The Zimbra AJAX Toolkit (AjaxTK) – A Toolkit for Developing Rich, Browser-based Applications

Zimbra, Inc.
www.zimbra.com

DRAFT

Copyright © 2005 Zimbra, Inc.

All rights reserved

Table of Contents

1 Executive Summary 3
2 Introduction 4
3 Source Packages 5
4 Event Model 6
5 DHTML Widget Toolkit (DWT) 6
5.1 Component Hierarchy and Built-In Widgets 7
5.1.1 DwtControl 7
5.1.2 DwtComposite 7
5.1.3 Built-In Widgets 7
5.2 Event Model 12
5.3 Drag-and-drop 12
5.4 XForms 14
6 Network Programming 15
6.1 net Package 15
6.1.1 LsRpcRequest 16
6.1.2 LsRpc 16
6.1.3 LsPost 16
6.2 soap Package 16
6.2.1 LsSoapDoc 16
6.2.2 LsSoapFault 17
6.3 xml Package 18
6.4 util Package 18
6.4.1 LsCookie 19
6.4.2 LsDateUtil 19
6.4.3 LsStringUtil 19
6.4.4 LsTimedAction 20
6.4.5 LsVector 20

© 2005 Zimbra, Inc.
All Rights Reserved

1 Executive Summary

Since the launch of the Web, application developers have faced a trade-off between “thick” client applications (such as those based on PowerBuilder, VisualBasic, and .NET) and “thin” client applications that run in a web browser and have traditionally consisted of HTML pages constructed by a web application server via technologies such as JSP, ASP, or PHP. While web-based applications have freed developers from the need to provision, update, and secure software on the client, they have not, at least as of yet, provided the richness of thick clients. That gap in user experience between web-based and native clients has been closing with the introduction of technologies like Dynamic HTML (DHTML), which combines JavaScript and Cascading Style Sheets within web pages. With the recent emergence of Asynchronous JavaScript and XML (AJAX), the gap is closer to disappearing altogether.

While the name AJAX has only recently been coined, the underlying technologies (JavaScript a.k.a. ECMAScript, XML, SOAP, and so on) have been maturing for years. AJAX applications combine the rich UI experience of programmed clients (after all, AJAX means the UI logic is a JavaScript program) with the low-cost lifecycle management of web-based applications (no client installations or updates). Under AJAX, the relatively thinner layer of UI rendering logic is programmed in JavaScript and downloaded to the client, but the business logic remains on the server, where it is typically coded in Java, C#, PHP, C/C++, and so on—indeed any programming language with XML and web service bindings. The communication model between AJAX client and server accommodates the same rich interaction scenarios (including server push to the client) afforded by traditional client/server.

Moreover, AJAX is more seamlessly integrated with web standards (HTML, URI, etc.) than alternative browser plug-in technologies such as Flash or Java applets. In fact, AJAX is already a web standard, so it runs uniformly (admittedly with a bit of testing) across browsers, operating systems, and hardware. (We anticipate that some existing Flash/Flex users will choose to build their next-generation web applications in AJAX, while incorporating existing, especially less interactive, Flash content within their AJAX UIs.) AJAX also leverages the same security model of traditional web applications: privacy via SSL/TLS and authentication via userid/password or via public key infrastructure). As with other web applications, authorization is done on the server-side.

So in a nutshell, AJAX uniquely combines:

- Richly interactive user interface;
- Richly interactive client/server communications;
- The zero-cost administration of a web client;
- The look and feel of the web;
- Seamless integration with other web content (e.g., “mash ups”); and
- The investment protection of a web standard.

The only gate we anticipate to explosive growth in AJAX deployment is the current lack of easy programming. While a truly compelling UI can be built with AJAX, it is far from easy to do so today. The state of the art is perhaps not unlike the very early days of client/server Unix applications before the emergence of standard toolkits and WYSIWYG (What You See Is What You Get) authoring environments like PowerBuilder and VisualBasic.

The Zimbra AJAX Toolkit (AjaxTK) represents a substantial step forward. Zimbra, Inc. spearheaded the development of AjaxTK because it was crucial to the user interface of the Zimbra Collaboration Server. AjaxTK can reduce the programming work required for typical browser applications by 80% or more, increase application portability across browsers, as well as

© 2005 Zimbra, Inc.

