.e.

```
ContainerBuilder cb
      = ComponentBuilderFactory.createContainerBuilder();
    cb.setStyle(context.getStyle("horizontal-layout"));
```

When the `Container` component is rendered, the `sample.HorizontalLayoutRenderer` class will be used. If a new renderer class is developed to achieve the horizontal layout using a different HTML technique, the `horizontal-layout` style can simply be re-configured to associate it with another renderer class. As long as that class takes the same input (a `Container` component), other widgets which use this style will not require any update.

More information about configuring renderers and other plug-ins is provided in "Configuring Renderers" on page 65.

A Text Field Widget with No Auto-completion

Objective

To show how input controls can be customized with custom widgets and their edit renderers.

Prerequisites

A knowledge of the behavior of Cúram form pages and a reading of the first three chapters of this guide.

Introduction

This chapter will describe edit renderers used to mark up read/write values with HTML. It will expand on the details in the previous chapters by introducing more advanced concepts related to the creation of input controls on HTML forms.

The sample widget presented in this chapter is a text field widget useful for entering sensitive information such as social security numbers (SSN). By default, the `TextEditRenderer` plug-in class is configured as the edit-renderer for most text and numeric values in the out-of-the-box application. The plug-in displays a HTML text input control. For the input of an SSN, it may be desirable to prevent the web browser from storing the SSN in its cache of entered form data and subsequently providing SSN values using its form field auto-completion feature. Microsoft Internet Explorer supports a non-standard HTML attribute to disable auto-completion of the value of a HTML input control. This `autocomplete` attribute will likely have no effect in other web browsers, but may be useful in environments where Internet Explorer is used. The sample will show how to render the HTML text input control, integrate it into a form page, and add the new attribute to disable auto-completion in Internet Explorer.

Defining the HTML

The HTML for the sample text field widget requires only one element, but many attributes. The values of many of the attributes are not defined here and are just shown with a question mark. The values will be provided by the renderer, as explained later.

```
<input type="text" autocomplete="no"
       id="?" name="?"
       value="?" title"?"
       tabindex="?" style="?"/>
```

Figure 38. HTML Output of the Date Picker Widget

Defining the Renderer Class

The `NoAutoCompleteEditRenderer` class is defined in much the same way as the `EMailAddressViewRenderer` class, except that the base class is `AbstractEditRenderer` instead of `AbstractViewRenderer`. The `render` method is the same, as it is defined by the `DomainRenderer` interface that is shared by both abstract base classes.

```
public class NoAutoCompleteEditRenderer
       extends AbstractEditRenderer {

  public void render(
        Field field, DocumentFragment fragment,
        RendererContext context, RendererContract contract)
        throws ClientException, DataAccessException,
               PlugInException {
    // Create the HTML here....
  }
}
```

Figure 39. Declaration of the `NoAutoCompleteEditRenderer` Class

Handling Form Items

A HTML form page contains HTML input controls, such as text fields and check-boxes. Input controls are required where a UIM `FIELD` element contains a `TARGET` connection, as the user must have somewhere to enter the value before submitting it to the targeted server interface property. An edit-renderer must create the appropriate HTML to present an input control.

To select an edit-renderer, the system identifies the domain definition associated with the server interface property of the target connection. Each domain definition has associated edit-renderer and view-renderer plug-in classes. As a target connection is present, the system will automatically use the edit-renderer instead of the view-renderer when rendering the field.

When a form page is presented to a user, the user sets the values of the input controls in the browser. The user then submits the form to send these values to the server's client-tier in a new request. The edit-renderer plug-in type differs from the view-renderer in that the edit-renderer must declare to the system what input control it adds to a form page, so that the system can process the corresponding values when it receives the form submission request. A view-renderer does not add input controls, so it has no such requirement.

The `RendererContext` provides a method for recording form items as they are added to the form page. The `addFormItem` method returns the identifier that should be used as the value of the `id` and `name` attributes of the HTML element. Before

calling this method, the title (or label) of the field must be determined.

```
String title = getTitle(field, context.getDataAccessor());
    String targetID = context.addFormItem(field, title, null);
```

Figure 40. Adding a Form Item to Get a Target ID

The abstract base class provides a `getTitle` method that can determine the title of the given field. This renderer passes the field and this title value to the `addFormItem` method. The third parameter, null, specifies an optional extended path value. Extended path values for form items are not supported in custom widgets.The `addFormItem` method returns a target ID string value that must be used to identify the input control that will be created to correspond to this newly registered form item.

The `addFormItem` method uses the `Field` object and the title string to record the target path of the entered value of that control, the domain definition of the targeted server interface property, and the label of that field. As the form page is rendered, the system records the form items added by all of the edit renderers and embeds all of this extra information into the HTML form on the page.

When the user submits the form, the values of all of the input controls are submitted as ID/value pairs. The ID is the `id` or `name` attribute value of the respective HTML input control element (which attribute is used depends on the browser, so both attributes are added and set to the same value by the edit-renderer plug-in). The information about the form items recorded and embedded in the form by the system is also submitted at this time. The system combines the input control's ID and value with the embedded form item data that records IDs and target paths. The system can thus determine automatically which submitted values should be assigned to which server interface properties identified by the target paths. The label is used in the event of a validation error, so that the error message can report the label of the field in error.

Accessing the Data

As described in an earlier chapter, the `Field` object has a `Binding` property that defines the source path and target path that identify the data that is bound to the field. For a view-renderer, only the source path is set; it can be resolved to get the value to be displayed. For an edit-renderer, the target path is always set, as it determines where the value will go when the form is submitted. However, the source path may or may not be set. If the source path is set, then the resolved value is used as the initial value of the input control. If the source path is not set, then the input control will have no explicit initial value.

When no explicit initial value is defined, an initial value may still be displayed. The UIM `FIELD` element supports a `USE_DEFAULT` attribute. If this attribute is set to `false`, then no default initial value will be displayed in the absence of a source connection. However, if the attribute is set to `true`, then the default value is determined from a default value domain plug-in. The domain of the targeted server interface property is identified and the associated default value plug-in is invoked to get the default value to be displayed in the input control. If not set, the value of the `USE_DEFAULT` attribute is assumed to be `true`.

Default value plug-ins are configured for all domains out-of-the-box in Cúram, but they may be customized. Typically, the default value of a string domain is an empty string, the default value of a numeric domain is zero and the default value

of a date or date-time domain is the current date and time. See the *Cúram Web Client Reference Manual* for more information about default value domain plug-ins and the user of the `USE_DEFAULT` attribute.

Catering for explicit or default initial values is still not sufficient to determine the correct initial value. When a validation error occurs, the system renders the form again and displays error messages detailing what fields are in error. The values displayed in the HTML input controls in this case are the values entered by the user before submitting the form. Regardless of what initial values where originally shown, the user may have changed any or all of these values. Depending on circumstances, then, the initial value of the HTML input control could be set from the source path, set from a default value plug-in or set by the user. To simplify the handling of these conditions, the `RendererContext` provides a facility to get the appropriate initial value for a form item.

```
boolean useDefault = !"false".equalsIgnoreCase(
        field.getParameters().get(FieldParameters.USE_DEFAULT));
    String value = context.getFormItemInitialValue(
        field, useDefault, null);
```

Figure 41. Getting the Initial Value for a Form Item

First, the renderer retrieves the parameters of the *field* argument. The parameters are a map that associates named parameters with values, all strings. These represent, for the most part, the attributes set on the UIM `FIELD` element. Where attributes are not set in the UIM and default values for those attributes need to be handled, the renderer must respect this requirement. Above, if the value of the `USE_DEFAULT` field parameter is anything other than `"false"`, including if it is not defined, then the *useDefault* variable will be set to true, which is the correct default value for this UIM attribute and field parameter.

The appropriate initial value for the input control can now be retrieved by calling `getFormItemInitialValue` on the *context* object. The third argument, null, is an optional extended path value that is not supported in custom renderers.

Generating the HTML Content

As before, the DOM Core API is used to create the HTML content and the content to be rendered is appended to the `DocumentFragment` passed to the `render` method.

```
Element input = fragment.getOwnerDocument()
        .createElement("input");
    fragment.appendChild(input);

    input.setAttribute("type", "text");
    input.setAttribute("autocomplete", "no");
    input.setAttribute("id", targetID);
    input.setAttribute("name", targetID);

    if (title != null && title.length() > 0) {
      input.setAttribute("title", title);
    }

    if (value != null && value.length() > 0) {
      input.setAttribute("value", value);
    }
```

Figure 42. Marking Up the Input Control

The first statement creates the HTML `input` element. The `input` element is then added to the document fragment. The required attributes are then set on the

element. Note that both the `id` and the `name` attributes are defined and assigned the same target ID value; this ensures compatibility with most web browsers. The `title` and `value` attributes are only set if they are not null and not empty strings.

There are several other features of fields in UIM that the renderer must support. The code required to implement the basic features is shown below.

```
if ("true".equals(field.getParameters()
                        .get(FieldParameters.INITIAL_FOCUS))) {
      input.setAttribute("tabindex", "1");
    }

    String width
        = field.getParameters().get(FieldParameters.WIDTH);
    if (width != null && width.length() > 0
        && !"0".equals(width)) {
      String units;
      if ("CHARS".equals(field.getParameters()
                              .get(FieldParameters.WIDTH_UNITS))) {
        units = "em";
      } else {
        units = "%";
      }
      input.setAttribute("style", "width:" + width + units + ";");
    }

    setScriptAttributes(input, field);
```

Figure 43. Supporting Other UIM Features

When a form page is first shown, the input focus is normally given to the first input control on that page. However, if the `INITIAL_FOCUS` attribute is set to `true` on a UIM `FIELD` element other than the first one, the input focus should be given to that field instead. If not specified, the `INITIAL_FOCUS` attribute is assumed to be set to `false`.

Support for this feature can be achieved by setting the `tabindex` attribute of the HTML `input` element to 1 if the field object's `INITIAL_FOCUS` parameter is set to `"true"` (as it reflects the value defined for the corresponding attribute in UIM). The parameter value may be null, but calling the `equals` method on the literal string value is still safe in that case and yields the desired result.

The width of an input control is set by combining the *WIDTH* parameter value with the *WIDTH_UNITS* parameter value. Both values are optional and may be null. If the *WIDTH* parameter is null, is empty, or is explicitly set to zero, then the width is not set on the input control. If the *WIDTH_UNITS* parameter is null or not recognized, then `"PERCENT"` is assumed. The width is set using the `style` attribute of the `input` element.

UIM `FIELD` elements support child `SCRIPT` elements that define JavaScript handlers to be associated with the rendered HTML content. The `SCRIPT` elements are transposed into further parameter values on the `Field` object passed to the renderer. For example, this UIM `SCRIPT` element will be represented as a parameter named *ONCLICK_ACTION* with a value set to the value of the `ACTION` attribute in the UIM:

```
<SCRIPT EVENT="ONCLICK" ACTION="doSomething();"/>
```

There can be many different scripts for different events. A helper method provided by the abstract base class can set all of the appropriate event attributes on a HTML

element for these scripts. Simply call `setScriptAttributes` passing the HTML element to which to add any required event attributes and the `Field` object on which the parameters record the necessary information.

