Merative Social Program Management 8.1

Cúram Custom Widget Development Guide

Note

Before using this information and the product it supports, read the information in Notices on page 93

Edition

This edition applies to Merative™ Social Program Management 8.0.0, 8.0.1, 8.0.2, 8.0.3, and 8.1.

Contents

1 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.

1.1 Overview

The objective of the 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 that are used throughout the guide are intentionally blurred because you 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 the section is to explain briefly what widgets are, what can be achieved through the customization of widgets and how the rest of the guide is structured to aid the developer in the task of developing custom widgets.

This is a guide for client application developers who want to customize the presentation of Social Program Management application pages in ways that are not possible through UIM or through the reconfiguration of the set of widgets that are provided in the Social Program Management.

Prerequisites

The developer is proficient in Social Program Management 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 that is 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.

From Cúram 6.0 onwards, support is introduced for the customization of *widgets*. Widgets are the elements of the user interface that is used to present the values of the fields that are 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;

- The presentation of all email address values can be customized so that, instead of being shown as simple text, they are shown as HTML `mailto:` links beside an email icon;
- A photograph of a person who is stored in the application database can be displayed as the value of a field;
- The details of a person can be presented by using a richer and more compact layout than possible with a UIM `CLUSTER`;
- Widgets can be reused within other widgets, so that the email 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 Social Program Management application page by using UIM, but the page is displayed in a user's web browser by using HTML.

Behind the scenes, the CDEJ translates the `CLUSTER` and `LIST` elements of the UIM page into HTML elements and then presents or *renders* the labels and values of the `FIELD` elements within the structure that is provided by those HTML elements. Typically, the CDEJ renders a cluster or list by using an 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 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.

Social Program Management 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 Social Program Management widgets. The development and configuration of such custom widgets is the subject of the guide.

Outline of this Guide

The next section, 1.2 Approaches to Customization on page 14 guides the developer on the choice of approach to achieving the required customization of the user-interface while the development effort is minimized.

1.3 How Widgets Work on page 19, presents more detailed information about the components of a widget and their configuration.

1.4 An EMail Address Widget on page 23 introduces the fundamental principles of the widget development process and the subsequent widget configuration. The section shows how to create a simple widget that presents an email address more appealingly in the context of a typical UIM page.

1.5 The Sample Context Panel Widgets on page 27 presents some samples of context panels that are used within the tabbed user interface. These sample context panels are constructed by 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 sections. Each section introduces new concepts in widget development that build upon what is gone before until the complete context panels are created and configured.

All of the widgets that are described to that point are used to present read-only values. 1.9 A Text Field Widget with No Auto-completion on page 46 introduces a widget for editing values on a form page. Widgets that are 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, when the user submits a form page that contain the widget, the entered field value reaches its destination on the server interface and that any validation errors are handled correctly.

Often, the deployed Social Program Management 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. 1.10 Internationalization and Localization on page 52 and 1.11 Accessibility Concerns on page 54 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, are 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 sections at the end of this guide supplement these other sources where they lack specific information that is related to widget development. Throughout the guide, the developer's attention is drawn to the relevant section.

Conventions of this Guide

For clarity, the source code that is presented throughout the guide is abridged. Import statements are omitted and package names are not shown.

1.18 Source Code for the Sample Widgets on page 85 provides the full, unabridged source code listings that show 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 that relates to the configuration of the widget presented. The real configuration file within an application component typically contains all of the configuration entries for all of the domain definitions to which customizations are applied.

Limitations and Restrictions

The focus of the 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 that is 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 that is offered for use of custom widgets in contexts other than those contexts presented in this document.

Throughout the guide, other limitations or restrictions are highlighted in the relevant contexts.

1.2 Approaches to Customization

Use this section to understand when UIM is 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 is.

Prerequisites

A basic knowledge of the capabilities of UIM and the structure of web pages that are 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 approaches doing it all with UIM or doing it all with a widget, there 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 requirements. This is so if the appearance of the content needs to be kept consistent with content that is 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 can 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 continues to work when the Cúram application is upgraded. Therefore, while a custom widget might present all of the page

content, it is 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 that is listed here 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 approaches here.

Using Only UIM

Before the developer decides to develop a custom widget, the developer first assesses if the required presentation can be achieved by using the layout and styling capabilities that are supported by UIM. If the presentation requirement can be achieved by using only UIM, there is 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 be used to achieve 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 that is 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 can identify a UIM-only solution to the presentation requirement, but might need to apply this solution to many pages. Doing this one page at a time might 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 might 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 or to make minor styling changes to the content (or both), then this approach of using only UIM might 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 might also be insufficient. In those cases, a more advanced approach might 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 that is used for a particular type of data to be a different standard widget. Where custom widgets are 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 that is 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, causes 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 section and the configuration process is covered in detail in 1.15 Configuring Renderers on page 72. Also described in that section 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 might 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 that are 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 be satisfied only 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.

1.4 An EMail Address Widget on page 23 explains how to develop a simple widget for viewing the value of a field; 1.9 A Text Field Widget with No Auto-completion on page 46 explains how to develop a simple widget for editing the value of a field. Both sections describe briefly how to configure these widgets and more information about the configuration of custom widget can be found in 1.15 Configuring Renderers on page 72.

In the simple case, a widget replaces the HTML content that is produced for the value of a UIM `FIELD` within the context of a normal UIM `CLUSTER` or `LIST`. The value of the field is still a single string, number, or date, only styled more elaborately. The general layout of the page is not 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 are 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 becomes.

Some indicators of increased complexity are:

- The value of the field can be more than a simple string or numeric value. For example, the value can be an XML document that contains several separate values, such as the data for a bar chart.
- Multiple values can be presented to the user differently from the usual grid layout of a cluster or list. For example, a photograph of a person can be presented with the person's name below the image and with no field label to the side.
- A widget can 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 can be an XML document that contains all of those details. A single widget is started by the CDEJ for that XML document value. That widget can 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 the contents of the cells in the grid layout that is presented by a UIM cluster are rendered.

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 is used only if the presentation requirement is such that it cannot be achieved by using only UIM, or by using a combination of UIM and one or more simple widgets.

Even if a presentation requirement *can* be met by using only UIM, the developer can prefer to use a custom widget to allow the customization to be applied automatically to many application pages, through 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 can be necessary.

This guide presents the development of several complex widgets in later sections. The developer does 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 always opts 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 control over the layout of more data. If a presentation requirement cannot be met by using only UIM, the developer can need to create a custom widget. However, the complexity can be reduced by developing only the widgets that are necessary and using UIM as much as possible to achieve the goal. The developer assesses if a combination of UIM with several simple widgets might achieve the wanted 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 can then associate simple custom widgets with individual fields, replacing the default HTML content for those fields with custom content. Further, the developer can 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 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 can identify that some widgets might 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 satisfy only 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 section presents 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 evolve with each new release. Widgets that attempt to emulate the output that is produced by standard elements of the Cúram user interface, such as clusters and lists, 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 is 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 that is required to achieve a consistent look across all web browsers is not 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 cannot have a free hand to implement all presentation requirements as specified. Most jurisdictions implement regulations and guidelines that require 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 can 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 that is 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 does not underestimate the effort that is 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 sections and advises on how they can be addressed. By choosing the simplest approach to achieve a presentation requirement after evaluating if the presentation requirement can be modified to permit a simpler approach, the widget developer can minimize the effort that is required to meet all of these added responsibilities.