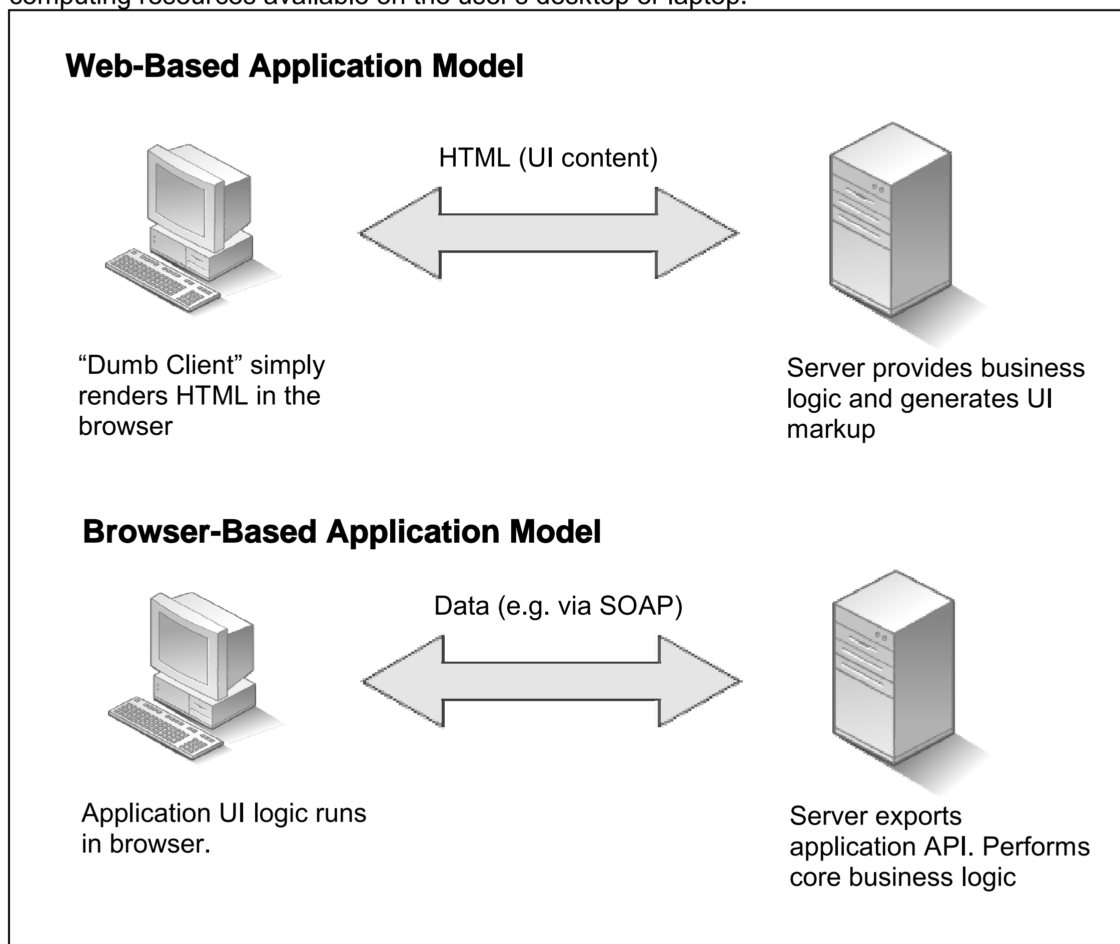
All Rights Reserved

make those applications far easier to evolve going forward. So while we at Zimbra are not in the next-generation UI business, our AjaxTK is nevertheless a potential major leap forward for launching your own AJAX applications. The next anticipated milestone for AjaxTK is support within existing GUI authoring tools (The AjaxTK widget set was actually modeled after Eclipse SWT). Stay tuned!

2 Introduction

Most IT organizations prefer to provide web-based applications to users, because such applications are easier to deploy and manage than traditional desktop applications - as anyone who has had to update tens of thousands of desktops with the latest security patch can attest. Paradoxically, many users are reluctant to use web-based applications, as they tend to be more cumbersome and much less feature-rich than their desktop counterparts.

This is quite unfortunate as modern browsers such as Firefox, Safari, and Internet Explorer 5.5+ provide a powerful application framework upon which rich client applications may be developed. Such applications are able to match, and even exceed, the capabilities of their desktop counterparts. In fact we prefer to call such applications “browser-based” rather than "web-based", as they are in truth clients that exchange not UI, but rather data, with the server, much as is done with traditional client/server applications. The benefits of this approach is that all UI and UI logic is rendered and executed within the browser, thus making full use of the powerful graphical and computing resources available on the user's desktop or laptop.



All Rights Reserved

There are several key elements that are provided by modern browsers that enable them to provide a rich platform for full-featured application development:

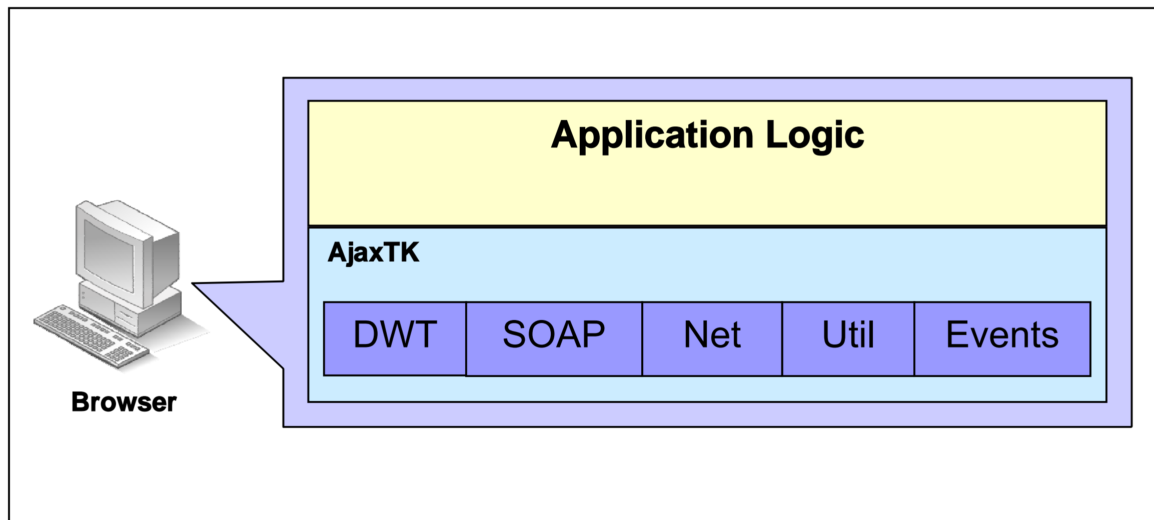
- Support for the DOM (Document Object Model)
- CSS (Cascading Style Sheets)
- Scripting via the JavaScript programming language
- The ability to communicate with a server via HTTP requests and responses

The Zimbra Ajax Development Toolkit (AjaxTK) is a 100% JavaScript toolkit designed for developing browser-based applications. AjaxTK leverages the browser capabilities described above to provide application developers with the elements required to develop rich client applications.

Some of the key components of the AjaxTK are:

- User interface development
- Network communications (both synchronous and asynchronous)
- SOAP programming
- XML document creation and manipulation
- UI event handling and management.

AjaxTK runs completely within the browser, providing a platform for developing applications that deliver the rich client/server experience that is typical of traditional thick desktop clients and to which end-users are accustomed. Applications implement their logic on top of the facilities provided by AjaxTK as illustrated below.



The remainder of this document provides an overview of AjaxTK and its component parts.

3 Source Packages

This section provides an outline of the source packages that compose AjaxTK.