Configuring the Widget

To configure the SSN text field widget in isolation from other text field widgets, the data must be in a domain that is specific to SSNs. Here, the SAMPLE_SSN domain is assumed. The `DomainsConfig.xml` file should be added to the client application component, or the existing file should be modified if it already exists, to associate the edit-renderer plug-in class with that domain.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dc:domains
  <dc:domain name="SAMPLE_SSN">
    <dc:plug-in name="edit-renderer"
                class="sample.NoAutoCompleteEditRenderer"/>
  </dc:domain>

</dc:domains>
```

Figure 44. Configuring the SSN Edit Renderer

Applying the above configuration, the edit-renderer of the custom widget will now be invoked anywhere a UIM `FIELD` element has a *target* connection to a server interface property in the SAMPLE_ SSN domain. If the UIM `FIELD` has no target connection, the view-renderer will be used instead. As no view-renderer is defined in this configuration, the view-renderer of the parent or other ancestor domain, will be inherited and used. Typically, this will be the `TextViewRenderer` associated by default with the SVR_STRING domain.

More information about configuring renderers and other plug-ins is provided in "Configuring Renderers" on page 65.

Limitations on Support for Custom Edit Renderers

Only the development of custom edit-renderer plug-ins with these limitations is supported:

- The renderer must not be used within the context of a rendering cascade; it may only be used where invoked in direct correspondence to a UIM `FIELD` element.
- The renderer must not be used in the context of a UIM `LIST` element.
- The renderer must add no more than one form item to a form page.
- The renderer must not process code-table items.
- The renderer must not use any features of the Renderer API other than those demonstrated in this chapter.

Internationalization and Localization

Objective

To provide a basic understanding of the internationalization and localization processes and how they apply to widget development.

Prerequisites

A knowledge of the concept of a locale and an understanding of the impact of a locale on the operation of a software application.

Introduction

Internationalization is the process of enabling a software application to function equally well in any of its supported locales; to enable it to be localized. Localization is the process of modifying elements of an application to support the requirements of a particular locale. For any application required to support more than one locale, the widget developer must internationalize the widget to ensure that it can be localized with ease.

Note: Internationalization and localization are long words. They are commonly abbreviated as i18n and L10n respectively. The number in each abbreviation is the number of letters that have been removed between the first and last letters of the original word. A capital "L" is used in L10n to avoid confusion with the "i" in i18n, which may be capitalized at the start of a sentence. Internationalization is also sometimes referred to as "international-enabling" or "national language support" (NLS).

Localization is a process that usually takes place after development. The natural language text elements of the application are typically submitted to an agency that specializes in language translation. The agency returns the text elements translated into a new language and this text is then incorporated back into the application. This process is only possible if the application makes it easy to package up the text elements and replace them with text in another language; if the application has been properly internationalized.

There are many other aspects to localization. Some of these are handle automatically by the CDEJ and some remain the concern of the widget developer.

CDEJ Support for Internationalization

The CDEJ is internationalized in many ways. Not only are text elements separated out to standard Java properties files, but other elements are also localized automatically:

- All CDEJ plug-in classes of all types expose the locale and time zone of the active user through the `getLocale` and `getTimeZone` methods. The active user is the user who initiated the request for the HTML page currently being rendered on the web container's request service thread. The widget developer can access this information and use it as required.
- Locale-aware sort orders are supported by special locale-aware versions of the comparator plug-ins provided with the CDEJ. These use Java's Collator API, but can be overridden to support custom sorting rules if required.
- Locales can be define both the language and the country and the CDEJ uses this information to support spelling variations of the same language in different countries.
- The converter plug-ins for numeric values automatically apply the rules of the active user's locale when formatting or parsing numbers, ensuring that decimal points and grouping separators are presented or handled appropriately. For date values, similarly, non-numeric months names are translated as appropriate.

In general, there is no need to specify the locale when accessing the CDEJ rendering API, as the locale is automatically determined and applied when necessary. Some types of plug-ins, particularly the converter plug-ins described in the *Cúram Web Client Reference Manual*, need to handle the locale carefully, but this is generally not the case for renderer plug-ins. When renderer plug-ins resolve

paths to their values, the values are provided via the converter plug-ins, or other locale-aware sources, and the localization will happen automatically before the value is returned.

Widget Internationalization

Not all localization is handled automatically by the internationalization features of the CDEJ. Widgets may have specific localization requirements that are not covered by the CDEJ and the widget developer must internationalize the widget to accommodate these. The main internationalization issues of concern to the widget developer are:

- accessing and rendering localized text values;
- referencing localized versions of images or icons;
- providing locale information and localized text elements to JavaScript code used by a widget in the web browser;
- laying out content on the HTML page in a way that can accommodate the increased length of text when localized into other languages.

"Accessing Data with Paths" on page 68 provides details on how to construct paths that identify localized text properties resources on the classpath or in the Application Resource Store and to resolve these paths to the localized text values. Examples of this process are also provided in that appendix. Once retrieved, the localized text can be incorporated into the HTML mark-up produced by a renderer plug-in class.

Localized images are often required where the images contain text or other symbols that are specific to one language or culture. The developer should avoid including text in images where possible. It makes the application harder and more expensive to localize and also affects the accessibility of the application. "Accessibility Concerns" on page 47 describes how applications are often required to be accessible to as many people as possible. People with visual impairments may find that text in images is difficult to read or entirely unreadable. Nevertheless, internationalizing such elements is a simple process. The HTML produced by the widget's renderer plug-in class includes a `img` element with a `src` attribute that references an image resource on the application server. These image resources can be added to the `WebContent` folder of an application component. A simple scheme to support internationalization then places image files in sub-folders named for the locales. For example, create an `images` folder within the `WebContent` folder. Create folders named `en` (English) and `es` (Spanish) within that `images` folder. Now place the localized image files for English and Spanish into their respective locale folders. Within the renderer, the localized image can be referenced as shown in the example below. The context of the example is the `render` method of a renderer plug-in class.

