Sausalito Developer's Guide

Beta 12, 5/7/01

555 Ellis Street
Mountain View, CA 94043
650 623-2500
www.cobalt.com

April 23, 2001 12:37 pm

Appendix D Base Data Types

Appendix E Cobalt System Configuration Protocol

Acknowledgements

I would like to acknowledge the following people who have been essential to writing this book: Tim Hockin, Jonathan Mayer, Adrian Sun, Mike Waychison, Will DeHaan, and Kevin Chiu. Thanks also to Karina Eichmann and Gordon Garb.

NOTE: Due to scheduling considerations, the Security and Active Monitor chapters of this book were not included. It is also the case that while this is the final version of this book, it is .76 in keeping with the CCE software release.

Chapter 1

Introducing The Sausalito Architecture

Chapter Contents

Introduction

SunCobalt™ completed its first Web server appliance software architecture in 1998 and began delivery of Sun Cobalt“ Qube appliance and Sun Cobalt RaQ“ server products that same year. In two short years, the Web appliance market has taken off. Today, Sun Cobalt“ RaQ servers host hundreds of thousands of web sites in data centers around the world. Sun Cobalt“ Qubes provide basic web services for thousands more small businesses and educational institutions.

Software developers and service providers now view the Web as the medium for delivering services. They increasingly see Web server appliances as the vehicle for cost effectively and easily delivering these services to the edge of their customers’ networks.

As *appliances*, these products are fundamentally more cost effective and easier to use than pre-Internet general-purpose servers. Just as most consumers receive television entertainment through set-top box appliances on the edge of cable networks, millions of businesses, previously excluded from the information technology market place, will receive services through Web-enabled appliances attached to the Internet.

Sun Cobalt“ recognized from the start that the user interface and underlying software architecture for these Web appliances must be designed specifically for this task. Sun Cobalt“ has worked with leading network providers, including several of the world s leading ISPs and network service providers, to *appliantize* their Web services. Their requirements are at the center of Sun Cobalt“’s second-generation software architecture, named *Sausalito*, which is described in this developer guide.

Sausalito is specifically designed for delivery of services through the web model. This model allows services, hosted on the appliance, to deliver content to many users simultaneously through a graphical user interface.

Sausalito is designed to provide a superb developer platform, with the following goals in mind.

- Provide an extensible architecture enabling third-party developers to customize, modularize, and implement services quickly. Sausalito interfaces are documented in this guide, including tools for tuning the user interface and interfacing with the built-in configuration database. The user interface also includes such features as a software update indicator and single-button install and delete capabilities.
- Provide an easy to understand environment for non-technical users. Sausalito masks the complexity of its underlying software and is intended to provide the framework for maintenance-free services.
- Use open standards for quick development time and strong security. Sausalito is designed to run on top of Linux and, in addition to its own interfaces, uses a number of standard services and interfaces like Apache and LDAP.
- Provide flexibility to localize User interfaces into multiple languages quickly. Sausalito includes a language library for all localized data.

Audience

The audience for this document includes developers who create hardware or software applications that run on Sausalito, Value Added Resellers, and others who want to customize Sausalito-based systems to fit their requirements.

About this Book

This book contains the following sections:

Chapter 2, About The Sausalito Architecture, provides a high-level tutorial of the components that make up Sausalito.

Chapter 3, User Interface, explains how the User Interface works with code samples and examples of how to change the style.

Chapter 5, Introducing The Cobalt Configuration Engine describes the interaction between the underlying software for Sausalito.

Chapter 6, Making Sausalito-Aware Applications, describes the file structure you must use to create an application that runs on the Qube 3.

Appendix A, User Interface Foundation Classes lists the methods in the User Interface Foundation Classes (UIFC).

Appendix B, Utility Classes lists the methods for the Utility Classes.

Appendix C, , Appendix C, About Style, lists the default styles used in Sausalito.

Appendix D, Base Data Types lists the base data types used in Sausalito. You should be aware of these data types so that your software does not overwrite them.

Appendix E, Cobalt System Configuration Protocol describes the CSCP protocol.