- **config** – configuration information and message localization files
- **core** - base exception classes and environment information
- **debug** – debug classes for runtime application debugging
- **dwt** – DHTML Widget Toolkit
    - **config** – CSS rules, images, and localized text

All Rights Reserved

    - **core** – exception handling and low-level DOM utility functions
    - **dnd** – drag-and-drop support
    - **events** – various events used by the toolkit, built on the general AjaxTK event support
    - **graphics** – points, rectangles, and CSS utilities
    - **widgets** –the DWT widget set and supporting classes
    - **xforms** – provides an XForms implementation for creating complex forms

- **events** – base event and event listener classes, as well as the event manager class that is responsible for event registration and dispatching
- **net** – network communications
- **soap** – SOAP document handling
- **util** – utility classes for such tasks as string manipulation, cookie management, data manipulation, deferred action support, and callback support
- **xml –** XML document handling

4 Event Model

Developers familiar with the Java event model will find the AjaxTK model to be quite familiar. The event model is a typical publish/subscribe model. Event subscribers indicate interest in various events by registering “listeners” with event publishers. Listeners are essentially nothing more than a method and an optional object context under which to execute the method. When events fire, the publishers notify the appropriate listeners by providing them with an event object that encapsulates the event data. This provides for a loose coupling between publishers and subscribers. The AjaxTK event model may be used as a basis for creating Model-View-Controller (MVC) style applications, by providing the event notification foundation for use between the MVC components

The following classes implement the AjaxTK event model:

- ***LsEvent*** – The base event class. All events derive from this class.
- ***LsEventMgr*** – Provides the framework for adding, removing, and notifying listeners. This class is used by the DWT toolkit components (see section 5). *LsEventMgr* is commonly used by applications when implementing a Model-View-Controller paradigm. For example, it is utilized by models to notify subscribers of data change events.
- ***LsListener*** – A listener is registered with event publishers to subscribe to events. It typically consists of a method and an object which provides the context for executing the method. The event manager (see above) will call the listener's handleEvent() method in order to call the method that was provided during construction.

5 DHTML Widget Toolkit (DWT)

DWT is a JavaScript-based UI Toolkit that is modeled after traditional UI toolkit paradigms such as Java AWT/Swing and SWT. It provides for and supports a number of the same components that UI developers are familiar with when developing traditional UI applications, such as buttons and menus. DWT hides many of the details of the DOM, cross browser impedance mismatches, and browser memory leaks. It provides a consistent modern interface for developing browser-based applications. While DWT provides a rich set of pre-built widgets, it is designed to be extensible so that developers may freely extend it with their own set of custom widgets. It should be noted that a key design goal of DWT is to not get in the way of the developer, such that they may use as much or as little of the toolkit as part of their development. This is borne out by the fact that it is quite easy to directly weave DWT components into raw HTML. In addition, there is an important static class (called Dwt) that is part of the core package which exports a large

number of methods which operate directly on HTML DOM objects and that provides important functions for eliminating potential browser memory leaks that an unwary developer could unwittingly trigger.

5.1 Component Hierarchy and Built-In Widgets

This section will describe the DWT component hierarchy and the set of pre-built widgets that are supplied with the toolkit.

5.1.1 DwtControl

DwtControl is the root of the component hierarchy. All components are directly or indirectly derived from it. *DwtControl* provides the following key capabilities:

- Component lifecycle management within the UI framework
- Dispatch of mouse and keyboard events to derived classes. By leveraging the DWT event framework, *DwtControl* provides a canonical publish/subscribe event model that fits in with the AjaxTK event model, thus eliminating the impedance mismatch between the different browser event models.
- A framework for drag-and-drop operations. Specifically, *DwtControl* minimizes the work that subclasses have to perform in order to support drag-and-drop operations.
- Supports APIs for setting bounds, cursor, visibility, z-index layering, tool tips, and more. These APIs are built on top of the methods provided by the Dwt class that is part of the core DWT package.

5.1.2 DwtComposite

If a component is not a container (for example a button widget), then it directly extends *DwtControl*; however, if it is a container (for example a menu contains menu items), then it will derive from *DwtComposite*. This class provides a framework and methods for managing child components (a *DwtTree* widget may have an arbitrary number of *DwtTreeItem*s). Examples of methods provided by *DwtComposite* are: `getChildren`, `getNumChildren`, and `removeChildren.`

5.1.3 Built-In Widgets

The rest of the components or widgets within DWT are derived from one of the two classes introduced above. The remainder of this section will briefly describe the widgets provided “out of the box” with DWT. It should be noted that Zimbra is continually adding to this set of Widgets in order to provide an ever richer palette from which to create applications

DwtButton

DwtButton derives from *DwtLabel* to which it adds button behavior. When the mouse is moved over a button it transitions to the “activated” state. Pressing and releasing the action button (generally the left mouse button) “triggers” the button which will fire off a DwtSelectionEvent (see 5.2) to any registered listener. Since *DwtButton* derives from *DwtLabel* it may display a combination of an image and/or text in various orientations. Besides providing a simple button implementation, *DwtButton* can be configured as a:

- Menu drop button: If a menu is added to the button, then the button will display a dropdown arrow icon which when pressed will display the menu. See *DwtMenu* for the various styles of menu that may be added to *DwtButton*.
- Toggle button: This kind of button can be toggled on and off. The state is changed by

© 2005 Zimbra, Inc.

All Rights Reserved

triggering the button from one state to another.

DwtCalendar

This class implements a calendar widget. Among other things, this widget allows you to specify the first day of the week, the number of days in the working week, selection mode (e.g. day, week, work week), whether to highlight "today" with a bounding box, etc. *DwtCalendar* may be instantiated as a standalone widget, or it may instantiated as the child of *DwtMenu* .

DwtColorPicker

DwtColorPicker creates a color picker displaying "Web safe" colors. Instances of this class may be used with *DwtMenu* to create a color picker menu.

DwtDialog

Provides a standard popup dialog what has a title and a set of buttons (standard buttons - OK, Cancel, and Details – and/or custom buttons). The dialog is not popped up in a separate browser window but rather emulates a popup in the context of the window from which it is invoked.

DwtHtmlEditor

The *DwtHtmlEditor* widget implements a WYSIWYG HTML editor that permits users to create HTML content. The editor can be switched between plain text and HTML modes; in addition, it can detect if the browser in which it is being instantiated provides the necessary support for HTML editing. If it does not, then it reverts to plain text mode. *DwtHtmlEditor* provides APIs for setting font style, font family, font size, text style, text justification, paragraph style, as well as APIs for inserting HTML content such as tables, images, horizontal rules, and links. In addition, DwtHtmlEditor publishes a *DwtHtmlEditorStateEvent* which allows for a component such as a toolbar to react to the content under the cursor.