```
Element img = fragment.getOwnerDocument().createElement("img");
img.setAttribute("src",
    "../images/" + getLocale().toString + "/icon.png");
```

Figure 45. Referencing Localized Image Files

The `getLocale` method returns the locale of the active user, so the image source URI could be generated as, for example, `../images/en/icon.png` for a user in the English locale and `../images/es/icon.png` for the Spanish locale. Alternatively, the locale folder could be omitted and the locale could could appear in the image file name.

A problem with this scheme is that a user with a locale `en_US` will not seen any image, as there is no `en_US` folder within the `images` folder. For text properties, a locale fall-back scheme is used, but that does not apply in the example above. There are a number of ways to accommodate extra locales:

- create one folder for each supported locale and place the localized images in those folders, even if the image is the same for several locales, such as if `en`, `en_US` and `en_GB` were supported simultaneously and there were no spelling variations across those locales for the words used in the images;
- for each image, define a property in a localized text properties resource containing the path to image appropriate for the locale of that properties resource. Instead of constructing the path in the renderer, resolve the text property that contains the path and use that. This scheme is similar to the use of the Images.properties file in UIM development described in the *Cúram Web Client Reference Manual* and allows the normal locale fallback mechanism to operate. (An overview of this fallback mechanism is provided in "Accessing Data with Paths" on page 68.)

There is a separate type of text-based image generation and localization feature in the CDEJ that is described in the *Cúram Web Client Reference Manual*. It is not directly related to widget development.

Widgets that depend on JavaScript libraries and scripts may require that the JavaScript be internationalized. The two main requirements are to supply the JavaScript code with the correct locale to ensure that localization features of the JavaScript library are used correctly and/or to supply localized text elements to the JavaScript routines. The specific requirements vary between widgets and are beyond the scope of this guide. However, the basic approach for the widget developer is to generate JavaScript content containing the required information from locale information and localized text values available to the renderer plug-in class. For example, the renderer plug-in can generate a script containing a class to a JavaScript function that passes the value of the active user's locale. The locale value is embedded in the function call in a same way it was embedded in the image URI in "Widget Internationalization" on page 46, by calling `getLocale` and converting it to a string. Localized text elements retrieved by the renderer plug-in class can also be embedded into a script, perhaps into a JavaScript array or object, depending on requirements.

The layout of a page may also be affected by localization requirements. The text of a label in one language may become much longer when translated into another language. An average of 30% more space should be added for any English text to accommodate the replacement of that text with text in other languages. However, depending on the language and the phrase, the text could be require twice the amount of space or even more.

Accessibility Concerns

Objective

To introduce the developer to accessibility concerns in the context of custom widget development and to provide some guidance on how to address those concerns.

Prerequisites

A basic knowledge of HTML.

Introduction

The accessibility of the application determines how usable the application is by people of all abilities and disabilities. Typically, accessibility concerns focus on the needs of people with disabilities, such as visual or motor impairments, and the compliance with the regulatory requirements to accommodate their needs. Their needs may include some of the following:

- higher contrast visual presentation to make the content easier to read;
- color schemes that are suitable for people with deficiencies in their color vision;
- the ability to zoom in to the content on the page or increase font sizes independently of the application's styling;
- access key support to allow the application to be used with a keyboard only and not require a mouse;
- additional information associated with images and form input controls to allow a screen reader (voice browser) to identify them to the user.

The regulatory requirements differ between jurisdictions. There is no universal solution for all of the accessibility requirements. However, many local regulations and guidelines draw from those developed by the W3C Web Accessibility Initiative (WAI) and its *Web Content Accessibility Guidelines* (WCAG). The WAI is a good starting point for widget developers wishing to learn more about accessibility and its application to the web. The widget developer should identify what the accessibility regulations and guidelines are for the jurisdiction in which the application will be employed and aim to comply with those. It is beyond the scope of this guide to cover all of the possible regulations.

Labels for Form Input Controls

The correct labeling of input controls on forms is typically the most important accessibility concern of the widget developer. A visually impaired user may use a screen reader to access the application. A screen reader is a software application that converts the text of a web page (or other application) into speech, allowing the user to hear what is present and respond appropriately. When using a form, the screen reader will inform the user of the input control that currently has the input focus. For example, the user may use the Tab key to move the focus to the text field with the label **Date of Birth** and the screen reader will announce "Date of Birth, edit"; adding the word "edit" to notify the user that the control is editable. This is only possible if the screen reader can associate the label of the field with the input control for that field.

All of the accessibility standards require that input controls on forms be identified by labels that can be used by a screen reader. The implementation guidelines for these standards often demonstrate the use of the HTML `label` element that allows the label text to be marked up with an element that defines the ID of the input control for which that text is the label. Some validation tools then enforce this particular implementation guideline to the exclusion of all others. The CDEJ does not use the HTML label element to associate label text with form input controls, it uses an alternative method. The HTML of a Cúram application page may fail an automated accessibility validation check for this reason, but this failure is erroneous and does not affect the accessible of the form input controls to a screen reader application.

The technique used by the CDEJ is the same technique that widget developers should use. The visible label of the input control is rendered separately and automatically by the CDEJ and the `title` attribute of the `input` element is set to

the value of the label that should be read by the screen reader for that control. There are several reasons why this approach is used by the CDEJ instead of the often suggested `label` element:

- The `label` element displays its label as the visible label on the page for the form control. It is not possible to associate a single `label` element with more than one input control, as it may only have one ID value in its `for` attribute. For example, a UIM `CONTAINER` element is used and it contains two `FIELD` elements. One label, that of the container, will appear beside two input controls, one for each field. A search form may have a **Surname** label appearing beside a text field and a check-box. The user inputs the surname into the text field and checks the checkbox if the search should find names that sound like that surname. Using a label element, it is not possible to label these controls without displaying two labels on the page and that is not desired. However, it is easily achieved using the `title` attribute on the `input` elements for the text field and the checkbox. The values of the `title` attributes are set from the labels of the UIM `FIELD` elements, not the `CONTAINER` element, so the labels can be specific to each input control while the visual presentation is still uncluttered.
- Most browsers use the `title` attribute of an input control as the text displayed in a tool-tip shown then the use hovers over the control with the mouse pointer. This allows sighed users to identify controls even if the specific label for the control is not shown on the page. For example, the label of the **Sounds Like** check-box in the example above. Using the `title` attribute, therefore, makes the application more accessible to sighted users, too.
- For mandatory input fields, an icon is displayed beside the label of the field to alert the user to the fact that a value must be entered. This icon is not apparent to a screen reader application, as it is applied using a CSS style rule and is not part of the content of the HTML document. For accessibility, the word "mandatory" can be appended to the label value used in the `title` attribute of the input control while omitting it from the visible label that already has the visible mandatory icon. It is not possible for the visible label to differ from the input control label in this way if the `label` element is used.
- When rendering a page, the CDEJ renders the field label before invoking the widget's renderer plug-in for the field value (assuming labels of shown to the left of the values). As the CDEJ does dictate what input control will be produced by an edit-renderer plug-in, it cannot know in advance what the ID of the control will be and cannot set an ID in the `for` attribute of a `label` element. It is not possible, therefore, to use the `label` element while allowing widgets for the field values to be customized. This is not a problem, as the `label` element is not desirable for all of the other reasons described above.

These are the main reasons why the CDEJ uses and recommends the `title` attribute in preference to the `label` element. The application pages are equally, if not more, accessible to screen reader applications and users as a result. Any spurious errors from accessibility validation tools relating to the non-use of the `label` element can be safely ignored once the presence of the `title` attribute has been confirmed.

Font Sizes

It is recommended that the use of, relative font sizes when styling a widget's HTML output. Relative font sizes, specified as a percentage of the web browser's base font size, allow the user to change the base font size in their browser to effectively magnify all of the text on the page. Some modern web browser can scale up the text even if fixed font sizes are specified, but some browsers do not

change fixed font sizes properly when scaling the page, or will only scale the text along with all other non-text content, which may not be the user's preference.

Overview of the Renderer Component Model

Elements of the Model

More complete details of the renderer component model are provided in the CDEJ Javadoc. The information presented here is an overview of the main elements in the model and how they relate to each other.

There are three main categories of elements in the renderer component model:

- Elements that define components of the page. These are the elements of the model that are passed to renderer plug-in classes for rendering.
- Elements that provide additional information about a component.
- Elements that are used to create components.

The elements of the model are defined using Java interfaces. All of the interfaces are defined in the `curam.util.client.model` package.

The main interfaces that define the component of the page are as follows:

`Component`
The `Component` interface defines the common properties of all elements that may be rendered to HTML by renderer plug-ins. A component can be associated with a style and rendered with a component-renderer plug-in.

`Field` The `Field` interface extends the `Component` interface and adds the binding and domain properties. The binding records the connections defined in UIM for the field. The domain records the domain of the server interface property of the target connection, or that of the source connection if there is no target connection. A `Field`, being a `Component`, can be associated with a style, but it is more usual to associate a field with a domain. If both a domain and a style are defined, the domain will be used when selecting the appropriate renderer plug-in. A field may also be rendered with a component-renderer plug-in, but a view-renderer or edit-renderer will be used if the domain property is set.

`Container`
The `Container` interfaces extends the `Component` interface and allows the component to contain other components. The children of a container are recorded in a list; the order in which the children are added will be the iteration order of that list. A container can be associated with a style and rendered with a component-renderer plug-in.

The main interfaces that provide additional information about a component are as follows:

`Binding`
A `Binding` is used exclusively with a `Field` object to record its source and target path defined by the corresponding connection in UIM. A binding defines other paths, mostly related to the use of the UIM `INITIAL` connection element, but their use, or the use of the `INITIAL` element, in combination with custom widgets is not supported in the Cúram application.

`ComponentParameters`
A component's parameter values, usually derived from the corresponding

UIM attributes, are stored in a ComponentParameters object retrieved by calling Component.getParameters. The interface extends java.util.Map<String, String>, but the returned map may not be modified. When building new components at run-time, add additional parameters as necessary.

Link A Link represents a hyperlink to another destination. A link defines a target and an arbitrary collection of parameters. The target and the parameter values are defined using paths, not literal values. However, paths can be constructed to represent literal values if required. See "Accessing Data with Paths" on page 68 for more details.

The main interfaces that are used to create new components are as follows:

ComponentBuilder
A ComponentBuilder is used to build basic components. This interface also defines the properties common to the other builder interfaces.

FieldBuilder
A FieldBuilder extends a ComponentBuilder to allow the source path, target path and domain to be set. Other paths may be set, but their use is not supported in the Cúram application.

ContainerBuilder
A ContainerBuilder extends a ComponentBuilder to allow components, fields or other containers, to be added to a new container.

Building Components

Components of the model are constructed using the *builder pattern*, which is a software design pattern. Different types of components require the use of different builders. The interfaces for these builders were listed in the previous section. However, a concrete implementation of a builder is required to do any real work. Builder *objects* can be created using the ComponentBuilderFactory class defined in the curam.util.client.model package. The factory class provides a number of *factory methods* to create builders. Only the use of the following factory methods are supported in the Cúram application:

createComponentBuilder
Creates and returns an object implementing the ComponentBuilder interface. Use this to build generic components that do not require a binding and that do not contain other components.

createFieldBuilder
Creates and returns an object implementing the FieldBuilder interface. Use this to build fields that are bound to data sources.

createContainerBuilder
Creates and returns an object implementing the ContainerBuilder interface. Use this to build components that may contain other components of any kind.

The component builders present a simple, flat API for creating components. They eliminate the need to understand the internal structure of components. In particular, the properties of the objects that hold additional information about a component, such as bindings, parameters and links can be defined directly through the builder interface; there is no need to create instances of these objects or understand how they are stored.

To use a builder, instantiate it using the appropriate factory method and then call the appropriate *setter* methods to set the properties of the component being built. When complete, call `getComponent` to get the instance of the newly built component object. When `getComponent` is called and has returned the new component, the builder object resets all of the properties and may be reused to build another component. Until `getComponent` is called, many of the simple properties can be set again to overwrite their existing values. However, this may not work for properties that represent items in collections, such as the parameters of the component.

Once built, components are immutable, much like `java.lang.String` objects, or the `Path` objects described in "Accessing Data with Paths" on page 68. The only way to change a property of a component is to build a new component with the modified value for that property. Component builders can be used to create entirely new components, but are commonly used to create new components that are modified copies of other components to overcome this immutability. The starting point in this process is the component that will act as the *prototype* for the new component. Create the builder object and then pass the prototype component to the builder's `copy` method. This will set all of the properties of the component component to be built from the properties of the prototype component. Use the setter method of the builder to overwrite (including with a null value) the properties of the new component that differ from the prototype component. Finally, call the `getComponent` method on the builder to get the new component that is the modified copy of the original, prototype component. A typical use of this copy-and-modify process is when making multiple copies of a `Field` object, changing the domain and extending the paths, before delegating the copy of the field for rendering by another renderer plug-in class.

When copying a prototype `Container` object using the builder's `copy` method, all of the child components of the container are copied by reference. A reference is sufficient, as the child components are immutable. Because references are used, any child that is itself a container will become a child of the new container complete with its own child components. When it is necessary to change the children of a `Container` that must be copied using a builder, the `copyShallow` method should be called on the `ContainerBuilder` instead of the `copy` method. The `copyShallow` method does not copy any references to the child components. Copy these one-by-one by iterating over the child components of the prototype container and then calling the `add` method on the `ContainerBuilder`. The child components can be copied and modified, or even selectively omitted, during this process if required.

Design and Implementation Guidelines

Introduction

Custom widgets provide the developer with considerable power and flexibility when meeting challenging presentation requirements. However, widget development can be complex and it raises many design issues that are usually not a concern of a client application developer used to using only UIM to define the content of pages. The next section presents some guidelines for writing renderer plug-in classes to assist the developer in avoiding some of the common pitfalls.

Some renderer plug-ins also need to support the requirements of field-level security. This is explained and demonstrated in the final section.

Guidelines for Writing Renderers

Do Keep Things Simple

"Approaches to Customization" on page 4 described the approaches to widget development in order of increasing complexity. Endeavor to keep the complexity of any new widget as low as possible by selecting the simplest viable approach. It is always possible to change to a more complex approach later if necessary, but it is much harder to simplify a widget after first committing to a complex approach.

Pay particular attention to widgets that will be used very widely. Simplicity and efficiency are very important in this case. A complex widget that will be used on many pages by many concurrent users can be difficult to develop without much prior experience.

Do Divide and Conquer

A complex widget implemented as a single, large `render` method is difficult to maintain and offers no opportunity to reuse its component parts, as it has none. Where a widget renders more than a single value, consider dividing it up into a group of cooperating renderer plug-ins. This will result in smaller, more manageable components. These components can be reconfigured or reused in other contexts to meet future requirements.

Development of a complete renderer can progress toward the final goal in stages. For example, take the widget described in "A Details Widget Demonstrating Widget Re-use" on page 28. This requirement could not be met using multiple fields in a UIM `CLUSTER` element because the layout would not fit into the strict grid provided by a cluster. However, an alternative approach to its development is this sequence:

1. Create a UIM page containing a `CLUSTER` element and place separate fields for the details within the cluster.
2. Create widgets to render each of the fields a manner closer to that required in the final details widget.
3. Assess if the solution is "close enough" to be acceptable and release the change if it is.
4. If the cluster layout is still too limiting, develop a widget to lay out the fields in the required manner. This will require a change to the data to make it a single value, an XML document. Reuse all of the smaller widgets in a rendering cascade.

All of the widgets developed in the second step are reused in the context of the last step. This allows greater flexibility in planning the work, as the functionality can be released early and refined at a later time, if it is still necessary. The individual widgets developed in the second step can also be reused when developing other details panel widgets, or widgets for unrelated purposes.

Do Check for Nulls

Renderer plug-ins may be supplied with null values, so check for null values to avoid errors. The main values that may be null are the paths of the field's binding, the field's parameters and the values resolved using paths.

The CDEJ will never supply null arguments to the `render` method, but if one renderer invokes another, this cannot be guaranteed. In a view-renderer, the field's source path will never be null, but the target path will always be null; these do not

need to be checked if this is assumed. In an edit-renderer, the field's target path will never be null, but the source path may or may not be null and should always be checked.

The field's parameters may or may not be null. Typically, the parameters reflect the attributes used in the UIM. However, if an attribute was set to the same value as its default value, or was not set at all, then the parameter value is likely to be null. Always check parameter values for null and, if they are null, ensure that the renderer treats this the same as the default value for the corresponding UIM attribute. The default values for the attributes are described in the *Cúram Web Client Reference Manual*.

On resolving paths using the `DataAccessor`, the values may be null in some cases. A path to a server interface property will not resolve to null, the `DataAccessor` will throw an exception instead. Paths to values within an XML document that are resolved using a `SimpleXPathMarshal` may result in a null value. See "Extending Paths for XML Data Access" on page 73 for details on the conditions that can result in null values.

Do Take Shortcuts

Renderer plug-in classes must extend the prescribed abstract base classes identified earlier in this guide. However, the extension does not have to be direct. There is no prohibition against creating new base classes custom renderers or extending other custom renderer plug-in classes as long as the prescribed abstract base class is an ancestor class of any custom renderer class. This option can be exploited to share code between custom renderers more effectively and to develop renderers that are variations on other renderers without implementing all the code from scratch. Note however, that the extension of the CDEJ renderer plug-ins for custom widget development, is not supported in the Cúram application.

Widget development, particularly in the area of creating and manipulating DOM nodes for the HTML content can be repetitive. Consider writing a simple utility class to wrap up common operations, such as checking if a string value is null or empty before setting an attribute on an element, or creating and appending text nodes.

Do Go with the Flow

Combining several renderer classes into a rendering cascade is a powerful technique for enabling maximum reuse of widgets in other contexts. However, this technique requires that the renderers conform to the expectations of the renderer API and the CDEJ that manages it rather than try to do things another way. Renderer classes should respect the imperative to render the data referenced by the paths in the Field object's binding without trying to examine what the paths represent of react differently to different kinds of paths. Any renderer class that implements special handling of paths or other information is likely to be unusable in all but the context for which it was first developed.

The key to going with the flow in a rendering cascade is to develop view-renderer and edit-renderer classes in a manner that makes them suitable for direct use in combination with a UIM `FIELD` element. This should be the case even for renderer classes that are never intended to be used directly in this way and only intended to be used in the context of a complex widget's rendering cascade. Making this the design goal ensures that the renderer class is context independent and will maximize the possibilities for its reuse.

When using XML document, it may be necessary to change the structure of the data to suit the rendering cascade. For example, a contact details widget is required to display the contact details of a person. The widget is expected, when complete, to provide reusable widgets that display the postal address and e-mail address of the person in the required form. The developer first conceives that the XML consumed by the new contact details widget will have the form shown below.

```
<contact>
  <name>James Smith</name>
  <street>Main Street</street>
  <city>Springfield</city>
  <phone>555-555-0101</phone>
  <e-mail>james@example.com</e-mail>
</contact>
```

Figure 46. An XML Document Describing Contact Details

The initially invoked renderer plug-in for the new widget, the contact renderer, will use the copy-and-modify technique on the `Field` object described in "Overview of the Renderer Component Model" on page 50 and demonstrated in "Tying Widgets Together in a Cascade" on page 32 and then delegate the rendering of these copied objects to the other renderers. To the address widget, the contact widget delegates a `Field` object whose source path is extended with `/contact` and the address widget will further extend this path with `/street` and `/city` to resolve and present the address values.

This arrangement will work, but the reusability of the address widget has been compromised by the order in which the paths have been extended. This is a consequence of the structure of the XML document. Were the address renderer to be used in a standalone address widget, its XML data might look like this:

```
<address>
  <street>Main Street</street>
  <city>Springfield</city>
</address>
```

Figure 47. An XML Document Describing an Address

The street and city elements are contained within an address element, as the XML document would not be valid without a single root element. This requires that the address renderer extend the source path (in this case just the path that identifies the server interface property itself) with `/address/street` and `/address/city`. These path extensions are not the same as those used with the address renderer was invoked by the contact renderer, so something is wrong.

This problem could be solved by having the contact renderer set a field parameter on the copy of the field passed to the address renderer instructing the renderer to extend the paths in different ways. This field parameter would not be set if the address renderer were invoked directly in correspondence with a UIM `FIELD` element, so the context could then be determined. However, this complicates both renderers in several ways. The contact renderer must accommodate the requirements of the address renderer to extend paths in one of two ways, the address renderer must check a field parameter value and then operate differently depending on the result. The XML is different for the address in each case, so any code that generates this XML would need to accommodate the requirements of the two renderers. Testing also becomes more difficult, as there are more paths through the code and more edge cases to consider. Therefore, this is not the right solution to the problem.

The alternative is much simpler: simply revise the structure of the XML document to the form shown below.