Appendix F, CCE Class Definitions lists the properties of CCE classes.

NOTE: This draft includes a subset of the set of final chapters and appendices that will be available when complete. It is being made available in its beta form to help third-party developers create applications.

Related Documents

For information on using Sausalito, please see the *Qube 3 User s Guide*, which is available at `http://www.cobalt.com/support/resources/manuals.html`. Information about Sausalito will also available at `http://www.cobalt.com/products/index.html`.

Document Roadmap

This roadmap tells you where to find information for specific tasks.

Table 1—1 {{need to update this}}

Task	Where to find information
Adding a new menu item	Adding a New Navigation Node on page 3—6
Changing the logo	Making Other Style Changes on page 3—11
Changing the background color	Changing the User Interface Style on page 3—10
Internationalizing your application	Using i18n and l10n in Sausalito on page 4—1
Adding a new service	Building a New Service Module on page 6—3
Working with the Object Database (ODB)	Appendix D
Working with the UIFC classes	Appendix A
Working with the Utility classes	Appendix B
CSCP Libraries	Appendix E
What are the base classes for Sausalito	Appendix C
What are the CCE class definitions	Appendix F

Conventions Used in this Guide

Typographical Conventions

Bold is used for emphasis, for example:

Each UIFC page should have **one and only one** `page` object.

Bold is also used for words found in the user interface, for example:

`test.xml` is shown adjacent to **Style**.

Italic font is used for variables, for example:

`require ::=` *string*

Italic font is also used for new terms when they are first used, for example,

these *widgets* are manipulated from a PHP script by the developer.

`Courier` is used for program names and code, and web resources, for example: `CCE Auth` command returns NULL for failure or a session key for success.

```
char *cce_auth_cmnd
```

```
http://www.cobalt.com/support/resources/manuals.html
```

Programmatic Conventions

The class definitions use the following conventions:

- All class names have the first character capitalized. For example, `System`. If they have more than one word, the first character of all words is capitalized. For example, `MailList`.
- `Namespace` names follow the same rule as class names.
- All property names start with an all lowercase first word. If a property name has more than one word, the first characters of the second word onwards are capitalized. For example, `gateway` and `stylePreference` are valid property names.

Terminology

Sausalito has its own unique terminology:

`Cobalt Configuration Engine` **(**`CCE`**):** A general name for the entire configuration architecture.

`Cobalt System Configuration Protocol` **(**`CSCP`**):** The protocol which connects the CCE client to the `session manager` and the Cobalt Object database. CSCP connections provide object- database functionality and executes handlers as necessary.

`Event`**:** A change in a property of a object within the database.

`Client`**:** Aprogram using CSCP to request or change information.

`Handler`**:** A program called by CCE to affect an event.

`Cobalt Configuration Engine daemon` **(**`cced`**):** The server process which handles incoming connections and signals.

Chapter 2

About The Sausalito Architecture

Chapter Contents

This chapter provides a tutorial-style overview of the Sausalito architecture. It describes the basic concepts, the issues that were addressed in creating this appliance architecture, and the solutions that were implemented to address them.

The Appliance Concept

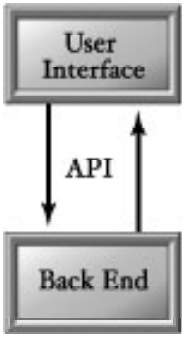
When designing software for a general purpose server, the designers must put as few restrictions on flexibility as possible. However, an appliance does not have this restriction. A Sun Cobalt™ appliance is designed with a single goal in mind: providing a full range of services through a single user interface, while keeping the ease of use of household appliances. This goal enables us to narrow the scope of the software, and consequentially tightly integrate the software into the system.

The Sausalito software architecture is an answer to the appliance concept. Sausalito allows Sun Cobalt™ to provide a single back-end mechanism for monitoring and manipulating the system software. Through this mechanism, a very simple user interface can operate, while keeping the details of the backend system logically separate.

This separation of interface and implementation is a cornerstone of reusable and reliable software design. This allows developers to have a stable exported interface that can be used in their applications for complete integration into the Sun Cobalt™ environment. This is one of the major goals of Sausalito.