1.3 How Widgets Work

A developer defines a Social Program Management application page by using UIM. However, the page is displayed in a user's web browser by using HTML, as described in the previous section. 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 can see the value that is 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 can see the value that is 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 started 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* " section of the *Cúram Web Client Reference Manual* and further in this guide in 1.15 Configuring Renderers on page 72.

When a complete UIM page is rendered at runtime, 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 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 uses a widget so configured whenever it needs to render the value of a field with a matching mode and domain definition.

The configuration that is used by the CDEJ to associate widgets with domain definitions is the same configuration that is 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* " section 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. The widget developer can also change how the values of fields are presented. To change the field value presentation, the widget developer must first understand the relationship between widgets and domain-specific plug-ins.

Prerequisites

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

Anatomy of a Widget

To a user, a widget is just what is shown in the web browser. To a widget developer, a widget comprises all the resources that are involved in the generation and presentation of what a user sees. From this development perspective, a widget can be composed 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 that is 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 by 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 subfolder, but the Java package name must 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 not required or 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* might be synonymous to a developer. However, most widgets require more resources, and sometimes more 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 that is 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 that is 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 exist. The configuration process is covered as required in the other sections of this guide and in more detail in 1.15 Configuring Renderers on page 72.
- **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 by 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 start the functions that are defined in that resource. External JavaScript resources can be placed into the application component. They are copied into the correct location during the build.

- **Images**
  Images can be included by embedding an 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 sections 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 added to the application component. Unlike JavaScript and image resources, CSS resources are not requested explicitly by a renderer plug-in. The style rules that are 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 are applied nonetheless. See the *Cúram Web Client Reference Manual* for more details on the incorporation of custom CSS resources.
- **Localized Text Properties**
  Any text that is produced by a renderer plug-in other than the actual field value is required to be internationalized, that is, 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 1.10 Internationalization and Localization on page 52.

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 depends 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 is not supported in the resolution of any issues that are related to the use of artifacts, or types of artifact, not explicitly covered in the later sections of this guide.

How Widgets Work In Depth

As explained in previous sections, widgets are selected and started 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 by 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 type of data and mode of operation, the same renderer plug-in class is started 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 is used. Thus, the developer can produce any wanted custom HTML mark-up to present the data of any UIM `FIELD` and see the mark-up applied consistently across the application.

The same widget is often used for many different types of data in a mode. For example, the application presents most of view-only data by using a single widget that 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 that are applied.

The CDEJ starts widgets in the course of transforming a UIM page to HTML. For widgets associated with UIM `FIELD` elements, this always happens at runtime. During the rendering of the page, the CDEJ constructs a `Field` object from the information that is 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 that is provided by the `Field` object, the renderer plug-in uses the DOM Core API to create the DOM nodes that represent the required HTML and field value and adds these nodes to the `DocumentFragment` object. When the renderer plug-in returns, the CDEJ takes the now populated `DocumentFragment` object, serialize it to an HTML text stream, and add this to the stream that is 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 starts 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 populate the fragments with HTML nodes and the first renderer can add the contents of those fragments to its own before control is returned 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 are presented in later sections of the guide.

The configuration file, which is 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 Social Program Management 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 configuration that is 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 1.15 Configuring Renderers on page 72. Crucially, the configuration that is 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 Social Program Management application and that are described at the relevant points in this guide.

When a custom widget controls most of the page content, often 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 started by the domain-specific renderers when necessary. Styles and component-renderer plug-ins are covered in 1.8 Tying Widgets Together in a Cascade on page 40.

1.4 An EMail Address Widget

The presentation requirements of many pages can be satisfied with simple UIM pages that contain 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. The section shows how the email address can be enhanced instead of presenting it as plain text. A link is added to allow the user to click the address and open their email software and also an icon is added.

The objective of this section is 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.

Defining the HTML

By default, string values are presented in the Social Program Management application, such as email addresses, without any HTML mark-up. The string value is added to the HTML page in the appropriate location.

The email address widget must produce HTML in the following form for an email 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 Email Address Widget

The HTML here is formatted for clarity, but it is generated without any indentation or line breaks, as this punctuation is not necessary for the browser to present the email address properly and increase only the size of the page.

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

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

Figure 2: Custom CSS for the Email Address Widget

The CSS vertical-align style applies only to the `img` element. It ensures that the email address text that is shown to the user lines up with the centerline of the text, rather than the baseline. This looks more appealing. The same styling goal might 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 by using different CSS selectors without the need to change the HTML structure that is 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 Renderer API defines the `DomainRenderer` interface that is used when the renderer plug-in classes are written, such as for the email 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 that is 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 email address widget produces a read-only value, so it is presented by using a view-renderer plug-in based on the `AbstractViewRenderer` class. The developer places the *`EMailAddressViewRenderer.java`* source file in the *`sample`* package subfolder 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 in the Javadoc documentation that is 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 that is 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 that represent the data and its HTML mark-up to this fragment. The system automatically serializes 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 started. It includes facilities to delegate rendering to other renderers, to resolve the data that is identified by the paths that are 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 that is described later in the guide.

See the Social Program Management 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 started 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 1.9 A Text Field Widget with No Auto-completion on page 46). Paths can represent values other than server interface properties. The developer is not 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 1.16 Accessing Data with Paths on page 75. The code to retrieve the email address string value is shown here.

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

Figure 4: Getting the Email Address Value

The source path is retrieved from the field's binding and passed to the `get` method of the data accessor that is retrieved from the context. The source path never is null for a view-renderer plug-in. The `get` method returns the value of the (in this case) server interface property. The value is formatted to a string representation appropriate for the active user. This formatting is performed by using the `format` method of the `DomainConverter` plug-in that is associated with the domain of the server interface property. The formatting of an email address value is trivial (the value is returned as is). However, other values, such as dates and date-times must be formatted by using the active user's locale, time zone, and date format. Regardless of the type of the underlying data, this is all handled automatically by the converter plug-ins. The returned string is 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 email 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 are created as needed and the attribute values and body content is escaped automatically.

All content that is created by 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 by 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 here (see Source Code for the E-Mail Address Widget on page 86 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 Social Program Management 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 email address widget, the data must be in a domain that is specific to email addresses. Here, the SAMPLE_EMAIL_ADDR domain is assumed. The *`DomainsConfig.xml`* file is added to the client application component, or the existing file is modified if it 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 configuration here, the view-renderer of the custom widget is now started 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, is inherited and used . Typically, this is the associated `TextEditRenderer` by default with the SVR_STRING domain.

More information about configuring renderers and other plug-ins is provided in 1.15 Configuring Renderers on page 72.