```
<contact>
  <address>
    <street>Main Street</street>
    <city>Springfield</city>
  </address>
  <phone>555-555-0101</phone>
  <e-mail>james@example.com</e-mail>
</contact>
```

Figure 48. A Revised XML Document Describing Contact Details

The address details are now embedded in the contact details XML document in the same form as they would appear in a standalone address XML document. As before, the contact renderer extends the path with `/contact` before delegating to the address renderer and then the address renderer extends that path further with `/address/street` and `/address/city`, just as it would do in the standalone use case. There is no need for any conditional processing and the need to deliver an address renderer that works in the context of a rendering cascade or when directly associated with a UIM `FIELD` element has not resulted in any added complication.

The situation for the e-mail address value is slightly different. In the standalone use case, the e-mail address renderer does not expect an XML document, just a string value containing the e-mail address. To accommodate this, the contact details renderer should extend the path for the e-mail address using `/contact/e-mail` before delegating the rendering of the value. Both renderers can now operate without any additional complication, as the e-mail address renderer will blindly resolve its source path to the e-mail address value and be unaffected by the fact that the path may either directly refer to a server interface property value or be extended to refer to a value within an XML document. In either case, the result of calling `DataAccessor.get` on the source path will be the string value of the e-mail address.

To design a rendering cascade that is effective in reusing renderers in a new context, proceed as follows:

- Design the individual renderers first as if they were to be invoked directly in association with a UIM `FIELD` element and define the format of the data that they will consume and the paths that they may extend to access that data.
- Move on to the design of the delegating renderer that will delegate to the above simple renderers. Determine how it will create new components and extend their paths to accommodate the needs of the simple renderers.
- Leave any decisions about the form of the aggregate XML document until the end, as it will follow from the design of the renderers in the cascade, not the other way around.

Taking this bottom-up approach to the design will ensure that each of the ultimate elements in the rendering cascade are clearly defined and readily reusable. Taking a top-down approach may seem to work well at first, but it is almost inevitable that some problem will occur at the final level that results in the need to start the whole design again, as the design flaw cascades back in the opposite direction to the intended rendering cascade.

Don't Introduce Concurrency Issues

The application may service requests from many users at the same time. Even when a single user is active, the application may still receive concurrent requests

for several pages that are presented to that user in the tabbed user interface. At run-time, only one instance of each renderer plug-in class is created for each domain or style. The application may use the same plug-in instance to service concurrent requests from one or more users. This places some restrictions on the implementation of a renderer plug-in class to avoid concurrency problems. The restrictions also apply to all other kinds of domain and style plug-ins, as they share the same life-cycle as renderer plug-ins.

Maintaining state information within a plug-in instance will cause concurrency problems. A developer may introduce a dependency on state information when factoring a large render method into smaller, more manageable, private methods. If, instead of passing all information between methods using method arguments, the developer passes information through fields of the plug-in class, concurrency defects will arise. “Don't Introduce Concurrency Issues” on page 56 shows such a defect.

```
        public class DefectiveEMailAddressViewRenderer
      extends AbstractViewRenderer {
          private String emailAddress;
        public void render(
       Field field, DocumentFragment fragment,
       RendererContext context, RendererContract contract)
       throws ClientException, DataAccessException,
              PlugInException {

    emailAddress = context.getDataAccessor()
       .get(field.getBinding().getSourcePath());

    Document doc = fragment.getOwnerDocument();

    Element span = doc.createElement("span");
    span.setAttribute("class", "email-container");
    span.appendChild(createAnchor(doc));
    fragment.appendChild(span);
  }

  private Element createAnchor(Document doc) {
    Element anchor = doc.createElement("a");
    anchor.setAttribute("href", "mailto:" + emailAddress);

    Element img = doc.createElement("img");
    img.setAttribute("src", "../Images/email_icon.png");
    anchor.appendChild(img);

    anchor.appendChild(doc.createTextNode(emailAddress));
    return anchor;
  }
}
```

Figure 49. A Plug-in Class with a Concurrency Defect

The `DefectiveEMailAddressViewRenderer` class is similar to the `EMailAddressViewRenderer` class developed in “An E-Mail Address Widget” on page 13. The defective class has a `createAnchor` method to organize the code for improved readability. However, rather than pass the e-mail address value as a method argument, the e-mail address is defined as a field of the class that is set by the `render` method and read by the `createAnchor` method. At run-time, there may be concurrent requests for pages containing e-mail addresses, so the `render` method of a single instance of the renderer plug-in for e-mail addresses may be invoked from more than one thread. This can lead to a defect where the shared field value becomes corrupt.

For example, thread T1 services a request from user U1 and thread T2 services a request from user U2. T1 calls the `render` methodon the same plug-in instancejust before T2 does. T1 sets the `emailAddress` field value to e-mail address E1and then T2 immediately sets the field to E2. Now, when T1 invokes `createAnchor`, e-mail address E2 will be rendered and shown to user U1. This may not be a serious problem for e-mail addresses, but the same defect could lead to unwanted leaking of more sensitive information. In the case of edit-renderer plug-in initializing form field values when modifying entities, the problem could also result in incorrect values being written to the database.

It is also important to note that concurrency problems do not necessarily arise because there are two or more users active; they arise because there are two or more requests active. With the tabbed user interface, it is very likely that a single user can trigger concurrent requests for pages. Do not dismiss potential concurrency problems on the mistaken assumption that data that is local to a user, such as data stored in Java EE session attributes, is immune from such problems.

The remedy for this problems is simple: do not use fields of a class to pass information between methods; use the methods' arguments instead. "Don't Introduce Concurrency Issues" on page 56 shows the alternative implementation that has no concurrency defect because the e-mail address value is passed as an argument to the `createAnchor` method.

```
public class DefectiveEMailAddressViewRenderer
       extends AbstractViewRenderer {

  public void render(
        Field field, DocumentFragment fragment,
        RendererContext context, RendererContract contract)
        throws ClientException, DataAccessException,
               PlugInException {

    String emailAddress = context.getDataAccessor()
       .get(field.getBinding().getSourcePath());

    Document doc = fragment.getOwnerDocument();

    Element span = doc.createElement("span");
    span.setAttribute("class", "email-container");
    span.appendChild(createAnchor(doc, emailAddress));
    fragment.appendChild(span);
  }

  private Element createAnchor(
        Document doc, String emailAddress) {
    Element anchor = doc.createElement("a");
    anchor.setAttribute("href", "mailto:" + emailAddress);

    Element img = doc.createElement("img");
    img.setAttribute("src", "../Images/email_icon.png");
    anchor.appendChild(img);

    anchor.appendChild(doc.createTextNode(emailAddress));
    return anchor;
  }
}
```

Figure 50. A Plug-in Class without a Concurrency Defect

In general, avoid fields of a plug-in class unless they are constants declared static and final. Carefully consider the potential for concurrency defects before

considering the introduction of any non-constant fields and must never introduce fields simply to shorten the argument lists of private methods.

The fields of a plug-in class are the most obvious place to store state information during rendering. However, a developer might store state information in other places, such as in attributes of the Java EE session or application, in *ad hoc* data caches and in helper classes. In introducing any such state storage, consider concurrency issues with the same care given to fields of a plug-in class.

Don't Convert Data in a Renderer

Renderer plug-ins are responsible for marking up field values with HTML for presentation. Converter plug-ins are responsible for converting the server interface property values from their Java object representations to strings formatted appropriately for the active user. Endeavor to maintain this separation of concerns and avoid converting data within a renderer plug-in.

The `format` method of converter plug-ins, described in the *Cúram Web Client Reference Manual*, is called by the CDEJ when servicing the `get` method calls on the `DataAccessor` within the renderer. The `format` method is responsible for converting the Java object representation of a server interface property value to a string. The method applies the active user's locale, time zone, date format and other preferences as appropriate. Implementing this processing in a renderer is redundant, complicated and prone to error. It can also introduce inconsistencies with the presentation of the same type of data in other places in the application. Where the data is not available in a suitable format, consider developing a new converter plug-in to produce the required string representation before developing the renderer plug-in.

Where the data to be converted is retrieved from an XML document, configure and use the `SimpleXPathADCMarshal` class as the domain marshal. When the XML has a suitable form, this domain marshal will automatically invoke the correct converter class for the data, parse it from its generic string representation to a Java object representation and then format it to a string representation appropriate for the active user. This domain marshal is introduced in "A Photograph Widget" on page 20 and described in detail in "Extending Paths for XML Data Access" on page 73.

Don't Do Too Much

The client-tier of the application produces a HTML response for each page request. This CDEJ begins to send this HTML response to the web browser before the full HTML content of the page is complete. The CDEJ invokes a renderer for each field, serializes the `DocumentFragment` populated by the renderer to a HTML string, and then writes this HTML string to the response before invoking the next renderer. This way, very little of the response is held in memory at any one time and resource usage is minimized. This is particularly important for pages that can contain a lot of content or when the application is under heavy load.

A renderer plug-in class is free to produce any HTML content for a field, but bear in mind that the contents of the `DocumentFragment` will be held in memory until the `render` method returns. Only at this time is the fragment serialized and its allocated memory freed. The memory use of widgets that produce a large volume of HTML content may or may not pose a problem. If such a widget is used on many pages and by many concurrent users, asses the potential impact of its high memory use. For widgets that are used rarely or by only a limited number of users, memory use may not be a significant problem.

Using a lot of memory when producing the HTML is not the only resource use issue that can be caused by a renderer plug-in. Renderer plug-ins can also consume a lot of processing resources. Technologies such as Extensible Stylesheet Language Transformations (XSLT) can be employed by renderers to manage the generation of the HTML content. Such processing can require significant processing resources (in addition to memory). Determine if such processing is necessary and plan from the beginning to reduce the impact this may have on the application as a whole.

XSLT processing, for example, is both memory and processor intensive. However, this can be mitigated to some degree by taking care to avoid unnecessary processing. XSLT stylesheets can be loaded from resource on the classpath, but this only needs to be performed once. An instance of a `javax.xml.transform.Templates` object can maintain a copy of the stylesheet in memory and can be used multiple times in a thread-safe manner to eliminate the overhead of loading the XSLT stylesheet each time it is required.

Not only can single, large processing operations pose a problem, so can an excessive number of smaller operations. A renderer is invoked every time the value of a field is rendered on a page, both in clusters and in lists. Minor inefficiencies in renderers that are used to present field values in clusters may go unnoticed, but the same inefficiencies may pose a serious problem in the context of long lists of data. The same view renderer plug-in is used to present read-only fields values in a cluster or in a list where the type of the data is the same. If one or two values are presented in a cluster, the resource use may be acceptable. However, if hundreds of values are presented in a long list, the resource use will increase dramatically.

Renderers that depend on receiving their data in the form of XML documents are a particular common concern. While XML is suitable and convenient in many cases, it is inadvisable to use it for values that may be presented in lists. For each field in a list column, the CDEJ will create an XML parser, parse the XML document, store the result, allow the renderer to query the result, and then, at the end of the request, free all of the used resources. This may appear to perform adequately in a development environment with a single user, but is unlikely to perform well with concurrent users on a heavily loaded application server. Pagination in its current implementation does not change this. All of the data in a paginated list is still rendered up front, it is just presented as if it were being rendered piecemeal.

To avoid serious resource use issues, a developer may decide to present values used in clusters in one way and values used in lists, another. This is only possible if the values have different domain definitions, as it is not possible to configure renderer plug-ins based on the context (cluster or list) in which they will be used. Using two different domain definitions for the same data can require considerable changes to the application UML model.

Supporting Field-level Security

The Cúram client application enforces security at two levels: the page and the field. Page-level security depends on securing the server interfaces that represent the functions of the server application. Any UIM page that declares a server interface will not be displayed if the authenticated user is not authorized to access all of the server interfaces invoked from that page. Field-level security is enforced when a property of a server interface is accessed. It is permitted for a user to access a page even though the page contains some fields connected to server interface properties that the user is not authorized to view. In this case, the values of those secured fields should not be shown to the user. For example, a user may

be able to view the details of a person, but may not be authorized to view the salary of a person. The salary field may be presented on the person entity home page for all users, but if a user is not authorized to view the salary, the value of that field may be presented as a sequence of asterisks, **** instead of a monetary amount.

In the case of page-level security, the page is never rendered, so the renderers plug-ins will never be invoked. Therefore, page-level security is not a concern for the widget developer. In the case of field-level security, the renderer *is* invoked, so it is the responsibility of the widget developer to ensure that the renderer plug-in handles a field-level security violation appropriately. In the example given above, it is the renderer plug-in that produces the **** value instead of the monetary amount.

The field-level security violation is triggered when the renderer uses the `DataAccessor` to resolve a path to a server interface property that the active user is not authorized to access. The invoked method on the `DataAccessor` throws a `DataAccessSecurityException` instead of returning a value. If the renderer plug-in does not catch this exception and handle it, the rendering of the page fails and an error message is displayed. Where the required behavior is to display, say, **** instead of the secure value, the renderer must catch the exception and produce that value instead. The example below demonstrates this; the context is the `render` method and the `DataAccessSecurityException` class should be imported from the `curam.util.common.path` package.