DwtLabel

A label can consist of static text content and image content. The text and image data can be placed relative to each other (e.g. text on the left, image on the right). *DwtLabel* is the base class for *DwtButton*.

DwtListView

An abstract base class for creating list views. *DwtListView* provides the functionality for resizing, moving, showing and hiding columns. In addition it implements the list selection model which allows the user to select a single row, or multiple rows by using the Shift or Ctrl keys. Subclasses implement a method that provides the actual content for each row in the list view.

DwtMenu

This class supports several menu styles:

- `BAR_STYLE` – Implements a traditional bar style menu.
- `POPUP_STYLE` – Popup menus are typically used for context menus (when the user clicks the right mouse button over an item).
- `DROPDOWN_STYLE` – drop down menus are typically sub-menus, or menus that are associated with a DwtButton.
- `COLOR_PICKER_STYLE` – This style of menu can have only a single DwtColorPicker instance as a child.
- `CALENDAR_PICKER_STYLE` – This style of menu can only have a single DwtCalendar instance as a child.

All Rights Reserved

DwtMenuItem

Menu items are created as children of *DwtMenu* objects. Menu items may have one or more of the following styles:

- `CASCADE_STYLE` – cascade style menu items are the default style and appear in `POPUP_STYLE` and `DROPDOWN_STYLE` *DwtMenu*s. .
- `PUSH_STYLE` – This style of menu item is used only as the top level item in a `BAR_STYLE` *DwtMenu*. In fact if the parent of the *DwtMenuItem* is a `BAR_STYLE` *DwtMenu*, then the style of any immediate children of the menu is forced to `PUSH_STYLE`.
- `SEPARATOR_STYLE` – Provides a separator between menu items. If a `SEPARATOR_STYLE` menu item is the immediate child of a `BAR_STYLE` menu, it will be a vertical separator; for other styles of menus it will be a horizontal separator.
- `RADIO_STYLE` – Radio style menu items belong to a group of other radio style menu items. A menu could have multiple radio groups. Only one member of a particular radio group may be "checked".
- `CHECK_STYLE` – Check style menu items are simple toggle (checkbox) menu items.

DwtMessageDialog

This subclass of *DwtDialog* implements a message dialog for displaying status, errors, etc. Clients interact with this class via its `setMessage` method. This method accepts four parameters:

- A message string to be displayed
- An optional detail string which is displayed if the detail button is selected
- The style, which can be one of: critical, warning, or informational. Each style causes an appropriate icon to be displayed in the dialog.
- A title that is displayed in the dialog's title bar

DwtPropertyPage

DwtPropertyPage class is an abstract class that provides a framework for displaying properties of a data object using HTML elements, form elements and DWT widgets. A developer will extend this class to take advantage of the framework for creating forms and tracking user input. *DwtPropertyPage* provides flexible methods that allow rendering data forms inside HTML tables. Members of *DwtPropertyPage* take care of creating and tracking unique IDs of data fields, assigning event handlers to data fields, and setting the `_isDirty` flag which indicates if a user has edited any of the displayed data. *DwtPropertyPage* derives from *DwtComposite* and therefore may contain and control any other DWT widgets.

DwtSash

This class provides the ability to place a horizontal or vertical sash between any two components. A sash provides the ability to resize two components such that as one component is made smaller, the other grows to occupy the space previously occupied by the first component. One use case for *DwtSash* is separating a message list and message preview panel in a mail application.

DwtSelect

DwtSelect provides the same functionality as the native HTML select tag. The reason that the native select tag is not used is that under certain browsers the select tag does not respect z-index values and hence shows through UI layers.

© 2005 Zimbra, Inc.
All Rights Reserved

DwtShell

When instantiated, *DwtShell* will occupy the whole window of the current browser (in the next version of the toolkit *DwtShell* will be able to be instantiated in the context of an `iframe` or other HTML element). It essentially provides an application's top level graphical context and handles such tasks as browser window resizing, blocking the browser context menu as appropriate, providing overlays for blocking user input, and keyboard event dispatching. *DwtShell* is the only component that does not require another DWT component as its parent, thus one must always be instantiated to provide the context under which other components may be instantiated.

DwtTabView

DwtTabView implements a horizontal tab view. *DwtTabView* manages instances of classes derived from *DwtTabPageView*. Tab views are added to *DwtTabView* via its `addTab` method.

DwtTabViewPage

An abstract class that is sub-classed to provide content for a tab page inside a *DwtTabView*. *DwtTabViewPage* interacts with *DwtTabView* for display and bounds management for classes that derive from it. The *DwtTabViewPage* class is derived from *DwtPropertyPage* class and therefore has access to all its methods for rendering data forms and tracking user input.
A developer may override the `showMe` method in order to implement lazy rendering of the tab view pages.

DwtText

This class provides a simple encapsulation of a text node DOM object.

DwtToolBar

This class implements a basic toolbar. A *DwtToolBar* may contain buttons, input fields, menu buttons, select objects, etc.

DwtToolTip

DwtToolTip implements tooltip behavior. A tooltip provides more information about the item over which the mouse is currently hovering. This information may be textual, image, a combination. There is generally a short delay between the time the mouse enters a component and when the tooltip "pops up". The content presented by *DwtToolTip* may be any HTML content and it may be retrieved and modified on the fly via the `getContent` and `setContent` methods.

DwtTree

DwtTree implements an arbitrary depth tree. Nodes in the tree are instances of *DwtTreeItem*s (see below); any node with children can be expanded and collapsed. A *DwtTree* can support single selection or multi-selection of the items it manages; this behavior is determined by the style parameter passed into its constructor. *DwtTree* also supports "checked item" semantics meaning that items added to it will have a checkbox next to them. This is useful for implementing checked lists etc. *DwtTree* has been designed to support thousand of nodes without major performance impact.

DwtTreeItem

A *DwtTreeItem* is essentially a node in a *DwtTree*. Of course a *DwtTreeItem* may contain other *DwtTreeItem*s as their children thus permitting a multi-level tree implementation. A tree item may contain an icon and content. The content may be another component or even HTML. This makes for interesting tree structures which may contain input fields, buttons, etc.