Figure 2—1 provides a basic view of Sausalito architecture. The interface provides the glue between the user interface and back end.

Figure 2—1 Overview of Sausalito



The User Interface Defines the Appliance

Appliances make complex systems very easy to use. The user interface plays a strong role in defining the appliance. Sausalito provides the foundation to build web-based user interface on the user interface layer. This layer communicates with users and routes information to and from the back-end. There are several components in the this layer: the navigation manager, User Interface Foundation Classes (UIFC) and `stylist`.

Sausalito was designed to meet internationalization requirements. Sausalito supports users at different locales by working with European and Asian languages. The user interface layer uses a internationalization library to handle this requirement. For example, when the user interface needs to display Welcome to users who speaks German, it asks the internationalization library to get the translated string Willkommen to display to users.

Navigating Around

The navigation manager component is designed to provide basic navigation capabilities to user interfaces. The idea is to separate data that defines the site map and the navigation managers that walk through the map. On a site map, each node denotes a page on the user interface and each page can have multiple widgets. Information about the nodes are stored in special files.

Given a site map, it is up to navigation managers to determine how to walk through them. Different navigation managers can walk through the same site map differently. Some navigation manager can provide a step-by-step walk through while other can show the whole map as a tree structure so that users can pick the right node instantly.

Building Pages

Sausalito provides many utility libraries and UIFC as a widget set on which you can build user interface pages. One of the goals of UIFC is to provide consistency among different pages on a user interface. This is extremely important for the interface s ease of use. For example, UIFC, fields that represent boolean selections always look the same. Otherwise, boolean selections can be represented as a checkbox, two radio buttons or a change-state-button.

UIFC is object-oriented. Each widgets have their corresponding classes. Also, UIFC is currently implemented in PHP. Developers must have a basic understanding of object-oriented programming and PHP before examining UIFC. PHP is a very easy-to-learn and versatile scripting language designed to build web pages.

`HtmlComponentFactory` is a UIFC class that construct widgets and talk to the internationalization library. The basic task of this class is to instantiate UIFC widget classes in common ways and give them parameters of the desired locale. This is the first class to understand within UIFC.

`ServerScriptHelper` is a utility class that simplifies page building. Its main job is to communicate with CCE for authentication and for getting user preferences. It also provides methods to make page building easy.

Ideas Behind UIFC

UIFC is a layer above user interface implementation mechanisms, such as HTML. When we think in HTML, we think at the level of checkbox, radio button, select field and text field. When we think in UIFC, we think at the level of boolean selection, option selection, set selection and typed inputs. HTML is for implementation, UIFC is for design concept. UIFC frees UI designers from very low level implementation details.

UIFC is designed to provide consistency for user interface. Take a simple example: one UI designer can use a checkbox to represent a on/off selection on one page, while another designer can use two radio buttons, one for on and one for off, to represent the same concept. Such kind of discrepancy makes an user interface hard to learn and use. UIFC provides a single widget for boolean selection, so such concept are always represented consistently.

UIFC improves portability. Especially for web-based user interfaces, portability across multiple browser platform has always been a trouble spot. Because UIFC users express things as concepts, UIFC can choose the appropriate implementation through platform-specific tuning or lowest common denominator approaches.

Sometimes, UIFC can be restrictive. For example, what if UIFC expresses boolean as a checkbox while a UI designer has to use radio buttons for the same concept? To provide this bit of flexibility, pages using UIFC can be mixed with HTML and JavaScript; however, UIFC users should be cautious when mixing UIFC and other tools.

User Interface with Style

There are lots of style properties on an user interface. A web user interface includes fonts, font size, color, images, alignment, and other properties. UIFC support style properties, which are stored in special files. UIFC widgets parses through these files to get the right style to display. Sausalito allows multiple styles coexist on the system and allows users to choose ones they prefer. Styles are pretty much like skins in some applications.