```
String value;

try {
  value = context.getDataAccessor.get(
      field.getBinding().getSourcePath());
} catch (DataAccessSecurityException e) {
  value = "****";
}
```

Figure 51. Implementing Field-level Security

After the try... catch block, the *value* variable holds either the real value of the server interface property indicated by the field's source path, or ****, depending on whether or not the current user is authorized to access that server interface property. In either case, the value can be appended to the renderer's `DocumentFragment` to include it in the HTML response. The system is fail-safe. If the developer neglects to catch the security exception, then the page will not be rendered. If the developer catches the security exception, the secure value is never made available to the renderer class, so it is not possible for the developer to write code that would display the value accidentally.

The application security design should not expect to enforce field-level security on form pages. For example, a user may attempt to modify a person entity, but the user is not authorized to access the salary field. The user may see the salary text field on the person modification form initialized with the **** value. If the user submits the form, this literal value will overwrite the real salary value on the database. More likely, the user will see a validation error stating that **** is not a number. In that case, the user could enter any valid number and save it as the new salary value. In an edit-renderer plug-in, therefore, the developer should not catch the `DataAccessSecurityException` and simply allow the rendering of the page to fail. No secure information will be revealed in this case and the page can be secured at the page-level instead, preventing the user from viewing the page at all. If the user must be allowed to modify some of the details of the person, then the

option to modify the secured salary field should be presented on a different from from the one that provides the option to modify the unsecured fields. Field-level security, then, is a concern for view-renderer plug-ins, not edit-renderer plug-ins.

Adding New CSS Rules for Custom Widgets

When developing custom widgets, the developer is in complete control of the HTML that is generated for their custom widget and what CSS classes it references. The developer should ensure the CSS is as specific as possible to their widget. The developer must also be aware of how their widget can inherit styling from the Cúram application's default CSS without adding any custom CSS for the widget. The developer has two choices:

- Inherit - Without writing any custom CSS for the widget, default styling (e.g., color) will be applied due to the cascading and inheritance rules of CSS. Choosing this option will mean the widget is subject to changes from any future release of the Cúram application.
- Specific - If the widget has specific styling requirements then ensure they are explicitly defined in custom CSS for the widget. This will help to insulate the widget from changes to the default styling within the application. The recommended approach is to use the features provided by the Custom Widget Development Framework to generate a unique identifier for your widget and apply that to id attribute of the root element. All CSS rules for the custom widget can then be based off this identifier. Consult the Cúram Widget Development Guide for more details.

Every visual aspect (color, font size, borders, margin padding etc.) for a custom widget should be analyzed and the developer should decide on whether it should be inherited or specific. Also, it is impossible to guarantee there will never be impact on custom CSS, even if it is as specific as possible. As a guideline, it would be expected that with minor service pack releases of the Cúram application, the underlying HTML and CSS will not change drastically. However, a major release of the Cúram application may bring a new user interface and with it major changes to HTML structure and CSS. Even if a custom widget has specific CSS, it may need to be updated to adhere to the Cúram application's new look and feel.

Testing, Troubleshooting and Debugging

Introduction

Writing a widget's renderer plug-in class (or classes) is only half the battle. In the case of many widgets, particularly those that depend a lot on JavaScript and custom CSS styling, the battle has only just begun. The following sections provide some guidance on what to do next.

Testing

There are several aspects to the testing of widgets that pose different challenges to the developer or tester. The developer must:

- Test that the HTML produced by the renderer has the correct structure for all potential inputs.
- Test that the widget is presented correctly within the browser when the CSS styling has been applied.
- Test that any associated JavaScript operates correctly on the widget in the browser.
- Test the CSS and JavaScript across all supported browsers.

The best way to get started is to create a UIM page to host the widget. Sometimes, several test pages are required for the different use cases of the widget, though sometimes these can be combined in to a single UIM page. On building and running the application, open the page to check that the widget is presented correctly.

There are several testing tools available that can automate the process of checking the structure of the HTML produced by the widget. Tools such as Canoo WebTest can be run from Apache Ant build scripts and can be integrated into the build and test process. Alternatively, the structure can be checked manually by viewing the source of the HTML page.

Manual testing is required when checking that the HTML is presented correctly after the CSS styles are applied. This also has to be repeated in all browsers and versions of browsers that will be supported, as each browser has its own way of interpreting and implementing the CSS standards.

JavaScript, similarly, can behave differently in different browsers. Testing tools exist for testing both the JavaScript code directly and testing the behavior of the JavaScript with the browser environment. The performance of JavaScript code can also vary dramatically between different browsers. It is important to establish early on if any of the supported browsers may exhibit performance problems and to change the approach early in the development cycle if necessary.

Cross-browser support is often the most difficult aspect of renderer development to get right. When problems arise, search Internet forums and web sites for others who may have had the same problem. Sometimes there is an easy solution to the problem that would have taken a long time to figure out alone. However, sometimes there is no such magic bullet and compromises in the quality of the rendering on some browsers have to be accepted.

Troubleshooting

There are a number of common problems that arise during renderer development. The first place to start is with the error messages that are reported.

When an error occurs in a renderer, the rendering of the page fails and an error page is displayed. During development, it is very useful to enable the option to display the stack trace of the exceptions in a HTML comment within the error page. This option is normally turned off in production, but can be enabled by setting the errorpage.stacktrace.output property to `true` in the `ApplicationConfiguration.properties` file (described in the *Cúram Web Client Reference Manual*). Then, then an error occurs, view the source of the HTML page to see the embedded stack trace.

The exceptions reported in the stack trace are often deeply nested. The top of the stack trace will usually show a series of nested exception messages before displaying the first trace. This first series of error messages is often sufficient to diagnose the problem. Each error message is reported with an error number. Look up the error number in the *Cúram Web Client Error Message Guide* to find out what the error means and what the possible causes may be. Do not ignore these errors or dismiss them or fail to follow the resolution steps in the documentationthese errors are rarely ever misleading.

The domain and style configurations are a common source of issues. Naming clashes or incorrect assumptions about the component order can cause problems. If

a renderer simply does not seem to be invoked at all, check that it is correctly configured, that the configuration has the highest priority in the component order and that the application has been built after these changes have been made. Make sure, also, that the names of custom styles do not clash with existing style names.

A renderer plug-in class populates a DOM document fragment with the nodes that represent the HTML mark-up. At present, the CDEJ serializes the document fragment to XML text. This is compatible with the W3C XHTML 1.0 recommendation. However, some browsers are not fully compatible with XHTML and do not properly parse empty element tags, requiring instead separate opening and closing element tags with no body content. When an element node in the document fragment is serialized to XML text, an empty element tag is used when the element has no body content. To avoid parsing problems in the browser, it may be necessary to add some content to the body of the element to cause the serializer to generate separate opening and closing element tags. The simplest way to do this without affecting the presentation of that content is to add a comment node to the body of the element. The elements that cause the most problems are empty `div` elements and empty `script` elements. The browser may parse the page incorrectly, treating the empty element tag as an opening tag and nesting the following content incorrectly within that element. An indication that this has happened is when the view of the source for the HTML page in the browser does not match the view of the browser's DOM document (the parsed version of that source). The DOM document can be viewed with the web development tools available for most browsers. Adding a comment node to the empty element will resolve this issue.

Debugging

During the development of a Cúram client application, Apache Tomcat can be used within the Eclipse IDE to start and test the application. Renderer plug-in classes run in the context of the client application server and debugger breakpoints placed into the renderer plug-in class can be used to inspect the operation of the plug-in at run-time. When a breakpoint is not reached when expected, the problem may be with the debugging configuration of the IDE or with the configuration of the renderer. Add tracing code to the renderer to determine which problem exists. If the trace messages are displayed in the log, then the configuration is correct and the problem is with the configuration of the debugger. The configuration of the debugger is beyond the scope of this guide.

Trace messages can be written to the client application log easily from a renderer plug-in class. Simply print the messages to standard output or standard error using, for example, `System.out.println`. When running Tomcat from within the Eclipse IDE, the messages will appear in the console view of Tomcat process. Once the trace messages have been used to successfully diagnose and resolve a problem, they can be removed or commented out.

Much of the debugging effort of a complex widget lies not in the Java code of the renderer plug-in class, but in the JavaScript code or the CSS stylesheets. Issues in these areas can only be debugged within the browser. One effective approach to investigate such problems is to use the Mozilla Firefox[1]web browser with the Firebug[2]add-on. Firebug provides a host of tools for analyzing styling and layout, debugging JavaScript code, inspecting the DOM document, monitoring network activity and more. Firebug also allows changes to be made to the HTML page and the CSS style rules in real time, reducing the time it takes to test experimental

1. See the Mozilla web site for details.

2. See the Firebug web site for details.

changes. Beware, however, that Firefox may not render the content in the same manner as other browsers, such as Microsoft Internet Explorer. If Internet Explorer is the browser for which support is required, check regularly that changes that correct the presentation and operation of the widget in Firefox also work in Internet Explorer.

Configuring Renderers

Introduction

The customization of the configuration that associates edit-renderer and view-renderer plug-ins with named domain definitions, is supported in the Cúram application. This feature is merely an extension of the existing customization features, presented in the *Cúram Web Client Reference Manual*, where it describes how plug-ins can be developed for custom data conversion and sorting. That manual also describes the configuration process in detail. The two kinds of renderer plug-in are just to more kinds to add to the existing kinds of domain plug-in. They are configured in the same way and in the same configuration file. Examples are provided in this appendix, but the *Cúram Web Client Reference Manual* should be consulted for more details.

Component renderers are associated with styles, not domains, so these are configured separately. Styles only support a single kind of plug-in, a component-renderer, so their configuration, which very similar to the domain configuration, is simpler. Styles are not defined in the UML model like domain definitions; they are simply defined by naming them in the configuration file. The creation of custom configuration file for styles and the syntax for defining custom style configurations are described in this appendix.

The configuration process is one of customization, rather than full replacement. The CDEJ provides the default configuration. The developer adds custom configuration files to one or more application components. These custom configurations can override the CDEJ default configuration. As there can be many custom configurations in the application, one per component, these must be *merged* before they are used to customize the default configuration. Where specific domains or styles in the default configuration are not customized fully or at all, the default configuration is *inherited* for those domains and styles. The details of this merging and inheritance behavior for domains are described in the *Cúram Web Client Reference Manual*. This appendix provides additional information about the style configurations.

warning: *Purpose* -based Configuration

The developer may see domain and style configurations in the default CDEJ configurations that configure domains or styles using a `purpose` attribute instead of a `class` attribute. Configuration using purposes is more complex then configuration using named classes and custom configuration using purposes is not supported within the Cúram application; only class-based configuration may be used.

warning: Limitations on Kinds of Plug-ins

The CDEJ domain configuration specifies a kind of plug-in called a *select-renderer*. The development of custom select-renderer plug-ins is not supported in the Cúram application at this time. No further mention of them is made in this guide.

The configuration of *marshal* plug-ins for domains is also unsupported outside of the specific cases of the two marshal plug-ins for accessing XML data described in the samples of this guide and in more specific detail in "Extending Paths for XML Data Access" on page 73.

Any references to select-renderer or marshal plug-ins in the Javadoc for CDEJ, or information provided in the Javadoc about their development or configuration, does not constitute an authorization or offer of support for their use.

Several of the CDEJ renderers are defined in classes whose names include the word "Legacy". These are deprecated, transitional renderer classes and the referencing of these legacy renderer classes in custom configurations is not supported in the Cúram application. Note, also, that a *rendering cascade* will fail if it delegates the rendering of a field whose domain is associated with a legacy renderer. Developers must avoid rendering cascades that may result in the invocation of a legacy renderer.

Configuring Domain Renderers

The *Cúram Web Client Reference Manual* provides detailed information about the customization of the domain configuration in the `DomainsConfig.xml` file of an application component. That information is not repeated here. The view-renderer and edit-renderer plug-ins are configured in the same file and in the same way as other domain plug-ins. The only difference is that the specific plug-in names `view-renderer` or `edit-renderer` are used in the `plug-in` elements of the configuration. An example is shown below.

What are the basic principles? Configuration inheritance for domain renderers, no inheritance for component renderers (styles). What is the default configuration? Only configure what you need to change; do not copy complete configurations, otherwise expected inheritance can be compromised in the future.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <dc:domain name="SAMPLE_DOMAIN">
    <dc:plug-in name="view-renderer"
                class="sample.SampleViewRenderer"/>
    <dc:plug-in name="edit-renderer"
                class="sample.SampleEditRenderer"/>
  </dc:domain>

</dc:domains>
```

Figure 52. An Example of a `DomainsConfig.xml` File

It is possible to override all of the plug-ins associated with a domain (subject to some support limitations described in the previous section). However, it is very important that the developer only specify the plug-ins that need to be customized and not repeat the configuration of existing plug-ins without changing them. When the developer partially customizes a domain, any unspecified plug-ins will be resolved using the CDEJ default configuration or inherited from an ancestor domain of the configured domain. This is the preferred behavior.