1.5 The Sample Context Panel Widgets

The previous section presented the main steps that are required to develop a simple custom widget and the artifacts that are required for its operation. Simple custom widgets, such as the email address widget, are often sufficient to meet presentation requirements. They can also be used in the context of more complex widgets. In the section, two such complex widgets is introduced. The following sections 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 are developed first in isolation. Each part is a widget in its own right and is configured for use on its own before the next part is introduced. When the parts are essentially complete, they are combined by using new renderer classes that delegate the rendering of these parts to form the full sample widgets. Later sections then show how issues such as text localization, locale-specific data formatting, and accessibility compliance can be addressed.

Prerequisites

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

The Sample Widgets

The first sample widget is a context panel that provides details about a person. The widget has two parts. The first part presents a photograph of the person above their name and an icon provides a hyperlink from the photograph to the home page of that person. The second part displays details about that person by using text with elaborate styling and icons.

The development of this context panel shows how these two parts can be created and used independently and how they can also be combined into a single widget. In the cases of both of these parts, the content and layout requirements cannot be met by using ordinary UIM pages.

The widget displays the following details:

- Photograph
- Icon links to person's home page
- Name
- ID
- Address
- Gender

- Date of birth
- Telephone number
- Email

The photograph widget introduces XML-based data sources and the use of the `FileDownload` servlet to deliver images to the web browser. At first, the details widget, demonstrates a more complex example of a widget that is backed by XML data. Later, the details widget is used to show how the email address widget developed in 1.4 An EMail Address Widget on page 23 can be reused through delegation to present the email address value. Also how the text can be localized and how locale-specific formatting can be applied to the date of birth value.

The second sample widget is a person list widget, which is another context panel widget. The widget displays a list of people by 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 that is developed for the first sample are reused to create this new context panel widget. However, this time the person's name is presented in a different way in the details panel and the ID number is omitted. This kind of reuse is more complex than the reuse of a simple email address renderer.

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 demonstrates further complex rendering techniques.

1.6 A Photograph Widget

The photograph widget displays a photograph of a person in the current context with their name and a link to an associated details page.

1.4 An EMail Address Widget on page 23 described how to access a single source value (the email address) and generate HTML markup to provide a more aesthetically pleasing representation of an email 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. The section shows how multiple source values can be combined and accessed by the widget.

The section also shows 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 an 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.

The objectives of the section are:

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

Prerequisites

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

Defining the HTML

As shown by the screen capture, 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 CSS rules for all content within the widget are defined. In this case, the root `div` element is given the `photo-container` class name. There are three child `div` elements that contain the link, the photo, and the person name. Each of these is also 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 is described in the section.

```
<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 7: HTML Output of the Photo Widget

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

Based on the screen capture, 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 that are 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 8: Custom CSS for the Photo Widget

The class name of the root `div` element is used when all CSS rules are defined to ensure that 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 that there is even spacing on each side of the image. The remaining CSS classes 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 extra style on the `description` element to set its font.

Defining Data in XML Form

The previous sections 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 not sufficient. It is often preferable for the value to be an XML document that contains all of the data that is 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 specify only one SOURCE connection to the required data, both the ID and the name must be passed back in a single-server interface property. The Social Program Management application provides support classes that make it simple to access data expressed as an XML document, so an XML document that contains the values is the preferred form when data is combined into a single-server interface property.

Here is a sample of an XML document that represents all of the information that is required to present the photograph of a person. The `id` element defines the concern role ID value that is passed to the `FileDownload` servlet by using the *id* parameter that is 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 that are provided with the Social Program Management application, the values are 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 9: An XML Document Describing a Photograph

Defining the Renderer Class

The skeleton renderer class for the photograph widget is shown here. The class extends the same base class as the email address widget, as it also is a view renderer. The class is 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 10: The Renderer Class for the Photograph Widget

Accessing Data in XML Form

For the photograph widget, the source value is no longer a simple string, instead it is an XML document. The approach that is used for the email address widget needs to be extended to allow values that are 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 here.

1.4 An EMail Address Widget on page 23 described how to access a single source value by using a `Field` object, its `Binding` property, and a source path.

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

Figure 11: Getting the Person Name and ID Values

The source path is retrieved from the field's binding in the same way as the email address widget in the previous section. However, the source path is not passed directly to the `get` method of the data accessor that is retrieved from the *context*. Doing this would return the entire XML document as a string. Instead, the source path is first extended by 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 that the data accessor retrieves 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 might 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 represented in the body content of an element means that the path formats shown in the section are all that is required to use in a widget. However, the 1.17 Extending Paths for XML Data Access on page 81 section describes XML data access through path extension in full detail.

Generating the HTML Content

With the data for the photograph widget that is 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 12: Marking Up the Photograph Data

The same techniques that are used to construct the email address widget by using the DOM API in the previous section, also apply here. The URI used to link to the details page, a static image and the `FileDownload` servlet are described here.

Linking to a UIM Page

The URI of Social Program Management 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.

Therefore, all UIM pages are considered to be in a locale “folder”. When one UIM page is linked to another, it is always in the same locale (or “folder”). Therefore, the locale is not specified in the URI when a link is generated. For example, in the sample code that is shown here, 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 13: Linking to a UIM Page

Linking to a Static Image

Linking to a static image was described when the email address widget is created in the previous section, 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 path prefix 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 14: Linking to a Static Image

Linking to the `FileDownload` Servlet

The `FileDownload` servlet is used to download an image resource from the Social Program Management 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 15: Linking to the `FileDownload` Servlet

The `FileDownload` servlet must be configured to use the parameters that are shown in the URI here to download the correct photograph. This is described in detail in later in the section.

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 is added to the client application component, or the existing file is modified if it exists, to associate the view-renderer plug-in class with that domain.

To access data in XML form and use the path extension feature that is described earlier a “marshal” plug-in *must* also be configured *exactly* as shown here. Failure to do so means 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 16: Configuring the E-Mail Address Widget

Applying the configuration here, the view-renderer of the custom widget is now started 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 is used instead. As no edit renderer is defined in this configuration, the edit-renderer of the parent or other ancestor domain, is inherited and used. Typically, this is the associated`TextEditRenderer` by default with the SVR_STRING domain. However, this type of widget is displaying a subset of the information the Social Program Management application captures about a person. An editable version of this widget would not be expected. Instead, the information would be edited through the standard Social Program Management screens that are 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 1.15 Configuring Renderers on page 72.

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 the HTML anchor tag that downloads the photograph when clicked, the photograph widget creates an HTML image tag by using the same URI that displays the image within the page. The example here 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 17: 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 uses this value when the URI is generated to the `FileDownload` servlet. The remaining configuration elements and attributes define the server facade to start and its 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 18: Example of the HTML to Show an In-line Image

The HTML for the image element should look like the example here. 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, starts the facade, and retrieves its output fields, which contain the file name and binary file data.

1.7 A Details Widget Demonstrating Widget Reuse

The presentation requirements of many pages can be satisfied with simple UIM pages that contain fields that are laid out using clusters and lists.

However, the presentation of this details widget requires more processing such as displaying the person's name and reference number in a different font, refer to The Sample Widgets on page 27. Also, the email address is presented in the same form as shown in 1.4 An EMail Address Widget on page 23. This widget is reused within the details widget.

The objectives for the section are:

- show how to develop a widget that presents the details of a Person by using formatting not possible on a plain UIM page.
- show how to reuse the email address widget described earlier.

Prerequisites

The previous sections in the guide.

Defining the HTML

In the details widget, there are a number of lines of plain text that display the person's address, date of birth and other details. 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 the application of CSS rules:

```
<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 19: HTML Output of the Details Widget

The HTML here is formatted for clarity, but it is generated without any indentation or line breaks, as these are not necessary for the browser to present the email address properly and increase only 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 CSS rules for all content within the widget are defined. The root `div` element is 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. The HTML representing the email address is identical to that described in 1.4 An EMail Address Widget on page 23.