Built-in Internationalization

Internationalization is built into Sausalito and supported through an internationalization library. Sausalito users often refer strings by their references rather than the actual strings. This way, the actual string can be fetched from the string catalog based on the locale preference of the user who read the string. Sausalito users can also set locale-specific

properties. For example, when an input field should only be displayed for Japanese but not for other languages, we can introduce a `inputField` property and set it to true only for Japanese. Of course, the code that manages this field must be made aware of this property.

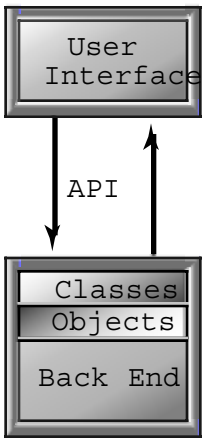
Sausalito is designed such that objects representation resides only on the user interface layer. Developers should not be surprised that anything below the user interface layer only passes references instead of the actual strings or other locale-sensitive resources. When the user interface gets the reference and decides to use it, the internationalization library is then called to resolve it.

Abstraction of the System into Objects

The first step towards separating the interface from the implementation is to separate the data from the process. System data, such as configuration options and users, can become abstract groupings of data or objects. These objects are self-contained, dictating only the information necessary to recreate themselves. An application can define a class or data structure to enable the system to know about and manipulate its data.

This provides developers a flexible way to define new configuration items to the system, as well as a convenient and single mechanism by which to read all system configuration data. Figure 2—2 shows the addition of classes and objects.

Figure 2—2 Adding Classes and Objects to Sausalito



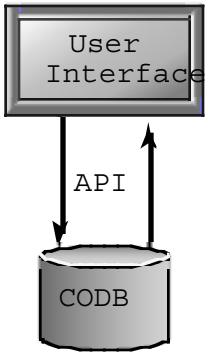
Storing the Objects

Once we have well-defined objects that can accurately represent the system, we need to define how and where to store them and how to retrieve and modify them. Unlike reading configuration files, such as /etc/passwd or httpd.conf, to determine the state of the system, a good abstraction should provide a single, flexible way to access all system configuration data.

The Cobalt Object Database (CODB) is provided as a place to store objects. It is not a database in the sense of commercial relational databases designed to run a corporate enterprise, but instead store the known state of the system. CODB acts as a buffer between a user interface and the system itself.

Objects can be stored, retrieved, modified, and destroyed, all without the user interface having to know about the details of any given application configuration mechanism. Figure 2—3 adds the Cobalt Object Database (CODB).

Figure 2—3 Adding CODB

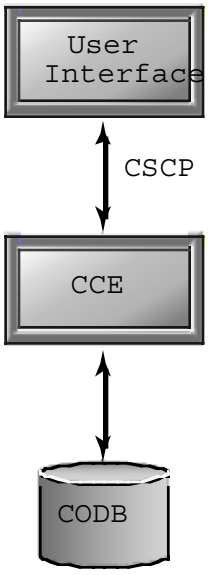


Manipulating the Objects

Now that we have objects that can be stored, created and destroyed, we need to define a mechanism by which to do these things. In order to provide a manageable and accountable access method, Sun Cobalt™ has defined the Cobalt System Configuration Protocol (CSCP), which connects clients to the Cobalt Configuration Engine (CCE). CCE is the process that implements CODB.

CSCP provides primitives to read, write, create, destroy, and search for objects. To make accessing CSCP easier, Sun Cobalt™ provides libraries in several common programming languages, such as C, Perl and PHP.

Figure 2—4 Connecting the UI to CCE and CODB



Extending the Objects

Now that application packages can export their configuration data via CODB classes, other software packages can take advantage of this. Many times, an application package adds some functionality to an existing object that didn't exist in the base object. Consider an application that provides some per-user configuration options. With CODB classes, it is easy to define a class for this data. Now the UI can create an object of this class (an instance) whenever a user is created, and destroying the instance whenever a user is destroyed.

There is one more problem, however. A good abstraction of the object knows nothing of the user interface, and a good user interface engine knows nothing of what classes are available. How, then, do we associate this new per-user class with a user object?