Defining unnecessary custom configurations for plug-ins can have unwanted effects that may be hard to diagnose. For example, the developer might copy the CDEJ default configuration of a domain from the CDEJ default configuration file together with the configurations of *all* of that domain's plug-ins and use this as a template of sorts in the custom configuration file. The developer might now change only one `plug-in` element to customize the view-renderer class used for the

domain and leave all of the other plug-in elements copied from the CDEJ intact and unchanged. All of these unchanged plug-in configurations are unnecessary, as the developer is not customizing them. If the CDEJ is now upgraded, any changes to the CDEJ default configuration of that domain will not be reflected in the application, as the developer has, in the custom configuration, effectively customized all of the plug-ins for that domain. While using the older version of the CDEJ, this went unnoticed, as the customization was the same as the default. However, on upgrading the CDEJ, the old CDEJ configuration that the developer copied to the custom configuration file continues to be given priority and any new CDEJ default configuration of any plug-in will not be reflected in the application. It is very important, therefore, that the developer customize *only* the plug-ins that must change and omit all references to other plug-ins.

Configuring Component Renderers

Configuring styles with component-renderer plug-ins is similar to configuring domains with view-renderer and edit-renderer plug-ins. To configure styles, create a `StylesConfig.xml` file in the application component. An example styles configuration is shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<sc:styles
  <sc:style name="sample-style">
    <sc:plug-in name="component-renderer"
                class="sample.SampleComponentRenderer"/>
  </sc:style>

</sc:styles>
```

Figure 53. An Example of a `StylesConfig.xml` File

While the namespace and element names are different, the styles configuration file is similar in form to `DomainsConfig.xml`, but there is only one plug-in per style configuration.

There can be any number of `style` elements within the `styles` root element. Styles are defined by naming them in the configuration file; there is no need to model them or declare them anywhere else. Unlike a domain definition, the name of a style does not have to be a valid Java identifier; any non-empty string value that is not entirely composed of whitespace characters is acceptable.

On the `plug-in` element, the `name` is always `component-renderer` and the `class` is the fully qualified name of the Java class for the widget's component-renderer plug-in.

Where more than one `StylesConfig.xml` file exists in the application (there can be one in each component) and where the same style is defined more than once, the configuration for that named style from the highest priority component will be used. As styles do not form a hierarchy like domains, there is no inheritance behavior in the in the configuration.

Using the name of a style defined in the CDEJ default style configuration will override the configuration. However, the overriding of the CDEJ default styles is not supported in the Cúram application. Take care not to use the name of an existing CDEJ style, as the results may be unpredictable. To avoid accidental overrides, particularly if using generic style names like `label`, or `panel`, use a custom naming convention. For example, prefix style names with a string that represents an *ad hoc*, private namespace: `sample::label` and `sample::panel`. The

prefix `sample::` is not used by the CDEJ, so it can act like a namespace. The double colon has no special meaning in a style name and any separator character(s) can be used. If using this approach, it is best to choose a separator that is different from any separator used for words in the style name to avoid accidental name clashes.

Accessing Data with Paths

Introduction

Paths are references to sources of data. They are similar in concept to file system paths used to access files or XPath expressions used to access data in a structured document. All access to data of any kind from a renderer is performed via paths. Paths can be used to access the values of server interface properties, text in localized properties files, localized properties resources in the database and other values. The terminology used to describe the parts of a path is shown in the figure below.

Figure 54. The Anatomy of a Path

1. Prefix Path
2. Selector
3. Predicate
4. Step
5. Extended Path

The above path can be read as follows:

- The prefix path identifies the type of the data source. Here, `/data/si` indicates that it is a reference to the data of a server interface property.
- The following two path steps identify the name of the server interface (as declared in the UIM) and the full name of the property. Here, the `dtls$list$address` property of the `DISPLAY` server interface is referenced.
- A path step may have a selector or a selector followed by one or more predicates. The predicate is used to qualify the data identified by the path up to that point. Here, the predicate `[1]` is used to select the first address from the list of addresses in the property. Where predicates are used as numeric indexes, the index of the first value is one, as in XPath.
- An individual value of a server interface list property is selected by the first four steps of the path. The fifth step, `ADD1`, is the beginning of an extended path that

will be resolved, not by the DataAccessor, but by the domain marshal plug-in associated with the domain of the identified server interface property. Here, ADD1 may, if the marshal is the SimpleXPathMarshal described in "Extending Paths for XML Data Access" on page 73, select the value of an ADD1 element in an XML document that is the value of the server interface property.

For more information about the general structure of paths and their manipulation in code, refer to the Javadoc for the Path and Step interfaces in the curam.util.common.path package.

The Field object passed to a render method contains a Binding object that specifies a source path and/or a target path. Renderer plug-ins do not need to be concerned about the form of these paths, or what type of data sources they reference; renderer plug-ins only need to resolve these paths to their values and should do so without inspecting the paths or depending on then being in any particular form. It is this unquestioning processing of any given path that allows renderer plug-ins to be reused easily in many different contexts and in rendering cascades.

Renderer plug-ins resolve paths the DataAccessor object available from the RendererContext object passed to the render method. There are a number of DataAccessor methods that can be called. They all take a single path argument:

get(Path)
Gets the formatted text value of the data. For domain-specific data, this is the value *returned by* the format method of the converter plug-in for that domain.s

getRaw(Path)
Gets the raw value of the data. For domain-specific data, this is the value *passed to* the format method of the converter plug-in. The type of the value is also the same as the type returned by the parse method of the converter plug-in.

getList(Path)
Gets the list of formatted text values of the data.

getRawList(Path)
Gets the list of raw values of the data.

count(Path)
Gets a count of the number of values that will be returned by getList or getRawList.

Where the data is not domain-specific, such as the contents of a properties file, the getRaw method usually returns the same string value as the get method. Some data sources may only support a subset of these methods. The get method is always supported, but the getList, getRawList and count methods may not be supported for all data sources. There are other methods on the DataAccessor, but their use is not supported in the Cúram application.

Creating New Paths

For the most part, a renderer plug-in just resolves the values of the paths given to it in the Binding of its Field object. However, in some cases, the renderer requires data other than that referenced by the given paths. For example, a renderer may require a localized text value to use as a label within the HTML that it produces. In this case, the renderer must create a new path that references the required data and then resolve it to the required value.

New paths are created by extending one of the supported prefix paths. These prefix paths are defined by the ClientPaths class in the curam.util.client.path.util package. Each prefix refers to a different type of data source. Only a limited set of data sources for use in custom renderers are supported in the application. The supported prefix paths for those data sources are defined by these constants on the ClientPaths class:

GENERAL_RESOURCES_PATH
A reference to a localized text property within a Java properties file available on the classpath.

APP_PROP_RESOURCE_PATH
A reference to a localized text property within a Java properties file stored in the *Application Resource Store* in the database.

LITERAL_VALUE_PATH
A path encoding a literal value that can be resolved without reference to any external data source.

The prefix path is extended with further path steps to identify the required data. The forms of the paths required for each of the supported data sources are described in the following sections. The use of constants in ClientPaths, or their corresponding prefix path values, other than those listed above, are not supported in the Cúram application.

General Properties Resources

The general properties path refers to a localized text property stored in a Java properties file on the classpath. The prefix path is extended with two further steps: the first step is the resource identifier for the properties file; the second step is the property key. Java properties files can be added to any package within the javasource folder of an application component, the same location used for the renderer plug-in classes.

The resource identifier to use to locate the properties should correspond to the location of the properties resource on the classpath. For example, if the properties file X.properties is placed in a Java package sample.resources, after building the application it will be stored in a JAR file on the classpath as the file /sample/resources/X.properties. Then the resource name will the be sample.resources.X. See the Javadoc documentation for the standard java.util.ResourceBundle API for more information on the naming convention and mechanism used to locate the properties for properties files in more than one locale.

The example below shows how a renderer plug-in may retrieve the value of the age property from the PersonDetails.properties file in the sample Java package. The code is defined in the context of the render method. The localized text value is stored in the *ageLabel* variable ready to be added to the appropriate point of the HTML document.

```
Path agePath = ClientPaths.GENERAL_RESOURCES_PATH
    .extendPath("sample.PersonDetails", "age");
String ageLabel = context.getDataAccessor().get(agePath);
```

Figure 55. Accessing General Properties

Only the get method is supported when accessing general properties resources. If no such property can be found, the get method will throw a DataAccessException.

Path objects are immutable; they are similar to java.lang.String objects in that respect, or to the component objects described in "Overview of the Renderer Component Model" on page 50. Operations such as extendPath, do not modify the path, they return a new path (see the Javadoc for details). Therefore, if several properties are required from the same resource, a path can be created that includes the resource identifier step and then that path can be extended again and again to retrieve individual property values. This is shown in the example below, where the value of the *dtlsPath* variable is never changed by calls to extendPath after it has been initialized.

```
Path dtlsPath = ClientPaths.GENERAL_RESOURCES_PATH
    .extendPath("sample.PersonDetails");

DataAccessor da = context.getDataAccessor();

String ageLabel = da.get(dtlsPath.extendPath("age"));
String dobLabel = da.get(dtlsPath.extendPath("date.of.birth"));
String nameLabel = da.get(dtlsPath.extendPath("name"));
String addressLabel = da.get(dtlsPath.extendPath("address"));
```

Figure 56. Accessing Multiple General Properties

Where properties files are supplied for several locales, the properties file name will differ, but the path used to reference the property should not include the locale. For example, if the properties files PersonDetails_en_US.properties and PersonDetails_es.properties are defined in the sample package folder, the above code will not change; the resource identifier remains sample.PersonDetails. The DataAccessor will automatically determine the locale of the active user and select the correct properties resource. The usual locale fall-back sequence, described by the java.util.ResourceBundle API, is followed.

Resource Store Properties Resources

Files of any kind are allowed to be uploaded and stored in the database of the application, for later retrieval. This service is called the Application Resource Store. Once a file is uploaded, it no longer exists as a file, but as the value of a field in a database record. This database record is referred to as a *resource*. By constructing and resolving the appropriate path, a renderer plug-in can access property values from Java properties resources uploaded to this store.

The path form is a little different from the paths used for general properties files resources on the classpath, as it accommodates other path forms that are not supported in the custom renderers within the Cúram application. Also, as these are no longer properties files, there are differences in the way the resources are identified. Properties resources are loaded to a local cache when they are requested. The cache stores the properties in a form that optimizes locale fall-back operations and reduces memory usage through de-duplication, so the individuality of the original resources is lost. However, this results in an efficient system that is a good alternative to classpath-based properties resources, particularly where resources may need to be modified at run-time.

The path is created by extending the prefix path defined by ClientPaths.APP_PROP_RESOURCE_PATH. The extension adds a single step. The selector of the step is the name of the resource and a single predicate contains the name of the property key. The resource is identified using the name assigned to the resource when it was uploaded to the resource store. For example, if an administrator uploads the file PersonDetails.properties to the resource store and names the resource PersonDetails.properties, then that is the identifier that must

be used. The .properties name suffix (which is not a file extension, as a resource is not a file) is not added or removed by the system and must be used as the identifier of the resource. The name could be set to just PersonDetails, without any suffix, but adding the suffix may help to make the type of the resource more readily identifiable from its name when administering the resource store. Either way, the resource identified in the path should match the resource name in the resource store exactly. An example of the construction of a path to request the age property from the resource store resource named PersonDetails.properties is shown below.