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

Figure 20: Custom CSS for the Details Widget

The `header-info` and `contact-info` classes allow the specific presentation requirements (for example, changing the font) to be implemented. 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 that is 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 that contains 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@company.com</e-mail>
</details>
```

Figure 21: An XML Document Describing a Person

The XML here 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 here. The class extends the same base class as the email address widget and the photograph widget, as it also is a view renderer. The class is 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 22: The Renderer Class for the Details Widget

Accessing Data in XML Form

Data from the XML document are accessed in the same way as the photograph widget described in the previous section. 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 23: Getting the Person name and Reference Number

All values in the XML document can be accessed by using the same technique except for the email address value. The email address widget that is described in 1.4 An EMail Address Widget on page 23 is reused to output the email address. As shown in that section, the email address widget uses a `Field` object, its `Binding` property, and a source path to access the email address value. The next section will explain how to start that renderer.

Generating the HTML Content

The same technique, described in previous sections, of using the DOM API to generate HTML content can be used to output the HTML show earlier in the section. The only new concept comes at the point when the HTML for the email address is to be output. The email address widget is reused within the details widget to output the HTML required for an email address.

The `render` method of a widget is usually started by directly by the Cúram infrastructure. The parameters that are 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` elements used within a `FIELD` element. To start one widget from another it becomes the developer's responsibility to ensure that the appropriate widget is started and the correct parameters are supplied to it. The code that is 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 24: Starting the Email Address Widget from the Details Widget

The steps to start the email 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 here. See 1.12 Overview of the Renderer Component Model on page 57 for full details.
2. Set the domain of the `Field`.

   The underlying domain definition of a `Field` is used to select the appropriate widget. 1.4 An EMail Address Widget on page 23 showed how the email address widget was associated with the SAMPLE_EMAIL domain definition. This domain definition is set on the `Field` as shown here.
3. Set the source path of the `Field`.

   1.4 An EMail Address Widget on page 23 section showed how the email address widget used its source path to access the value of the email address. This is normally set based on the `CONNECT` in UIM. In this case, the source path for the widget must be specified “manually”. The details widget must tell the email address widget where to get its data from. As shown earlier the email address is embedded in the XML document that is supplied to the details widget. The path extension technique to access XML data, that is described in previous sections, can be used to specify the source path for the email address widget.

   The `setSourcePath` method of the `FieldBuilder` is used to set the source path as shown in the following excerpt from the example here. 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 email address widget.

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

   This demonstrates the benefits of the path system to access data. In 1.4 An EMail Address Widget on page 23, the email address was retrieved directly from a server interface property. In the section the email address is retrieved from an XML document. However, the email address widget is identical in both cases. It retrieves its data by using a source path and is abstracted from what source path resolves to “behind the scenes”.
4. Create a `DocumentFragment` for the widget content

   As shown in previous sections, the DOM API is 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 Social Program Management infrastructure.

In this case, the fragment must be created by using the `createDocumentFragment` as shown here.

5. Start the email address widget

   The email address widget is started 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 adds its HTML content to this fragment. The final parameter is reserved and is always be set to `contract.createSubcontract()`.

6. Append HTML generated from email address widget

   After the email address widget is started, 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 is added as a child of the `div` element with the `contact-info` CSS class.

The first three steps here build up a "component model", in this case a single `Field`. The remaining steps then *render* the model as HTML. The 1.12 Overview of the Renderer Component Model on page 57 section 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 is added to the client application component, or the existing file is modified if it exists, to associate the view-renderer plug-in class with that domain.

To access data in XML form and use the path extension feature that is described earlier a "marshal" plug-in *must* also be configured *exactly* as shown here. Failure to do so means 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 25: Configuring the Person Details Widget

Applying the configuration here, the view-renderer of the custom widget is now started 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 is used instead. As no edit renderer is defined in this configuration, the edit-renderer of the parent or other ancestor domain, is inherited and . Typically, this is the associated`TextEditRenderer` 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 be edited through the standard Social Program Management screens that are 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 1.15 Configuring Renderers on page 72.

1.8 Tying Widgets Together in a Cascade

This section expands on the reuse of widgets to produce the “Person Context Panel Widget”. The “Person Context Panel Widget” widget is a combination of the photograph widget and details widget that is positioned side by side. The previous section introduced widget reuse by showing how the details widget might delegate to the e-mail address widget to generate part of its HTML content.

Using the exact same technique, the “Person Context Panel Widget” might combine the output of the photograph and details widgets and display them side by side to produce the content that is 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 might be phrased as: “To combine the output of multiple widgets in a horizontal layout”.

The previous section 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” is responsible for building the component model. In this case, the component model is represented as a collection of `Field`'s; one for the photograph, the other for the person's details. The “Person Context Panel Widget” passes the component model to a new widget, the “Horizontal Layout Widget”. This widget in turn delegates to photograph and details widgets that are introduced in previous sections and combine their output. The advantage of this abstraction is the “Horizontal Layout Widget” might 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 the section the “Person Context Panel Widget” delegates to the “Horizontal Layout Widget”, which in turn delegates to the widgets introduced in earlier sections. This delegation is known as a “cascade”.

Prerequisites

The previous sections in this guide.

Defining Data in XML Form

The XML document for the “Person Context Panel Widget” widget is a combination of the XML documents that are used by the photograph and details widgets that are described in previous

sections, but combined in a new root element. The root element allows each of those renderers to be reused.