CODB provides the ability to extend a class with a namespace. A namespace is a set of properties, like a class, that piggy-backs onto other classes. We can change our per-user class into a user namespace. Now, whenever a user gets created or destroyed, the namespace goes with it. We also solve the issue of association. We know our new namespace is associated with user objects by it's namespace association.

Watching for Changes

At this point, we have the Cobalt Configuration Engine (CCE) running a database (CODB) which stores instances of classes and namespaces. This configuration engine understands the CSCP protocol to affect changes on the CODB. How do the changes made to the CODB become changes made to the system?

Application packages can register via configuration files to be notified when certain events occur. The registration mechanism provides the ability for any software package to register event handlers (or just handlers) on any class or namespace known to the system. Events understood by the CODB are create events, destroy events, and modify events.

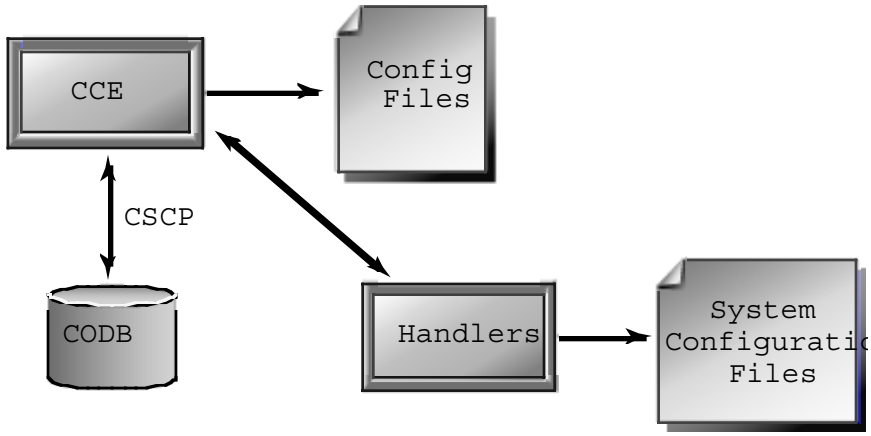
Now that we can register handlers, our software package can create a handler for any events about which it is concerned. For example, if we need to add a user to our application s access list, we might register on the user-create event. When a user is created, our handler is invoked, and we can do our specific task.

Actuating the Changes

When an event is triggered, CCE steps through the list of handlers for that event, and runs each of them, in turn, until one fails, or there are no more handlers to run.

It is the responsibility of each handler to make the appropriate changes to the system configuration to actuate the event.

Figure 2—5 Making changes to the system



Modularity — Doing Your Own Thing

At every stage of Sausalito, concern has been given to retain modularity. It is the goal of the architecture to make adding and removing software packages have no impact on the rest of the system. This principle should be adhered to as much as possible.

What CCE is Not

CCE is not a generic data-store. It is not a place for applications to store their data. It is meant to be a buffer between making abstract changes in the configuration of the system, and those changes happening on the system.

CCE is not a place to store user-interface definitions. CCE should know as little as possible about any particular user interface implementation.

CCE is not a mechanism for a user interface to trigger system events. The user interface should not rely on any knowledge of the handlers that are registered on an event.

CCE is not a replacement for other security mechanisms. While Sausalito tries to be secure and cautious in all cases, it should not forego other security methods.

CCE is not a replacement for a good backups. Much of the system's data is stored in files on the system, not in CODB. Protect your data and back up regularly.

Chapter 3

User Interface

Chapter Contents

Sausalito is Sun Cobalt™ s first fully open programming interface. It is designed to enable third-party developers to create applications that are seamlessly integrated into the look and feel of Sausalito-based appliances. One of the layers within Sausalito is the user interface, which enables you to:

- Manipulate navigation with navigation managers and site maps
- Manipulate look-and-feel style
- Build user interfaces pages that are consistent with Sausalito-based systems

How the Navigation System Works

The navigation system is a sub-system within the user interface that manages navigation through site maps. The system consist of site maps and navigation managers.