```
Path agePath = ClientPaths.APP_PROP_RESOURCE_PATH
    .extendPath("PersonDetails.properties[age]");
String ageLabel = context.getDataAccessor().get(agePath);
```

Figure 57. Accessing Resource Store Properties

As with general properties resources, only the get method is supported when accessing general properties resources. If no such property can be found, the get method will throw a DataAccessException.

Where multiple properties resource values are required, the path to the resource can first be created with an empty predicate and then the value of the predicate can be set again and again using the applyIndex method of the Path interface. This method returns a new path each time, it does not modify the existing path. The *index* value is used to set the value of the first empty predicate encountered in the path. This is shown below.

```
Path dtlsPath = ClientPaths.APP_PROP_RESOURCE_PATH
    .extendPath("PersonDetails.properties[]");

DataAccessor da = context.getDataAccessor();

String ageLabel = da.get(dtlsPath.applyIndex("age"));
String dobLabel = da.get(dtlsPath.applyIndex("date.of.birth"));
String nameLabel = da.get(dtlsPath.applyIndex("name"));
String addressLabel = da.get(dtlsPath.applyIndex("address"));
```

Figure 58. Accessing Multiple Resource Store Properties

The locale fall-back operation depends on all the resources in the sequence having the same name. When resolving properties using the local fall-back mechanism, the CDEJ does not modify the name of the requested resource, it only changes the value for the separate locale field in the resource store record. This differs from the way the java.util.ResourceBundle API creates new file names when searching for locale fall-back resources. When a resource is uploaded to the store, both the name and the locale should be specified separately through the administration interface. If the files PersonDetails_en_US.properties and PersonDetails_es.properties are uploaded, the administrator should assign the same name PersonDetails.properties (or just PersonDetails, if preferred) to both resources, but set the separate locale field value to en_US and es, respectively. If no locale is specified, then the resource is treated as the ultimate locale fall-back resource, just as the ResourceBundle API would treat a properties file with no locale code appended to its name.

Literal Values

Occasionally, the developer may need to represent a literal value using a path, as the widget API usually only supports paths to represent data. For this purpose, the developer may encode a literal value within a path, so that when the DataAccessor

resolves the path, the literal value is returned. An example is shown below.

```
Path literalPath = ClientPaths.LITERAL_VALUE_PATH
    .extendPath(PathUtils.escape("a //literal// value"));
```

Figure 59. Encoding Literal Values

The literal value may contain characters that could be confused with the path syntax, so the value must be escaped when constructing the path. The `PathUtils` class in the `curam.util.common.path` package provides an `escape` method for this purpose. In the example, the method escapes the forward slash characters in the literal value and prevents them from being interpreted as separating path steps by the `extendPath` method. When the path is resolved using `DataAccessor.get`, the escaping will be reversed automatically, so there is no requirement on the consumer of the path to treat it differently to any other.

Extending Paths for XML Data Access

Introduction

A special domain marshal plug-in was used in many of the examples in this guide to access data from XML document using paths resembling XPath expressions. This appendix describes the supported path forms in more detail and provides additional information about the automatic data conversion capabilities.

This appendix refers to the structure of path values. See the Javadoc for the `Path` and `Step` interfaces in the `curam.util.common.path` package for an explanation of the terminology used here.

When the path from the Binding of a Field object is resolved, and where that path identifies a server interface property, the value returned is the value of the server interface property. If the path is *extended* with extra path steps, then the domain marshal plug-in class associated with the domain definition of that server interface property is invoked to evaluate the extra path steps with respect to the value of the server interface property. The examples in this guide show how this can be used to extract data from XML documents returned in server interface properties. Two domain marshal plug-in classes are provided with the out-of-the-box Cúram application for this purpose.

The `SimpleXPathMarshal` class supports the resolution of XPath-like expressions against data returned in a server interface property value. All values are returned as strings, just as they appear in the XML document. The `SimpleXPathADCMarshal` class adds the ability to apply automatic data conversion and formatting to the resolved string values. This class can be used without automatic data conversion, but it is a little more efficient to use the former class if data conversion is not required. Both classes are defined in the `curam.util.client.domain.marshal` package.

Simple XPath Expressions

The “simple” XPath expressions supported by these marshal plug-ins are not true XPath expressions, though they aim to be as similar as possible to a very small and simple subset of the location paths defined by the W3C XPath 1.0 recommendation.

The paths operate on a DOM document created by parsing the XML string that is returned as the value of a server interface property. Each step in the path selects one or more nodes in the document and subsequent steps are evaluated within the

context of each of those selected nodes. The context starts with the document node, so the first step will identify the root element of the document.

The selector of a step (that part of the step before the predicate) defines the name of the element or attribute to be selected. The prefix `@` is used to indicate an attribute name; an element name requires no prefix. An element name may be followed by a single, optional predicate with an integer index value (starting from one) or an attribute selection expression.

```
<values id="a1" locale="en">
  <value domain="SVR_INT32">1234</value>
  <value domain="SVR_DATE">20080131</value>
  <value domain="ADDRESS_DATA">
    <address>Apt. 86</address>
    <address>1000 Main St.</address>
    <city>Hometown</city>
  </value>
</values>
```

Figure 60. A Sample XML Document

For example, if the XML document has the form shown in "Simple XPath Expressions" on page 73, then the path `/values` selects the `values` root element; `/values/value[3]` selects the third `value` element within the `values` root element; `/values/value[@domain='SVR_DATE']` selects the `value` element with the `domain` attribute value `SVR_DATE` within the `values` root element; `/values/value[2]/@domain` selects the `domain` attribute of the second `value` element within the `values` root element; `/values/value` selects all three `value` elements within the `values` root element; `/values/value/@domain` selects the three `domain` attributes from the three `value` elements within the `values` root element; and the paths `/values/value[3]/address` and `/values/value/address` both select the two `address` elements of the third `value` element within the `values` root element. When more than one node is selected, the selected nodes are returned in the order in which they appear in the document.

An attribute value expression can be used to select elements that have an attribute with a particular value. An example was given above. The expression is limited to a single attribute name, prefixed with `@` followed by an equals sign and a quoted string value. The attribute name must be on the left-hand side of the equals sign only. The string can be quoted with single quotes or double quotes. If single quotes are used, then the string can contain double quotes and vice versa. The string cannot contain any `/`, `[` or `]` characters; it is intended to be used only for matching ID values or other simple identifiers.

The selector `*` selects any element and the selector `@*` selects any attribute. For example, the path `/values/value[3]/*` selects the two `address` elements and the `city` element of the third `value` element within the `values` root element; the path `/values/@*` selects the `id` and `locale` attributes of the `values` root element; the path `/values/*/@*` selects all of the attributes of all of the child elements of the `values` root element; the path `/values/value[3]/*[3]` selects the third child element of any name of the third `value` element within the `values` root element, the `city` element in the case of the document above.

There are a number of restrictions on the steps that can be used and on their positions in a path. Where an element or attribute name appears below, a `*` can replace it. The allowed forms are as follows (the examples refer to the above sample document):

element-name
: An element name identifying the elements to be selected within the context provided by the previous path step. For example, `/values` selects the `values` root node, while `/values/value` selects all three `value` elements within the `values` root element.

element-name **[** *index* **]**
: An element name and an integer index value identifying one of several elements with that name in the context provided by the previous path step. For example, `/values[1]` selects the first `values` element, which, as it is the root element and the only `values` element, selects the same element as the simpler path `/values`; `/values/value[2]` selects the second `value` element that is a child of the `values` root element.

element-name **[@** *attribute-name* **=** *quoted-string* **]**
: An element name and an attribute selection expression identifying elements with that name and with that value for the named attribute in the context provided by the previous path step. See above for more details.

@ *attribute-name*
: An attribute name identifying an attribute of the element or elements selected by the previous steps in the path. An attribute selection step is only allowed as the last step in a path unless it is followed by a single function step (described below).

For convenience, the following step form may also be used in leading steps or the terminal step:

element-name **[]**
: An element name followed by an empty predicate. This is treated in the same way as a simple element name. This is not a true XPath expression, but it is convenient for situations when a path has an empty predicate to which an index will later be applieda common scenario if all that is required is a count of the nodes.

A valid path may select zero or more nodes. The values returned for these nodes depend on which method of the `DataAccessor` was called from the renderer class. The details are provided in the next section.

The `Path` interface does not support the representation of full XPath expression. Notably, XPath function calls that accept location paths as arguments cannot be represented, so a non-standard notation is used to provide some basic functionality. Instead of an expression of the form *`function-name`* `(`*`location-path`*`)` , the form *`location-path`* `/` *`function-name`* `()` is used instead. For example, to the get the qualified name of the third child element of the third `value` element in the sample document above, the path would be `/values/value[3]/*[3]/name()`; this is treated as if it were the expression `name(/values/value[3]/*[3])`.

A function may only appear as the last step in a path. The supported functions are as follows:

name()
: Gets the qualified name of the first node selected by the path. This will be the element or attribute name including any namespace prefix.

local-name()
: Gets the name of the first node selected by the path. This will be the element or attribute name not including any namespace prefix.

Evaluating the Paths

Paths are evaluated using the DataAccessor object available from the RendererContext that is passed to all render methods. When a path is extended into a server interface property value, the method called on the DataAccessor determine the method that is called on the marshal plug-in. For the SimpleXPathMarshal plug-in class data is converted generally as follows:

- The value of an attribute node is the string value of the attribute.
- The value of an element node is the concatenation of the values of all of the child text nodes of that element.
- If the are no selected nodes or a path evaluates to null, the result depends on which DataAccessor method was called. See below for details.
- The value of the result of a function call, is the string value of that result.

This behavior is consistent with use of the standard XPath string() function on the selected nodes or value, except, in the case of an element node, where only direct child text nodes of an element are concatenated, not all descendant text nodes as would be normal for XPath.

The DataAccessor methods refine the general behavior described above. For the SimpleXPathMarshal plug-in class, there is little difference between the formatted and raw variants, except for their handling of null values.

get Gets the string value of a the first node (in document order) selected by the simple XPath expression given by the path, or, in the case of a function call, the string value of the result of that call. If no nodes are selected, the result will be an empty string. To distinguish between an attribute or element that is present but has an empty string value and an attribute or element that is not present at all, use the getRaw method and test if the result is an empty string or a null value.

getRaw Gets the first raw value of the first node (in document order) selected by the path, or, in the case of a function call, the resulting value of that call. If no nodes are selected, the result will be null.

getList
Gets the list containing the string values of the nodes (in document order) selected by the path. For non-function-call paths, the values in the list represent the result of calling the get method on each selected node. If the path represents a function call, then the list will contain the single result of calling the function ones on all of the selected nodes, not a list of the results of the function call on each node. The functions operate only on the first node when presented with a list of several nodes.

For example, /values/value[3]/* selects all of the child elements of the third value element within the values root element. The resulting list will contain the three string objects, one each for the body text of each element. However, evaluating the path /values/value[3]/*/name() will return a list containing a single string that is the name of the first selected element (addr), not one string for the name of each selected element.

getRawList
Gets the list containing the values of the nodes (in document order) selected by the path. The conversion behavior of this method is the same as the getRaw method and the list handling is the same as the getList method.

count Counts the number of nodes selected by the given path. If the path represents a function call, then the count is the number of results from the function call (usually one).

Automatic Data Conversion

The `SimpleXPathMarshal` class is useful when extracting simple string values from XML documents. However, much of the time, the values are merely the string representation of other data types, such as dates, numbers and code-table items. The `SimpleXPathADCMarshal` extends the capabilities of the `SimpleXPathMarshal` by enabling automatic data conversion (ADC) using the domain converter plug-ins. The same XPath location paths supported by the `SimpleXPathMarshal` are supported by this ADC class.

This `SimpleXPathADCMarshal` plug-in will perform automatic data conversion (ADC) on the values in the XML content. This requires that the XML content represents values in a particular form: the value must be the body content of an element and the element must have a `domain` attribute identifying the name of the domain definition to apply to the value. The values must use the *generic* string form of the data, to be compatible with the `parseGeneric` method of the domain converter plug-in associated with the identified domain. In general, the generic string value is the same as the result of calling Java's `toString` method on the corresponding Java object, with the exception of date and data-time values, where the ISO 8601 *basic format* is used. ADC cannot be applied to the values of attributes or the results of XPath function calls, only to the body text of elements; however, attributes can still be used for values if ADC is not required.

Generic String Values: The generic string value of a server interface property is used to represent numbers, dates, date-times and other values unambiguously in string form when it is not possible to represent them using a more suitable Java object representation. The generic string value in some of the domain definition options in the application UML model and when transporting data in XML documents. The format avoids problems that may arise if values were formatted according to the rules or conventions of different locales, as these would add unnecessary complication and need to be communicated.

For numbers, the generic string representation must omit grouping separator characters (such as thousands separators), use only a period character (Unicode "FULL STOP" U+002E) as a decimal separator and, if the number is negative, place the minus sign character (Unicode "HYPHEN-MINUS" U+002D) on the left. The CDEJ is lenient when parsing numeric values that use a comma as a thousands separator, but these are best avoided. Using the `toString` method of class used for the Java object representation of numeric domain definitions will produce the desired result. The classes used for the Java object representations for all of the base domain definitions are listed in the *Cúram Web Client Reference Manual*.

Date and date-time values must be formatted using ISO 8601 basic format. ISO 8601 basic format represents date and date-time values as fixed-length character strings. The format for date values is *YYYYMMDD* , two-digit years are not allowed. The format for date-time values is *YYYYMMDD* `T` *hhmmss* , the `T` is a literal character denoting the start of the time value and the time uses the 24-hour clock. The `parseGeneric` method assumes the date-time values are in the UTC time zone. The active user's time zone will be applied when formatting the value for display.

Without ADC, the formatted values and raw values that are returned by the getter methods are both the literal string values that are retrieved from the XML

document (with only a difference in the handling of null values). With ADC, the formatted values are the values formatted according the the locale of the active user and the raw values are the Java object representations of those values appropriate for the indicated domain.

For example, with reference to the document in "Simple XPath Expressions" on page 73, if the path `/values/value[1]` is passed to the `get` method, then the result will be the string string `1,234` if the user's locale is, say, `en`, where a comma is used as a thousands separator. Similarly, if the path is `/values/value[2]`, then the result will be `31-Jan-2008` if the user's locale is en and if that particular date format is set. For raw values, the effect is similar, but the corresponding Java object will be returned instead of a formatted string. For example, it will be a `java.lang.Integer` for the SVR_INT32 domain, or a `curam.util.type.Date` for the SVR_DATE domain. Date and date-time values should be in the UTC time zone, they will be converted to the user's time zone when formatted.

Source Code for the Sample Widgets

Source Code for the E-Mail Address Widget