© 2005 Zimbra, Inc.
All Rights Reserved

DwtWizardDialog

DwtWizardDialog creates a step-by-step wizard dialog. Content for the steps of a wizard are provided by classes that derive from *DwtWizardPage* class. *DwtWizardDialog* takes care of creating "Next", "Previous", "Cancel" and "Finish" navigation buttons; it also takes care of creating a progress bar through the *DwtWizProgressBar* class. The steps of a wizard are added using the `addPage` method that accepts two formal parameters:

- `wizPage` – a reference to an instance of a class that derives from *DwtWizardPage* class
- `stepTitle` – a string title of a step that will be displayed in a progress bar

The `addPage` method returns an integer that is a key in *DwtWizardDialog*'s internal map of wizard pages. This key can be used to access and control wizard pages in the following methods:

- `getPage(pageKey) –` returns a reference to an instance of a class derived from *DwtWizardPage*; this instance holds the contents of a page identified by a `pageKey.`
- `goPage(pageKey) –` forces a wizard dialog to switch to the step identified by `pageKey`

DwtWizardDialog exposes the following methods to control the flow of a wizard:

- `popup` – pops up the wizard
- `finishWizard –` pops down the wizard
- `goNext` – forces the wizard to switch to the next step
- `goPrev` – forces the wizard to switch to the previous page

When a wizard switches to a step it calls the `showMe method of` a corresponding instance of *DwtWizardPage.*

DwtWizardPage

DwtWizardPage is an abstract class that provides the content for steps of a wizard created by *DwtWizardDialog. DwtWizardPage* interacts with *DwtWizardDialog* to manage the display, flow and bounds management for the classes that derive from it. The *DwtWizardPage* class is derived from *DwtTabViewPage* class and therefore has access to all methods of *DwtPropertyPage* for rendering data forms and tracking user input. *DwtWizardPage* exposes the following methods that allow controlling flow of a wizard:

- `switchToNextPage(pageKey)`
- `switchToPrevPage(pageKey)`

A developer may override these two methods if any validation is needed before leaving the current wizard page, and before switching to either the next or previous steps in a wizard. A developer may also override the `showMe` method in order to implement lazy rendering for the wizard pages.

DwtWizProgressBar & DwtStepLabel

These classes may be extended in order to customize the look and feel of a wizard dialog progress bar.

All Rights Reserved

5.2 Event Model

The DWT event model extends the AjaxTK event model. DWT events should not be confused with the events generated by the browser (e.g. mouse, keyboard, and resize events). While these events are trapped by DWT and mapped to a canonical form, they are just a subset of the events that the toolkit generates. With the exception of *DwtMouseEventCapture*, the rest of the classes in the DWT events package are the different event classes exposed by the toolkit. We will discuss *DwtMouseEventCapture* before introducing the event classes.

There are a number of scenarios where it's desired for all mouse events to be delivered to a specific component no matter where they occur. Examples of such scenarios are menu management and drag-and-drop. The *DwtMouseEventCapture* class facilitates this functionality by providing a browser independent mechanism for capturing mouse events and dispatching them to a particular component.

The remainder of this section will briefly introduce the set of events exposed by DWT:

- ***DwtControlEvent*** – Control events represent bounds changes to a component. Bounds changes include size and location changes. This event reports the old and new size and location coordinates of a component that has been moved and/or resized.
- ***DwtDateRangeEvent*** – This event is generated by the *DwtCalendar* widget when the range of dates it is displaying is changed.
- ***DwtDisposeEvent*** – DWT components are destroyed via their `dispose` method. When that method is called, it results in (potentially multiple if the component being disposed has child components) *DwtDisposeEvent*s being triggered.
- ***DwtEvent*** – Base class for all DWT events. Mostly defines a set of constants.
- ***DwtHtmlEditorStateEvent*** – This event is generated by *DwtHtmlEditor* when a state change occurs. An example of a state change is clicking on bolded text when the cursor was previously on plain text. The utility of this event is that formatting toolbars can track state at the cursor location, e.g. toggling the bold text button on/off as appropriate.
- ***DwtKeyEvent*** – Provides a canonical representation of keyboard events generated by the browser.
- ***DwtListViewActionEvent*** – Generated by *DwtListView* to indicate that a list view element has been actioned (double-clicked). This class derives from *DwtMouseEvent*.
- ***DwtMouseEvent*** – Provides a canonical representation of mouse events generated by the browser.
- ***DwtSelectionEvent*** – A selection event generally occurs when an item (or in certain cases multiple items) are "selected". The semantics of selection can be different things in different contexts. For example, in the case of *DwtButton*, selection represents the pressing and releasing of the mouse action button (generally the left button). In the case of a list item it represents the selection, or multiple selection, of an item(s) in the list.
- ***DwtTreeEvent*** – Represents events generated by *DwtTree*, for example when a node is expanded or collapsed.
- ***DwtUiEvent*** – The base class for browser-generated events such as key and mouse events. Normalizes such events into a browser independent canonical form. Also provides the mechanism for translating to and from browser-specific formats. One example of normalization is preventing event propagation, which is handled differently by different browsers.

5.3 Drag-and-drop

All Rights Reserved

There are two parts to implementing drag-and-drop: The visual mechanism provided by the UI components (for example a drag icon) as discussed in section 5.1; and the application model. The latter is the focus of this section.

While UI elements may know how to visualize drag-and-drop operations, the semantics of these operations fall to the application logic. For example, from the perspective of the UI, it may be perfectly reasonable for an item to be dragged onto another component, but whether it is semantically possible for that item to be dragged is specific to the application. Consider a tree item that is being dragged as representing an employee and assume that it is being dragged over another tree item representing a department. This operation could be seen as being perfectly reasonable; however, if the destination item is instead the employee's peer, then this may reasonably not be permitted. The point to keep in mind is that permissibility of operations depends on the underlying data model and its semantic behavior.

DWT provides a mechanism for application logic to interact with the drag-and-drop system to allow for consistent and correct behavior.

During a drag-and-drop operation, there is a drag source and zero or more possible drop targets. Continuing with our employee example, it may be possible to drag an employee to a new department or simply to a new manager within the current department. The employee being dragged is called the "drag source".