The navigation system on Sausalito is a dynamic system. It is generated from a set of files that define navigation nodes. By linking these nodes together, a site map is formed. Users can construct site maps by adding and removing nodes. These site maps are then interpreted by navigation managers. Whenever a user logs in to the user interface, navigation managers use the site map to govern how the user navigates.

Each node on the navigation graph is defined in a XML file. You must have a basic understanding of how XML works to use navigation; for XML information, see `http://www.sun.com/xml/`

Navigation node XML files are located under `/usr/sausalito/ui/menu` directory. These files are very simple. It does not matter where or in what subdirectories these files are placed. All graph related information are contained within the files. Directories can be used to group these files into a more maintainable manner. To add a node, create a new XML file under the directory. To remove a node, remove the file. Each XML file contains all the information the navigation system needs to know about a node.

XML Elements

There are three elements navigation node XML files can use. They are `item`, `parent`, and `access` elements. Each of the files must contain **one and only one** `item` element. Each `item` element contains zero to many `parent` elements. The `parent` elements can be viewed as links from the children to the parent node. A collection of nodes and links together composes a site map. Each `parent` element can have zero or more `access` elements.

With no `access` element, the parent link has no access control and anybody can traverse the link. With one `access` element, access is granted if and only if this requirement is met. With more than one `access` element, access is granted if any one of the multiple requirements are met; this is a `OR` condition.

Table 3—2, Table 3—3, and Table 3—4 lists possible attributes of these elements.

Table 3—2 Item Element Attributes

Name	Type	Description
id	[a-zA-Z0-9_\-]+	`id` must be unique among XML files. Therefore, it is advisable to prepend package or vendor tag to the id. See Using Unique Names on page 3—6
label	internationalizable string	`label` is a short readable string that labels the node. Navigation managers can display a list of labels for users to navigate to. The interpolate function of `I18n` module is used to internationalize this string.
description	internationalizable string	Labels can sometimes be too short. A `description` is used complement the label in describing the node s content. The interpolate function of `I18n` module is used to internationalize this string.

Table 3—2 Item Element Attributes

Name	Type	Description
type	string	`type` is used by navigation managers to distinguish items. They can then act on the items differently. Optional.
url	URL as described in RFC 1738, internationalizable	This `url` points to the content page of this node. The interpolate function of `I18n` module is used to internationalize this string. Optional.

Table 3—3 Parent Elements Attributes

Name	Type	Description
id	[a-zA-Z0-9_\-]+	This is the `id` of the parent node which is described in the `item` element.
order	integer	When there are several children nodes under a parent node, the navigation managers might need to know which child to use first. The smaller the integer, the more important the node is. Optional.
require	string	This is the access required to traverse the parent link. Optional.

Table 3—4 Access Elements Attribute

Name	Type	Description
require	string	This is the access required to traverse the parent link.

Navigation Manager

There are three navigation managers supported by the system currently. They are collapsible list navigation, flow navigation and single navigation. New navigation managers maybe added in the future.

Here is an example site map here to illustrate how navigation managers work:

Node A has no parent

Node B’s parent is A

Node C’s parent is A

Node D’s parent is C and E

Node E has no parent

Node F has no parent

Node G's parent is F

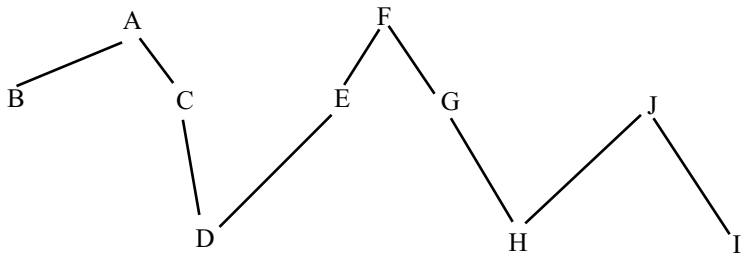
Node H's parent is G

Node I's parent is G and J

Node J has no parent

The site map looks like Figure 3—1

Figure 3—1 Site Map



The collapsible list navigation manager presents a site map in a collapsible list format and lets users navigate by clicking on items on the list. Users can expand or collapse parents to show and hide the children nodes, respectively.