```
<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@company.com</e-mail>
  </details>
</person>
```

Figure 26: 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 that are combined by placing them in the cells of an 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 27: 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 might be customized specifically when it is contained within the table that is shown here 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 here. The class extends the same base class as the previous widgets, as it also is a view renderer. The class is 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 28: The Renderer Class for the “Person Context Panel Widget”

The skeleton renderer class for the generic “Horizontal Layout Widget” is shown here. The widgets described up to now in the guide are “view renderer's” based on the `AbstractViewRenderer` class. The component model that is provided to each widget was a single `Field` (the first parameter of its `render` method). As described in the introduction to this section, “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 is 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 29: 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 Cúram application that is ready for immediate use 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 30: Building the component model and starting the "Horizontal Layout Widget"

The steps to build the model and start 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 here. See 1.12 Overview of the Renderer Component Model on page 57 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" is associated with the `horizontal-layout` style. This must be set by using the `setStyle` method as shown here. The style corresponds to a particular renderer implementation class. Configuration of this "style" is described later in the section 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 section, a `Field` is created using a `FieldBuilder`. Setting the domain definition to SAMPLE_PHOTO_XML ensures that the photograph widget is started. The next step is to set its 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". 1.6 A Photograph Widget on page 28 showed how data for the photo widget was accessed in the XML document by 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 here to account for the root `person` element. When the photograph widget runs, the paths are combined to ensure the full path corresponding to the combined document is used. The `Field` is created by 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 by using the `getComponent` method and added to the `Container`.

5. Create a `DocumentFragment` for the widget content

   As shown in previous sections, the DOM API is used to create HTML elements and add them to a `DocumentFragment`, supplied as the `fragment` parameter to the `render` method. The `DocumentFragment` is supplied by the Cúram infrastructure. In this case, the fragment is created by using the `createDocumentFragment` as shown here.
6. Start the horizontal layout widget

   The e-mail address widget is started 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 adds its HTML content to this fragment. The final parameter is reserved and is always set to `contract.createSubcontract()`.
7. Append HTML generated from horizontal layout widget

   After the e-mail address widget is started, 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 is 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 that is 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 in a previous section.

```
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 31: Generating an 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 subclass of the latter, a cast is required to a `Container`. A for loop is used to iterate over each item in the collection by 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. Start 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 here).
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 an HTML table. However, it is abstracted from the underlying details of the components it is outputting. It is 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 might 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 must be associated with a domain definition, SAMPLE_PERSON_XML is used. To allow access to values that are embedded in XML documents, a “marshal” plug-in *must* also be configured *exactly* as shown here.

```
<?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 32: Configuring the Person Context Panel Widget

Horizontal Layout Widget

As described in a previous section, 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 is added to the client application component, or the existing file is modified if it exists, to associate the component-renderer plug-in class with the `horizontal-layout` style as shown here.

```
<?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 33: Configuring the Horizontal Layout Widget

The `horizontal-layout` style is what the “Person Context Panel Widget” used when delegating to the “Horizontal Layout widget” , for example,

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

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

More information about configuring renderers and other plug-ins is provided in 1.15 Configuring Renderers on page 72.

1.9 A Text Field Widget with No Auto-completion

The section describes edit renderers that are used to mark up read/write values with HTML. It expands on the details in the previous sections by introducing more advanced concepts that are related to the creation of input controls on HTML forms.

The sample widget that is presented in the section 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 application that is ready for immediate use. The plug-in displays an HTML text input control. For the input of an SSN, it can be desirable to prevent the web browser from storing the SSN in its cache of entered form data and later providing SSN values by using its form field auto-completion feature.

Some browsers support an HTML attribute to disable auto-completion of the value of an HTML input control. The sample shows how to render the HTML text input control, integrate it into a form page, and add the new attribute to disable auto-completion in the browsers where it is supported.

Prerequisites

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

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 shown with a question mark.

The values is provided by the renderer, as explained later.

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

Figure 34: 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 35: 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 it is submitted 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 that is associated with the server interface property of the target connection. Each domain definition is associated edit-renderer and view-renderer plug-in classes. As a target connection is present, the system automatically uses the edit-renderer instead of the view-renderer when the field is rendered.

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

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

Figure 36: 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 is 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 that are recorded and embedded in the form by the system is also submitted now. 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 are assigned to which server interface properties identified by the target paths. The label is used in if a validation error occurs, so that the error message can report the label of the field in error.

Accessing the Data

As described in an earlier section, 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 goes when the form is submitted. However, the source path might or might 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 has no explicit initial value.