DWT provides the *DwtDragSource* as part of the drag-and-drop package. The constructor of this class accepts a single parameter that lists the operations that the source supports. Possible operations are:

- `DWT.DND_DROP_COPY` – the item being dragged may be copied
- `DWT.DND_DROP_MOVE` – the item being dragged may be moved

In our example, an employee cannot be copied (at least not until cloning is possible), so the only permitted operation is moving an employee; therefore when the application instantiates the *DwtDragSource* object, it will do so with `DWT.DND_DROP_MOVE` as the only supported operation. Next, the application will have to register a listener with the *DwtDragSource* object that it created. This listener will be called during various stages of the drag-and-drop operation.

When the listener is called, it will be passed a *DwtDragEvent* object by the drag-and-drop subsystem. This object has the following attributes:

- `operation` – the operation being performed (copy or move)
- `srcControl` – the control that is being dragged
- `action` - the drag action that is currently being performed. It can be one of the following:
  - `DwtDragEvent.DRAG_START` – the drag operation is beginning
  - `DwtDragEvent.SET_DATA` – the drag-and-drop system is requesting the data that backs the UI object, for example an employee object
  - `DwtDragEvent.DRAG_END` – the drag operation is ending
- `doIt` – the application sets this value to `false` if it wishes at any stage to abort the operation. Generally the drag listener does not interfere with the operation, but rather lets the drop target deal with data validation; however the application programmer is free to do as he or she wishes.
- `srcData` – The application sets this value during the `DwtDragEvent.SET_DATA` action. In the employee example this may be an object representing the employee, or perhaps just the employee's ID number

After registering a listener with the *DwtDragSource* object, the application must associate this object with one or more controls. For example, we may represent employees by a *DwtTree*. In

All Rights Reserved

this case we would associate the *DwtDragSource* object with each *DwtTreeItem* that represents an employee. This is done by calling the `setDragSource` provided by *DwtControl* and inherited by all classes derived from it.

So far we have created a drag source, but there must also be one or more drop targets onto which draggable components may be dropped. In our example we permit a department to be a drop target for employees.

DWT provides the *DwtDropTarget* as part of the drag-and-drop package. The constructor of this class accepts a single parameter that is an array or one or more transfer types that the drop target accepts. Transfer types are the data (object) types that the drop target accepts. For example, a department object may accept the Employee class as a transfer type. If an object is dragged over a drop target that does not accept that object type, the drop target is considered invalid for that data type. Transfer types are only part of the story, since the application often will have to perform additional checks. For example, it makes no sense to drag and drop an employee onto the department to which he or she already belongs.

Once the application has instantiated the *DwtDropTarget* and set the appropriate transfer types, it will need to register a listener with the *DwtDropTarget* object that it created. This listener will be called during various stages of the drag-and-drop operation.

When the listener is called, it will be passed a *DwtDropEvent* object by the drag-and-drop subsystem. This object has the following attributes:

- `operation` – the operation being performed (copy or move)
- `targetControl` – the control over which the drag source is hovering
- `action` - the drag action that is currently being performed. It can be one of the following:
  - `DwtDragEvent.DRAG_ENTER` – the object has been dragged over the target
  - `DwtDragEvent.DRAG_LEAVE` – the object has left the target object
  - `DwtDragEvent.DRAG_OP_CHANGED` – the drag operation has changed (currently not implemented)
  - `DwtDragEvent.DRAG_DROP` – the object is being dropped on this target. At this point the application performs the necessary actions to execute the operation. In our example, the application may make a SOAP call to the server requesting that the employee be moved, and upon success may move the employee under the new department in the tree view.
- `doIt` – the application sets this value to `false` if it wishes at any stage to abort the operation. The time when this most commonly occurs is during the `DwtDragEvent.DRAG_ENTER` action. It is at this time that the application will determine if the drag source is a viable source via validation checks. Setting this value to `false` will cause the UI to react appropriately (e.g. draw a red border around the drag source and not highlight the drop target).
- `srcData` – The application sets this value during the `DwtDragEvent.SET_DATA` action. In the employee example this may be an object representing the employee, or perhaps just the employee's ID number

5.4 XForms

The purpose of the DWT XForms package is to provide an optimized mechanism for producing complex display and data-entry forms on the client, while shielding the developer from writing the HTML required for creating these displays. The implementation is both a subset and extension of the W3C XForms specification (http://www.w3.org/TR/xforms/). That specification is not fully

implemented, and a large number of additional attributes are provided that allow the developer great flexibility over the appearance and behavior of the forms. All of the additional attributes are optional. It is possible to create an XForm using only elements detailed in the W3C specification. It should be noted that XForms are not only useful for creating complex input forms, but are just as useful for providing a standardized way to present complex data.

There are three elements at the heart of every XForm:

- An XForm object
- An XModel object
- A data instance

The XForm specifies the functionality and appearance of the form. It consists of a list of items, each of which has a representation on the display. Many item types are visual (e.g. labels, text fields, checkboxes and buttons), while some are provided for grouping or aggregating sets of form items for display purposes. Capability is provided for switching the set of fields being displayed, much like a tabbed preference dialog or a wizard, where different sets of options are presented sequentially. There is also the ability for handling lists of items, such that the XForm object will automatically create as many rows in the display form as there are rows in the instance data. XForm items are dynamically shown or hidden as instance data changes; for example, controls that are not relevant because of the current state of the data instance are hidden.

The XModel specifies the shape of the data -- how the data is organized hierarchically, the data types and any constraints (validators) on the data. The XModel is essentially a static set of items (referred to as "model items", to distinguish them from form items), and is mostly used to access property values in the current data instance. In addition, it guarantees that data written to the instance is consistent with all appropriate constraints. Note that an XForm may be instantiated without a corresponding XModel, in which case the XForm will automatically create an empty XModel instance.

The data instance may either be a simple JavaScript object or an instance of a JavaScript class. The fundamental purpose of the Xforms mechanism is to display and modify this data. By default, the XModel will access the properties of the instance directly according to XPath expressions embedded into the model item descriptions.

6 Network Programming

For browser-based applications to be useful, they must be able to exchange data with servers that in many cases implement core business logic. AjaxTK provides two levels of functionality to enable network communication. The first is a low-level network programming layer that is encapsulated in the 'net' package. The second is a SOAP layer that enables document style SOAP protocols. This layer is encapsulated in the 'soap' package

6.1 net Package

The 'net' package provides the AjaxTK low-level network programming APIs. It consists of three classes that assist in network communications between the client and the server.