```
public class EMailAddressViewRenderer
       extends AbstractViewRenderer {

  public void render(
        Field field, DocumentFragment fragment,
        RendererContext context, RendererContract contract)
        throws ClientException, DataAccessException,
               PlugInException {

    String emailAddress = context.getDataAccessor()
       .get(field.getBinding().getSourcePath());

    Document doc = fragment.getOwnerDocument();

    Element span = doc.createElement("span");
    span.setAttribute("class", "email-container");
    fragment.appendChild(span);

    Element anchor = doc.createElement("a");
    anchor.setAttribute("href", "mailto:" + emailAddress);
    span.appendChild(anchor);

    Element img = doc.createElement("img");
    img.setAttribute("src", "../Images/email_icon.png");
    anchor.appendChild(img);

    anchor.appendChild(doc.createTextNode(emailAddress));
  }
}
```

Source Code for the Photograph Widget

```
public class PhotoViewRenderer extends AbstractViewRenderer {

  public void render(final Field component,
      final DocumentFragment fragment,
      final RendererContext context,
      final RendererContract contract)
      throws ClientException, DataAccessException,
             PlugInException {
    String personID
         = context.getDataAccessor().get(component.getBinding()
            .getSourcePath().extendPath("photo/id"));
    String personName = context.getDataAccessor()
        .get(component.getBinding()
            .getSourcePath().extendPath("photo/name"));

    Document doc = fragment.getOwnerDocument();

    Element rootDiv = doc.createElement("div");
    rootDiv.setAttribute("class", "photo-container");
    fragment.appendChild(rootDiv);

    Element linkDiv = doc.createElement("div");
    linkDiv.setAttribute("class", "details-link");
    rootDiv.appendChild(linkDiv);

    Element anchor = doc.createElement("a");
    anchor.setAttribute("href", "Person_homePage.do?"
                               + "id=" + personID);
    linkDiv.appendChild(anchor);

    Element anchorImg = doc.createElement("img");
    anchorImg.setAttribute("src", "../Images/arrow_icon.png");
    anchor.appendChild(anchorImg);

    Element photoDiv = doc.createElement("div");
    photoDiv.setAttribute("class", "photo");
    rootDiv.appendChild(photoDiv);

    Element photo = doc.createElement("img");
    photo.setAttribute("src",
      "../servlet/FileDownload?"
      + "pageID=Sample_photo"
      + "&id=" + personID);
    photoDiv.appendChild(photo);

    Element descDiv = doc.createElement("div");
    descDiv.setAttribute("class", "description");
    descDiv.appendChild(doc.createTextNode(personName));
    rootDiv.appendChild(descDiv);
  }
}
```

Source Code for the Details Widget

```
public class PersonDetailsViewRenderer
       extends AbstractViewRenderer {

  public void render(
    Field field, DocumentFragment fragment,
    RendererContext context, RendererContract contract)
    throws ClientException, DataAccessException, PlugInException {

    String name = context.getDataAccessor().get(
      field.getBinding().getSourcePath()
                            .extendPath("/details/name"));
    String reference = context.getDataAccessor().get(
      field.getBinding().getSourcePath()
                            .extendPath("/details/reference"));
    String address = context.getDataAccessor().get(
      field.getBinding().getSourcePath()
                            .extendPath("/details/address"));
    String gender = context.getDataAccessor().get(
      field.getBinding().getSourcePath()
                            .extendPath("/details/gender"));
    String dateOfBirth = context.getDataAccessor().get(
      field.getBinding().getSourcePath()
                            .extendPath("/details/dob"));
    String age = context.getDataAccessor().get(
      field.getBinding().getSourcePath()
                            .extendPath("/details/age"));
    String phone = context.getDataAccessor().get(
      field.getBinding().getSourcePath()
                            .extendPath("/details/phone"));
    String email = context.getDataAccessor().get(
      field.getBinding().getSourcePath()
                            .extendPath("/details/e-mail"));

    Document doc = fragment.getOwnerDocument();

    Element detailsPanelDiv = doc.createElement("div");
    detailsPanelDiv.setAttribute("class",
      "person-details-container");
    fragment.appendChild(detailsPanelDiv);

    Element div;
    Element image;

    div = doc.createElement("div");
    div.setAttribute("class", "header-info");
    div.appendChild(doc.createTextNode(name));
    div.appendChild(doc.createTextNode(" - "));
    div.appendChild(doc.createTextNode(reference));
    detailsPanelDiv.appendChild(div);

    div = doc.createElement("div");
    div.appendChild(doc.createTextNode(address));
    detailsPanelDiv.appendChild(div);

    div = doc.createElement("div");
    div.appendChild(doc.createTextNode(gender));
    detailsPanelDiv.appendChild(div);

    div = doc.createElement("div");
    div.appendChild(doc.createTextNode("Born "));
    div.appendChild(doc.createTextNode(dateOfBirth));
    div.appendChild(doc.createTextNode(", Age "));
    div.appendChild(doc.createTextNode(age));
    detailsPanelDiv.appendChild(div);

    div = doc.createElement("div");
    div.setAttribute("class", "contact-info");
    detailsPanelDiv.appendChild(div);
    image = doc.createElement("img");
    image.setAttribute("src", "./images/phone_icon.png");
    div.appendChild(image);
    div.appendChild(doc.createTextNode(phone));
```

Source Code for the Person Context Panel Widget

```
public class PersonContextPanelViewRenderer
       extends AbstractViewRenderer {

  public void render(final Field component,
      final DocumentFragment fragment,
      final RendererContext context,
      final RendererContract contract)
      throws ClientException, DataAccessException,
             PlugInException {
    ContainerBuilder cb
      = ComponentBuilderFactory.createContainerBuilder();
    cb.setStyle(context.getStyle("horizontal-layout"));

    FieldBuilder fb
      = ComponentBuilderFactory.createFieldBuilder();
    fb.copy(component);
    fb.setDomain(context.getDomain("SAMPLE_PHOTO_XML"));
    fb.setSourcePath(
      component.getBinding().getSourcePath()
               .extendPath("person"));
    cb.add(fb.getComponent());

    fb.setDomain(context.getDomain("SAMPLE_DTLS_XML"));
    fb.setSourcePath(
      component.getBinding().getSourcePath()
               .extendPath("person"));

    cb.add(fb.getComponent());
    DocumentFragment content
      = fragment.getOwnerDocument().createDocumentFragment();
    context.render(cb.getComponent(), content,
                   contract.createSubcontract());
    fragment.appendChild(content);
  }
}
```

Source Code for the Horizontal Layout Widget

```
public class PersonContextPanelViewRenderer
       extends AbstractViewRenderer {

  public void render(final Field component,
      final DocumentFragment fragment,
      final RendererContext context,
      final RendererContract contract)
      throws ClientException, DataAccessException,
             PlugInException {
    ContainerBuilder cb
      = ComponentBuilderFactory.createContainerBuilder();
    cb.setStyle(context.getStyle("horizontal-layout"));

    FieldBuilder fb
      = ComponentBuilderFactory.createFieldBuilder();
    fb.copy(component);
    fb.setDomain(context.getDomain("SAMPLE_PHOTO_XML"));
    fb.setSourcePath(
      component.getBinding().getSourcePath()
               .extendPath("person"));
    cb.add(fb.getComponent());

    fb.setDomain(context.getDomain("SAMPLE_DTLS_XML"));
    fb.setSourcePath(
      component.getBinding().getSourcePath()
               .extendPath("person"));

    cb.add(fb.getComponent());
    DocumentFragment content
      = fragment.getOwnerDocument().createDocumentFragment();
    context.render(cb.getComponent(), content,
                   contract.createSubcontract());
    fragment.appendChild(content);
  }
}
```

Source Code for the Text Field Widget with No Auto-completion

```
public class NoAutoCompleteEditRenderer
       extends AbstractEditRenderer {

  public void render(
        Field field, DocumentFragment fragment,
        RendererContext context, RendererContract contract)
        throws ClientException, DataAccessException,
               PlugInException {

    String title = getTitle(field, context.getDataAccessor());
    String targetID = context.addFormItem(field, title, null);

    boolean useDefault = !"false".equalsIgnoreCase(
        field.getParameters().get(FieldParameters.USE_DEFAULT));
    String value = context.getFormItemInitialValue(
        field, useDefault, null);

    Element input = fragment.getOwnerDocument()
        .createElement("input");
    fragment.appendChild(input);

    input.setAttribute("type", "text");
    input.setAttribute("autocomplete", "no");
    input.setAttribute("id", targetID);
    input.setAttribute("name", targetID);

    if (title != null && title.length() > 0) {
      input.setAttribute("title", title);
    }

    if (value != null && value.length() > 0) {
      input.setAttribute("value", value);
    }

    if ("true".equals(field.getParameters()
                        .get(FieldParameters.INITIAL_FOCUS))) {
      input.setAttribute("tabindex", "1");
    }

    String width
        = field.getParameters().get(FieldParameters.WIDTH);
    if (width != null && width.length() > 0
        && !"0".equals(width)) {
      String units;
      if ("CHARS".equals(field.getParameters()
                          .get(FieldParameters.WIDTH_UNITS))) {
        units = "em";
      } else {
        units = "%";
      }
      input.setAttribute("style", "width:" + width + units + ";");
    }

    setScriptAttributes(input, field);
  }
}
```

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing

Legal and Intellectual Property Law.

IBM Japan Ltd.

19-21, Nihonbashi-Hakozakicho, Chuo-ku

Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

© Copyright IBM Corp. 2012, 2014

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Dept F6, Bldg 1

294 Route 100

Somers NY 10589-3216

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources.

IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Privacy Policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies or other similar technologies that collect each user’s name, user name, password, and/or other personally identifiable information for purposes of session management, authentication, enhanced user usability, single sign-on configuration and/or other usage tracking and/or functional purposes. These cookies or other similar technologies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and

IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/us/en/copytrade.shtml.

Apache is a trademark of Apache Software Foundation.

Microsoft and Internet Explorer are trademarks of Microsoft Corporation in the United States, other countries, or both.

Firefox is a registered trademark of Mozilla Foundation.

Java and all Java-based trademarks and logos are registered trademarks of Oracle and/or its affiliates.

Other names may be trademarks of their respective owners. Other company, product, and service names may be trademarks or service marks of others.

Printed in USA