When no explicit initial value is defined, an initial value might still be displayed. The UIM `FIELD` element supports a `USE_DEFAULT` attribute. If this attribute is set to `false`, then no default initial value is 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 started 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 Social Program Management domains that are ready for immediate use, but they can 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 that are detailing what fields are in error. The values that are displayed in the HTML input controls in this case are the values that are entered by the user before the form is submitted. Regardless of what initial values were originally shown, the user might have changed any or all of these values. Depending on circumstances, then, the initial value of the HTML input control might 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 37: 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, usually, 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 is 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 38: 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. 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 that is required to implement the basic features is shown here.

```
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 39: 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 is 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 that is defined for the corresponding attribute in UIM). The parameter value can be null, but calling the `equals` method on the literal string value is still safe in that case and yields the wanted 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 can 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 by 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 that is passed to the renderer. For example, this UIM SCRIPT element is 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 that is provided by the abstract base class can set all of the appropriate event attributes on an 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 is added to the client application component, or the existing file is modified if it 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 40: Configuring the SSN Edit Renderer

Applying the configuration shown here, the edit-renderer of the custom widget is now started 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 is used instead. As no view-renderer is defined in this configuration, the view-renderer of the parent or other ancestor domain, is inherited and used. Typically, this is the TextViewRenderer that is associated by default with the SVR_STRING domain.

More information about configuring renderers and other plug-ins is provided in 1.15 Configuring Renderers on page 72.

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 can be used only where started 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 the section.

1.10 Internationalization and Localization

The guide provides a basic understanding of the internationalization and localization processes and how they apply to widget development.

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 for each term. The number in each abbreviation is the number of letters that are 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 can 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 that are 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 is 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.

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.

CDEJ Support for Internationalization

The CDEJ is internationalized in many ways. Not only are text elements that are 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 that are provided with the CDEJ. These use Java's Collator API, but can be overridden to support custom sorting rules if required.
- Locales can 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. Similarly, for date values non-numeric months names are translated.

In general, there is no need to specify the locale when the CDEJ rendering API is accessed, as the locale is automatically determined and applied when necessary. Some types of plug-ins, particularly the converter plug-ins that are 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 through the converter plug-ins, or other locale-aware sources, and the localization happens automatically before the value is returned.

Widget Internationalization

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

The following are the main internationalization issues of concern to a widget developer:

- 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
- Lying out content on the HTML page in a way that can accommodate the increased length of text when localized into other languages

1.16 Accessing Data with Paths on page 75 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 that is 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 is harder and more expensive to localize the application and also affects the accessibility of the application. 1.11 Accessibility Concerns on page 54 describes how applications are often required to be accessible to as many people as possible. People with visual impairments can 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 that are named for the locales. For example, create an *`images`* folder within the *`WebContent`* folder. Create folders that are 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 here. 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 41: Referencing Localized Image Files

The `getLocale` method returns the locale of the active user, so the image source URI might 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 might be omitted and the locale might appear in the image file name.

A problem with this scheme is that a user with a locale `en_US` does not see 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 here. 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 that contain 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 that is 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 1.16 Accessing Data with Paths on page 75.)

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 can 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, or to supply localized text elements to the JavaScript routines. Both requirements may apply. The specific requirements vary between widgets and are beyond the scope of the guide. However, the basic approach for the widget developer is to generate JavaScript content that contains 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 that contains 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 53, by calling `getLocale` and converting it to a string. Localized text elements that are 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 can also be affected by localization requirements. The text of a label in one language can 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 might require twice the amount of space or even more.

1.11 Accessibility Concerns

The section introduces 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.

Overview

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 might include:

- 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 that is 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 who want to learn more about accessibility and its application to the web. The widget developer can identify what the accessibility regulations and guidelines are for the jurisdiction in which the application is employed and aim to comply with those. It is beyond the scope of the 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 can 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. When a form is used, the screen reader informs the user of the input control that currently has the input focus.

For example, the user can use the Tab key to move the focus to the text field with the label **Date of Birth** and the screen reader announces “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 can 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 that is used by the CDEJ is the same technique that widget developers 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 are 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 can have only 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, appears beside two input controls, one for each field. A search form can have a **Surname** label that appears beside a text field and a check-box. The user inputs the surname into the text field and checks the checkbox if the search finds 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 wanted. However, it is easily achieved by 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 tooltip that is shown then the use hovers over the control with the mouse pointer. This allows sighted 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 here. Therefore, using the title attribute 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 by 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 it is omitted 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 a page is rendered, the CDEJ renders the field label before the widget's renderer plug-in is started for the field value (assuming labels of shown to the left of the values). As the CDEJ does dictate what input control is produced by an edit-renderer plug-in, it cannot know in advance what the ID of the control is and cannot set an ID in the `for` attribute of a `label` element. Therefore, it is not possible 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 that are described here.

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 that relate to the non-use of the `label` element can be safely ignored after the presence of the `title` attribute is confirmed.

Font Sizes

It is recommended that the use of, relative font sizes when a widget's HTML output is styled. Relative font sizes, which are 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 the page is scaled, or scale only the text along with all other non-text content, which cannot be the user's preference.

1.12 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 that is 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 by 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 can 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 that are 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 is used when selecting the appropriate renderer plug-in. A field can also be rendered with a component-renderer plug-in, but a view-renderer or edit-renderer is 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 is 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 that is 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 Social Program Management application.

- **`ComponentParameters`**
  A component's parameter values, derived from the corresponding UIM attributes, are stored in a `ComponentParameters` object that is retrieved by calling `Component.getParameters`. The interface extends `java.util.Map<String, String>`, but the returned map can not be modified. When building new components at runtime, add more 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 1.16 Accessing Data with Paths on page 75 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 can be set, but their use is not supported in the Social Program Management 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 by 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 by using the `ComponentBuilderFactory` class that is 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 Social Program Management 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 that is being built. When complete, call `getComponent` to get the instance of the newly built component object. When `getComponent` is called and returns the new component, the builder object resets all of the properties and can 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 can 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 1.16 Accessing Data with Paths on page 75. 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 sets all of the properties of the 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 by 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 becomes 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 by using a builder, the `copyShallow` method is called on the `ContainerBuilder` instead of the `copy` method. The `copyShallow` method does not copy any references to the child components. Copy these references 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.

1.13 Design and Implementation Guidelines

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 not a concern of a client application developer who is 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

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.

1.2 Approaches to Customization on page 14 described the approaches to widget development in order of increasing complexity.

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

Do Divide and Conquer

A complex widget that is 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 results 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 that is described in 1.7 A Details Widget Demonstrating Widget Reuse on page 35. This requirement might not be met by using multiple fields in a UIM `CLUSTER` element because the layout would not fit into the strict grid that is provided by a cluster. However, an alternative approach to its development is this sequence:

1. Create a UIM page that contains 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 requires 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 that are 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 later, if it is still necessary. The individual widgets that are developed in the second step can also be reused when other details panel widgets, or widgets for unrelated purposes are developed.

Do Check for Nulls

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

The CDEJ never supplies null arguments to the `render` method, but if one renderer starts another, this cannot be guaranteed. In a view-renderer, the field's source path is never null, but the target path is always null; these do not need to be checked if this is assumed. In an edit-renderer, the field's target path is never null, but the source path might or might not be null and is always checked.

The field's parameters might or might not be null. Typically, the parameters reflect the attributes that are 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 value 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 by using the `DataAccessor`, the values might be null in some cases. If a path to a server interface property does not resolve to null, the `DataAccessor` throws an exception instead. Paths to values within an XML document that are resolved by using a `SimpleXPathMarshal` can result in a null value. See 1.17 Extending Paths for XML Data Access on page 81 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 that are identified earlier in the 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 used 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.

However, note , that the extension of the CDEJ renderer plug-ins for custom widget development, is not supported in the Social Program Management 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 whether 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 that is 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 maximizes the possibilities for its reuse.

When using XML document, it can 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 has the form shown here.

```
<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 42: An XML Document Describing Contact Details

The initially invoked renderer plug-in for the new widget, the contact renderer, uses the copy-and-modify technique on the `Field` object that is described in 1.12 Overview of the Renderer Component Model on page 57 and demonstrated in 1.8 Tying Widgets Together in a Cascade on page 40 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 further extends this path with `/street` and `/city` to resolve and present the address values.

This arrangement works, but the reusability of the address widget is compromised by the order in which the paths is extended. This is a consequence of the structure of the XML document. Were the address renderer to be used in a stand-alone address widget, its XML data might look like this:

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

Figure 43: 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 started 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 that is 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: 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 44: 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 stand-alone 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 stand-alone 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 did not result in any added complication.

The situation for the e-mail address value is slightly different. In the stand-alone 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 by using `/contact/e-mail` before the rendering of the value is delegated. Both renderers can now operate without any additional complication, as the e-mail address renderer blindly resolves its source path to the e-mail address value and be unaffected by the fact that the path can 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 is 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 arre to be started directly in association with a UIM `FIELD` element and define the format of the data that they consume and the paths that they can extend to access that data.
- Move on to the design of the delegating renderer that delegates to the above simple renderers. Determine how it creates 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 follows from the design of the renderers in the cascade, not the other way around.

Taking this bottom-up approach to the design ensures that each of the ultimate elements in the rendering cascade are clearly defined and readily reusable. Taking a top-down approach can seem to work well at first, but it is almost inevitable that some problem occurs 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.

Do Not Introduce Concurrency Issues

The application can service requests from many users at the same time. Even when a single user is active, the application can still receive concurrent requests for several pages that are presented to that user in the tabbed user interface.

At runtime, only one instance of each renderer plug-in class is created for each domain or style. The application can 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 lifecycle as renderer plug-ins.

Maintaining state information within a plug-in instance causes concurrency problems. A developer can 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 by using method arguments, the developer passes information through fields of the plug-

in class, concurrency defects arise. Do Not Introduce Concurrency Issues on page 63 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 45: A Plug-in Class with a Concurrency Defect

The `DefectiveEMailAddressViewRenderer` class is similar to the `EMailAddressViewRenderer` class developed in 1.4 An EMail Address Widget on page 23. 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 runtime, there may be concurrent requests for pages that contain e-mail addresses, so the `render` method of a single instance of the renderer plug-in for e-mail addresses can be started from more than one thread. This can lead to a defect where the shared field value becomes corrupted.