© 2005 Zimbra, Inc.
All Rights Reserved

6.1.1 LsRpcRequest

Modern browsers all provide a mechanism to directly make HTTP requests to a server; however the mechanism by which they surface this functionality is different. For example, Microsoft Internet Explorer provides an ActiveX object for performing HTTP requests while Firefox and Safari allow the direct instantiation of an *XMLHttpRequest* class for this purpose. *LsRpcRequest* hides these details from the developer.

In addition, *LsRpcRequest* provides the ability to execute requests to a server either synchronously or asynchronously. In either case, it exposes a single `invoke` method. This method has the following parameters:

- `requestStr` – the request string that is passed to the server. It essentially represents the information to be passed to the server
- `serverUrl` – the URL to which to post the request
- `requestHeaders` – an optional hash of HTTP request headers that are to be sent to the server
- `callback` – an optional instance of *LsCallback* (part of the AjaxTK 'utils' package). If present, this parameter causes *LsRpcRequest* to execute the request asynchronously. Upon completion of the request, this callback will be executed and will be passed the results returned by the server.

6.1.2 LsRpc

This static class layers the notion of a communication context on top of *LsRpcRequest* such that a pool of contexts are managed for network communications.

6.1.3 LsPost

LsPost is used to upload files from the client browser to the server using the file upload feature of POST. This singleton class makes an HTTP POST to the server and receives the response, passing returned data to a callback. The form should be within an *`iframe`*, which can then be filled by the callback as appropriate.

6.2 soap Package

The SOAP package provides a set of classes that enable SOAP requests and responses. This package consists of three classes:

- ***LsSoapDoc*** – provides the core functionality of the package by allowing creation and manipulation of a SOAP document
- ***LsSoapException*** – a SOAP exception
- ***LsSoapFault*** – represents a SOAP fault

6.2.1 LsSoapDoc

The core class providing this functionality is *LsSoapDoc*, which provides three static factory methods for instantiating *LsSoapDoc* objects:

- `create` – This method accepts three formal parameters from which it will instantiate a new *LsSoapDoc* object:
    - `method` – the name of the SOAP method
    - `namespace` – its namespace
    - `namespaceId` – an optional namespace ID

All Rights Reserved

- `createFromDom` – This method accepts a SOAP document in the form of a DOM object hierarchy as its only formal parameter and instantiates an *LsSoapDoc* object in the context of that object. This method calls the `LsSoapDoc.prototype._check`, an internal method which verifies that the document object that is passed is a valid SOAP document.
- `createFromXml` – This method is passed a string representation of a SOAP document as its only formal parameter. It then internally instantiates a DOM object hierarchy from that string. As with `createFromDom`, it verifies that it has a valid SOAP document by calling `LsSoapDoc.prototype._check`.

LsSoapDoc supports the following methods for manipulating the SOAP document that it is managing:

- `element2FaultObj` – If the SOAP document contains a SOAP *`Fault`* element, then this method will return an instance of *LsSoapFault* (see 6.2.2), else it will return null.
- `getBody` – returns the SOAP document's *`Body`* element
- `getDoc` – returns the DOM document object representing the SOAP document
- `getHeader` – returns the SOAP document's *`Header`* element
- `getMethod` – returns the SOAP document's *`Method`* element.
- `getXml` – returns an XML string representation of the SOAP document
- `set` – This method could more correctly be named `createElement`. It accepts three formal parameters:
    - `name` – the name of the element to create
    - `value` – the value of the element
    - `element` – if this parameter is not null, then the new element is created as the child of this parameter. If it is null, then the element is created as a child of the *`Method`* element
- `setMethodAttribute` – This method permits setting attribute values on the *`Method`* element, and accepts two formal parameters:
    - `name` – the name of the attribute that is being set
    - `value` – the value of the attribute

LsSoapDoc makes extensive use of *LsXmlDoc* (a component of the xml package described in section 6.3) for parsing and working with XML data

6.2.2 LsSoapFault

When a problem occurs, the SOAP specification provides a well-known way to indicate what has happened: the SOAP fault. *LsSoapFault* is a JavaScript class representing a SOAP *`Fault`* element. *LsSoapFault*'s constructor accepts a single SOAP *`Fault`* element as a formal parameter and extracts salient data from it. This data is exposed by the following public attributes:

- `faultCode` – This attribute maps to the Code element. It can have one of the following values:
    - `LsSoapFault.SENDER -` The problem was caused by incorrect or missing data from the sender.
    - `LsSoapFault.RECEIVER -` Something went wrong on the receiver while processing the message, but it wasn't directly attributable to the message contents.
    - `LsSoapFault.VERSION_MISMATCH -` The namespace on the SOAP envelope that was received isn't compatible with the SOAP version on the receiver.
    - `LsSoapFault.MUST_UNDERSTAND -` A header was received that was targeted at the receiving node, marked `mustUnderstand="true`", and not

All Rights Reserved

understood.
    - `LsSoapFault.DATA_ENCODING_UNKNOWN` – Data encoding is unknown.
    - `LsSoapFault.UNKNOWN` – An unknown error has occurred.
- `reason` – This attribute maps to the *`Reason`* element and contains one or more human-readable descriptions of the fault condition. Typically, the reason text might appear in a dialog box that alerts the user of a problem, or it might be written into a log file.
- `errorCode` – this attribute maps to the *`Detail`* element. We elected to expose this attribute by calling it an error code in order to encourage server developers to use the *`Detail`* element to communicate server error codes.

6.3 xml Package

Different browsers provide different mechanisms for instantiating and managing XML documents. For example, Microsoft Internet Explorer provides an ActiveX object that exports an interface to the MSXML XML parser. Firefox and Safari on the other hand implement the W3C `document.implementation.createDocument` method for creating XML documents.

Not only do the browsers provide different mechanisms for creating and working with XML documents, but they also provide different APIs. For example, Microsoft Internet Explorer provides the 'xml' attribute on every DOM element. The value of this attribute is the XML string representation of that element and all of its children. Firefox, for example, does not have that method.

The 'xml' package consists of a single class, *LsXmlDoc*, to handle all of the intricacies of XML document creation and management, including the encapsulation of cross-browser differences and the provision of a canonical interface.

LsXmlDoc provides three static factory methods for instantiating *LsXmlDoc* objects:

- `create` – creates an empty XML document
- `createFromDom` – accepts a DOM document in the form of a DOM object hierarchy as its only formal parameter and instantiates a *LsXmlDoc* object in the context of that object hierarchy
- `createFromXml` – accepts as its single formal parameter a string representation of an XML document. It then creates an empty instance of *LsXmlDoc* by calling `LsXmlDoc.create`, then calls the `loadFromString` method (described below) on the newly created object in order to construct a DOM object hierarchy representing the XML document.

LsSoapDoc supports the following methods for manipulating the XML documents:

- `getDoc` – returns the underlying XML document as a DOM object hierarchy
- `getXml` – returns a string representation of the underlying XML document
- `loadFromString` – given a string representation of an XML document, instantiates a DOM object hierarchy from it
- `loadFromUrl` – given a URL, loads the XML document rooted at that URL and creates a DOM object hierarchy representing it.

6.4 util Package

The AjaxTK 'util' package provides a number of useful utility classes. This section will briefly introduce some of the more salient ones.

All Rights Reserved

6.4.1 LsCookie

LsCookie is a static class that provides three methods for managing cookies:

- `setCookie` – sets a cookie. This method defines the following formal parameters
  - `doc` – the document on which the cookie is to be set (required)
  - `name` – the name of the cookie (required)
  - `value` – the value of the cookie (required)
  - `expires` – cookie expiration date (optional)
  - `path` – cookie path (optional)
  - `domain` – cookie domain (optional)
  - `secure` – whether the cookie is secure (optional)
- `getCookie` – given a document and a cookie name, returns the value for that cookie, or `null` if no such cookie exists
- `deleteCookie` – deletes a cookie

6.4.2 LsDateUtil

LsDateUtil is a static class that provides a number of methods for working with dates:

- `computeDateDelta` – computes the difference between the current time date/time and its formal parameter that is a date/time expressed in milliseconds since the epoch. `computeDateDelta` returns a localized string describing the difference. For example: “10 days ago”, 5 hours 25 minutes ago”, etc.
- `computeDateStr` – given a JavaScript Date object representing the current time and a target time represented as milliseconds since the epoch, `computeDateStr` will calculate a string relative to current date/time. For example, if the time passed in is within 24 hours of the current time, then a localized string describing the time will be returned; if the time passed in is within the same year as the current date/time, then a month/day string will be returned, etc.
- `daysInMonth` – given a year and a month, returns the number of days in that month
- `isLeapYear` – given a year, returns true if it was a leap year and false otherwise
- `roll` – This method is passed three formal parameters:
  - date – a JavaScript date object
  - field – the field in the date object. Valid values are:
    - `LsDateUtil.YEAR`
    - `LsDateUtil.MONTH`
    - `LsDateUtil.WEEK`
    - `LsDateUtil.DAY`
  - Offset – an offset to add to the specified field in the date

  The method will then add `offset` units to the field `field` of `date` and return the modified date object.
- `validate` - given a year, month, and a day, returns true if it is a valid date

6.4.3 LsStringUtil

LsStringUtil provides a number of methods to aid in string manipulation. It includes methods for:

- Trimming strings
- Extracting and formatting the text content from an HTML document
- HTML encoding/decoding
- Splitting strings
- XML-encoding and -decoding strings
- Replacing variables in a string with values from a list

All Rights Reserved

- Finding the differences between two strings

6.4.4 LsTimedAction

LsTimedAction permits execution of events in the future. A timed action consists of a method to execute, optional parameters to be passed to that method, and an optional object within whose context the method is to execute.

LsTimedAction defines two static methods for managing timed events:

- `scheduleAction` – This method requires an *LsTimedAction* instance, and a timeout in milliseconds as formal parameters. It will then schedule the timed action to be executed after a delay consisting of the given timeout. This method returns an action ID which may be used to cancel an action.
- `cancelAction` – given an action ID, cancels the execution of the corresponding timed action

In order to use *LsTimedAction*, the developer first instantiates an instance of the class and populates the `obj`, `method`, and `params` attributes accordingly. To then schedule the event to be executed, the `scheduleAction` method is called, and to cancel an action scheduled by `scheduleAction`, `cancelAction` is called.

It is important to note that timed actions rely on the `window.setTimeout` method, and so control must return to the browser's event loop before the event will truly be scheduled. Put another way, if an event is scheduled and then a long running piece of code is executed before returning to browser's event loop, then the action will not fire until the long running code completes execution. Also, event scheduling is not modal; once an event has been scheduled, subsequent code continues to execute until the browser's JavaScript engine returns to the event loop. For this reason, it is common to place the scheduling of an event (for example, one that retrieves data from the server) at the end of the code path.

6.4.5 LsVector

This class implements a vector on top on a JavaScript array. *LsVector* provides a large number of methods for manipulating and accessing its contents:

- `add` – adds an object at a specified index in the vector. If not index is specified, adds the object to the end of the vector.
- `addList` – concatenates a list, that could be an array or another *LsVector*, to the end of the vector
- `binarySearch` – performs a binary search for a value within the vector. A comparator may be passed into the search function.
- `clone` – clones the vector
- `contains` – performs a linear search for a value within the vector
- `containsLike` – returns true if the vector contains the given object, using the given function to compare objects. The comparison function should return a type for which the equality test (==) is meaningful, such as a string or a base type.
- `fromArray` – static method to create a *LsVector* instance from an array
- `get` – returns the object at a given index
- `getArray` – returns the vector's backing array
- `getLast` – returns the last element in the vector
- `indexOf` – given an object, returns its index position in the vector, or -1 if it is not contained in the vector
- `merge` – merges an array or *LsVector* at a give offset. Overlapping elements are replaced by the values in list that is passed in as the formal parameter to this method.

- `remove` – given an object reference, removes that object from the vector. Returns true if the object is found and removed, false otherwise.
- `removeAll` – removes all the elements in the vector
- `removeAt` – removes the element at the given index
- `removeLast` – removes the last element in the vector
- `size` – returns the number of elements in the vector
- `sort` – sorts the vector based on the sort function that is passed into the method, or the default `sort` method if none is provided
- `toString` – returns a string representation of the vector. This method can accept two optional parameters: a separator that will be used to separate each element in the vector within the string, and a flag indicating whether to compress the string (ignore empty array elements).

All Rights Reserved