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 starts `createAnchor`, e-mail address E2 is rendered and shown to user U1. This can not be a serious problem for e-mail addresses, but the same defect might 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 might 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 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 problem is simple: do not use fields of a class to pass information between methods; use the methods' arguments instead. Do Not Introduce Concurrency Issues on page 63 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 46: 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 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.

Do Not 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 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. 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 the renderer plug-in is developed.

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 automatically starts 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 1.6 A Photograph Widget on page 28 and described in detail in 1.17 Extending Paths for XML Data Access on page 81.

Do Not Do Too Much

The client-tier of the application produces a HTML response for each page request. This CDEJ sends this HTML response to the web browser before the full HTML content of the page is complete. The CDEJ starts 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 the next renderer is started.

This way, little of the response is held in memory at any one time and resource usage is minimized. This is important for pages that can contain much 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` is held in memory until the `render` method returns. Only now is the fragment serialized and its allocated memory freed. The memory use of widgets that produce a large volume of HTML content can or cannot pose a problem. If such a widget is used on many pages and by many concurrent users, assess the potential impact of its high memory use. For widgets that are used rarely or by only a limited number of users, memory use can 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 can 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 started 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 can go unnoticed, but the same inefficiencies can 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 can be acceptable. However, if hundreds of values are presented in a long list, the resource use increases 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 can be presented in lists. For each field in a list column, the CDEJ creates 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 can decide to present values that are used in clusters in one way and values that are 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 are used. Using two different domain definitions for the same data can require considerable changes to the application UML model.

Supporting Field-level Security

The Social Program Management 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 is not displayed if the authenticated user is not authorized to access all of the server interfaces started 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 that are 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 can be able to view the details of a person, but can not be authorized to view the salary of a person. The salary field can 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 can 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 is never started. 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. In the example that is 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 started method on the `DataAccessor` throws a `DataAccessSecurityException` instead of returning a value. If the renderer plug-in does not detect 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 detect the exception and produce that value instead. The example here demonstrates this; the context is the `render` method and the `DataAccessSecurityException` class can be imported from the `curam.util.common.path` package.

```
String value;

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

Figure 47: Implementing Field-level Security

After the try... catch block, the *value* variable holds either the real value of the server interface property that is indicated by the field's source path, or `****`, depending on whether 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 detect the security exception, then the page is not rendered. If the developer detects 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 can attempt to modify a person entity, but the user is not authorized to access the salary field. The user can see the salary text field on the person modification form that is initialized with the `****` value. If the user submits the form, this literal value overwrites the real salary value on the database. More likely, the user sees 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. Therefore, in an edit-renderer plug-in, the developer should not detect the `DataAccessSecurityException` and allow the rendering of the page to fail. No secure information is 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 can be presented on a different 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.

Related information

Adding New CSS Rules for Custom Widgets

When custom widgets are developed, the developer is in complete control of the HTML that is generated for their custom widget and what CSS classes it references. The developer might 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 Social Program Management 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 (for example, color) is applied due to the cascading and inheritance rules of CSS. Choosing this option means the widget is subject to changes from any future release of the Social Program Management application.
- Specific - If the widget has specific styling requirements then ensure that they are explicitly defined in custom CSS for the widget. This helps to insulate the widget from changes to the default styling within the application. The recommended approach is to use the features that are 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 and so forth.) for a custom widget can be analyzed and the developer can decide on whether it can 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 Social Program Management application, the underlying HTML and CSS do not change drastically. However, a new release of the Social Program Management application can include a new user interface and with it major changes to HTML structure and CSS. Even if a custom widget has specific CSS, it might be necessary to update it to adhere to the Social Program Management application's new look and feel.

1.14 Testing, Troubleshooting and Debugging

Writing a widget's renderer plug-in class (or classes) is only half the battle. For many widgets, particularly those that depend a lot on JavaScript and custom CSS styling, the battle has only started. 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 is 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 are supported, as each browser has its own way of interpreting and implementing the CSS standards.

Similarly, javaScript 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 can 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 the same problem. Sometimes there is an easy solution to the problem that would take 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 must 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 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, when 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 usually shows a series of nested exception messages before the first trace is displayed. 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 can be. Do not ignore these errors or dismiss them or fail to follow the resolution steps in the documentation. These 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 does not seem to be started at all, check that it is correctly configured, that the configuration has the highest priority in the component order and that the application is built after these changes are 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. Now, 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 can 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 can 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 is happening 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 resolves 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 that are run in the context of the client application server and debugger breakpoints that are placed into the renderer plug-in class can be used to inspect the operation of the plug-in at runtime.

When a breakpoint is not reached as expected, the problem can 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 the guide.

Trace messages that 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 by using, for example, `System.out.println`. When Tomcat from within the Eclipse IDE is run, the messages appear in the console view of Tomcat process. Once the trace messages are 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 that it takes to test experimental changes. However, beware that Firefox can not render the content in the same manner as other browsers, such as Microsoft™ Edge. If Microsoft™ Edge 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 Microsoft™ Edge.

1 See the Mozilla web site for details.

2 See the Firebug web site for details.

1.15 Configuring Renderers

The customization of the configuration that associates edit-renderer and view-renderer plug-ins with named domain definitions, is supported in the Cúram application.

Overview

Component renderers are associated with styles, not domains, so these are configured separately. Styles support only a single plug-in, a component-renderer, so their configuration, which is similar to the domain configuration, is simpler. Styles are not defined in the UML model like domain definitions; they are 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 section.

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 section, but the *Cúram Web Client Reference Manual* is consulted for more details.

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 section provides additional information about the style configurations.

warning *Purpose* -based Configuration

The developer can see domain and style configurations in the default CDEJ configurations that configure domains or styles by using a `purpose` attribute instead of a `class` attribute. Configuration that uses purposes is more complex then configuration that uses named classes and custom configuration that uses purposes is not supported in the Social Program Management application; only class-based configuration can 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 currently supported in the Social Program Management application. 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 that is described in the samples of the guide and in more specific detail in 1.17 Extending Paths for XML Data Access on page 81.

Any references to select-renderer or marshal plug-ins in the Javadoc for CDEJ, or information that is 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 can result in the invocation of a legacy renderer.

Configuring Domain Renderers

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.

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. An example is shown here.

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 48: An Example of a `DomainsConfig.xml` File

It is possible to override all of the plug-ins that are associated with a domain (subject to some support limitations described in the previous section). However, it is important that the developer specify only 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 are resolved by using the CDEJ default configuration or inherited from an ancestor domain of the configured domain. This behavior is preferred .

Defining unnecessary custom configurations for plug-ins can have unwanted effects that can 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 that is 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 is not 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 is not reflected in the application. Therefore, it is very important 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 here.

```
<?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 49: 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 is used. As styles do not form a hierarchy like domains, there is no inheritance behavior in the configuration.

Using the name of a style that is defined in the CDEJ default style configuration overrides 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 can

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 this approach is used, it is best to choose a separator that is different from any separator that is used for words in the style name to avoid accidental name clashes.

1.16 Accessing Data with Paths

Paths are references to sources of data. They are similar in concept to file system paths that are used to access files or XPath expressions that are used to access data in a structured document. All access to data of any kind from a renderer is performed through 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.

Overview Diagram

The terminology that is used to describe the parts of a path is shown in the figure below.

Figure 50: The Anatomy of a Path

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

The path shown here 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 can have a selector or a selector followed by one or more predicates. The predicate is used to qualify the data that is 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 is 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 1.17 Extending Paths for XML Data Access on page 81, 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 that is passed to a render method contains a `Binding` object that specifies a source path or a target path, or both. 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 need to resolve these paths to their values and do so without inspecting the paths or depending on them being in any particular form. It is this unquestioning processing of any path that allows renderer plug-ins to be reused easily in many different contexts and in rendering cascades.

Renderer plug-ins resolve paths that the `DataAccessor` object available from the `RendererContext` object that is 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 that is *returned by* the `format` method of the converter plug-in for that domains.
- **`getRaw(Path)`**
  Gets the raw value of the data. For domain-specific data, this is the value that is *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 is 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 can only support a subset of these methods. The `get` method is always supported, but the `getList`, `getRawList` and `count` methods cannot 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

Usually, a renderer plug-in just resolves the values of the paths that are given to it in the `Binding` of its `Field` object. However, in some cases, the renderer requires data other than that referenced by the paths.

For example, a renderer can 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 that encodes 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 that are 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 that are listed here, 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 the application is built, it is stored in a JAR file on the classpath as the file */`sample/resources/X.properties`*. Then the resource name becomes `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 following example of accessing general properties shows how a renderer plug-in can 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);
```

Only the `get` method is supported when general properties resources are accessed. If no such property can be found, the `get` method throws a `DataAccessException`.

`Path` objects are immutable; they are similar to `java.lang.String` objects in that respect, or to the component objects described in 1.12 Overview of the Renderer Component Model on page 57. 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 following example, where the value of the *dtlsPath* variable is never changed by calls to `extendPath` after it is 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"));
```

Where properties files are supplied for several locales, the properties file name differs, but the path that is used to reference the property does 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 code here does not change; the resource identifier remains `sample.PersonDetails`. The `DataAccessor` automatically determines 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 can be uploaded and stored in the database of the application for later retrieval. This service is called the Application Resource Store. When 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 that are uploaded to this store.

The path form is a little different from the paths that are 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 Merative™ Social Program Management 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 can need to be modified at runtime.

The path is created by extending the prefix path that is 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 by using the name that is 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 can help to make the type of the resource more readily identifiable from its name when the resource store is administered. 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 that is named `PersonDetails.properties` is shown.

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

As with general properties resources, only the `get` method is supported when general properties resources are accessed. If no such property can be found, the `get` method throws 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 by 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 that is encountered in the path as shown. This is shown here.

```
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"));
```

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 changes only 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 are specified separately through the administration interface. If the files *`PersonDetails_en_US.properties`* and *`PersonDetails_es.properties`* are uploaded, the administrator can 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`, as appropriate. 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 that is appended to its name.

Literal Values

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

An example is shown here.

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

Figure 51: Encoding Literal Values

The literal value can contain characters that might be confused with the path syntax, so the value must be escaped when the path is constructed. 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 by using `DataAccessor.get`, the escaping is reversed automatically, so there is no requirement on the consumer of the path to treat it differently to any other.

1.17 Extending Paths for XML Data Access

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

The section 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 that is 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 that is 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 that is associated with the domain definition of that server interface property is started to evaluate the extra path steps regarding the value of the server interface property. The examples in the guide show how this can be used to extract data from XML documents that are returned in server interface properties. Two domain marshal plug-in classes are provided with the Social Program Management application that are ready to use for this purpose.

The `SimpleXPathMarshal` class supports the resolution of XPath-like expressions against data that is 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 that are supported by these marshal plug-ins are not true XPath expressions, though they aim to be as similar as possible to a small and simple subset of the location paths that are defined by the W3C XPath 1.0 recommendation.

The paths operate on a DOM document that is 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 identifies 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 can 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 52: A Sample XML Document

For example, if the XML document has the form shown in Simple XPath Expressions on page 81, 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 here. 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 side of the equals sign only. The string can be quoted with single quotation marks or double quotation marks. If single quotation marks are used, then the string can contain double quotation marks 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 here.

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 sample document here):

- ***element-name***
  An element name identifying the elements to be selected within the context that is 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 that identifies one of several elements with that name in the context that is 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 that identifies elements with that name and with that value for the named attribute in the context that is provided by the previous path step. See the example here for more details.
- ***@ attribute-name***
  An attribute name that identifies an attribute of the element or elements that are 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 here).

For convenience, the following step form can 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 is later applied. A common scenario if all that is required is a count of the nodes.

A valid path can select zero or more nodes. The values that are 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 can only appear as the last step in a path. The supported functions are as follows:

- **name()**
  Gets the qualified name of the first node that is selected by the path. This is the element or attribute name that includes any namespace prefix.
- **local-name()**
  Gets the name of the first node that is selected by the path. This is the element or attribute name not including any namespace prefix.

Evaluating the Paths

Paths are evaluated by 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 that is 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 here 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, if 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 that is described here. 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 the first node (in document order) selected by the simple XPath expression that is given by the path, or, if a function call, the string value of the result of that call. If no nodes are selected, the result is 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, if a function call, the resulting value of that call. If no nodes are selected, the result is null.
- **`getList`**
  Gets the list that contains 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 contains 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 contains the three string objects, one each for the body text of each element. However, evaluating the path `/values/value[3]/*/name()` returns a list that contains 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 that contains 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 that are selected by the 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 simple string values from XML documents are extracted.

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 that are supported by the `SimpleXPathMarshal` are supported by this ADC class.

This `SimpleXPathADCMarshal` plug-in performs 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, except for 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 data in XML documents is transported. The format avoids problems that can 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 produce the wanted result. The classes that are 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 by 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 that denotes 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 is 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 that are formatted according to 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, regarding the document in Simple XPath Expressions on page 81, 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 is 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 are in the UTC time zone. They are converted to the user's time zone when formatted.

1.18 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));

      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);
    }
  }
```

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

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

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

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 actual people or business enterprises is entirely coincidental.

The licensed program described in this document and all licensed material available for it are provided by Merative under terms of the Merative Client Agreement.

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 Merative, 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. Merative, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. Merative shall not be liable for any damages arising out of your use of the sample programs.

Privacy policy

The Merative privacy policy is available at https://www.merative.com/privacy.

Trademarks

Merative ™ and the Merative ™ logo are trademarks of Merative US L.P. in the United States and other countries.

IBM®, the IBM® logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Adobe™, the Adobe™ logo, PostScript™, and the PostScript™ logo are either registered trademarks or trademarks of Adobe™ Systems Incorporated in the United States, and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft™, Windows™, and the Windows™ logo are trademarks of Microsoft™ Corporation in the United States, other countries, or both.

UNIX™ is a registered trademark of The Open Group in the United States and other countries.

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