To use the collapsible list, users needs to supply the root node of the site. For example, using the above site map as an example with A as root, the collapsible list will look like:

Item B

Item C

Item D

Note that the node A and E are not being shown. This is because node A is the root and there is no path to descend down from the root to node E. The nodes F, G, H, I and J are not shown either because they are on a separate branch.

The URL for collapsible list navigation manager is at `/nav/cList.php`. It needs a root parameter that specifies the id of the root. Therefore, use it like `http://<ip>/nav/cList.php?root=<root>`.

The flow navigation manager allows users to navigate forward or backward through a site map. Conditional branches for forward are supported.

The root node needs to be supplied to the flow navigation manager and that becomes the first step of the flow.

Take the above site map as an example with F as the root. Navigation starts at F. Users can move forward to G. No moving backward is allowed on the root node. At G, users can move backward to F or forward to either H or I. Moving to H or I depends on a condition check at G. At H or I, users can move back to G or finish the navigation. Note that users at I cannot move backward to J.

Conditional forward is supported by a JavaScript interface. At the node where a conditional forward is necessary, that is, G in the above example, a `getNextItemId()` JavaScript function must be specified in the content page. This function should return the `id` of the next node when it is called with no parameters.

Error checking is supported. When an user wants to move forward, all the `submitHandler()`s of all the form elements of the content page are called with no parameters. The forward operation proceeds only if all the `submitHandler()`s return true. If you use UIFC to build your content page, `submitHandler()`s are automatically defined.

During the forward operation, after error checking is done at the front end, the form on the content page is submitted. The handler of the form submit should tell flow navigation manager if submission is successful or not. If successful, navigation moves to the next node. Otherwise, it stays at the same node. To notify the navigation manager, the form handler should return a page which sets the JavaScript variable `flow_success` to true or false, with true indicating success. Note that this variable is automatically handled by the `toHandlerHtml()` method of the `ServerScriptHelper` class.

The URL for flow navigation manager is at `/nav/flow.php`. The root parameter needs to be supplied, so use something like `http://<ip>/nav/flow.php?root=<root>`.

Single navigation manager only supports one single node and does not allow users to navigate into other nodes.

The URL for single navigation manager is at `/nav/single.php`. The root parameter needs to be supplied, so use something like `http://<ip>/nav/single.php?root=<root>`.

Adding a New Navigation Node

The example below demonstrates how to add nodes to the User Interface (UI). We will add two nodes in the example. Figure 3—2 on page 3—9 shows the result of this addition.

helloMenu.xml

```
<item
    id="sample_helloworldmenu"
    label="Hello World App"
   description="This menu contains the Hello World application">
  <parent id="base_administration" order="100"/>
</item>
```

hello.xml

```
<item
    id="sample_helloworld"
    label="Hello"
    description="This item says hello to the world"
    url="/sample/hello/helloWorld.php">
    <parent id="sample_helloworldmenu" order="0"/>
</item>
```

Node `sample_helloworldmenu` is a child to node `base_administration` and node `sample_helloworld` is a child of node `sample_helloworldmenu`.

Using Unique Names

You must use unique names for navigation nodes to avoid name conflicts. Sun Cobalt™ recommends that you choose a vendor-specific name for your modules, and create directories with the vendor name. For example, if your company name was ivory, your XML files for the account information page would be `ivory_account.xml` in the `/usr/sausalito/ui/menu/ivory` directory.

Building Pages

Pages on Sausalito are built with PHP, because UIFC, the widget classes, and other utility classes are currently implemented in this language. These classes the foundation of Sausalito user interfaces and they are available to be used by developers.

NOTE: More information on building pages will be included in future drafts.

The object classes, called the UIFC define objects such as buttons, lists, checkboxes, and radio buttons. These widgets are manipulated from a PHP script by the developer, and then are automatically turned into proper HTML for display to the user. All the UIFC widgets have been built with the concept of styles. This allows the look of the entire UI to change, with no code changes. UIFC has also been designed to work seamlessly with internationalization, commonly referred to as `i18n`. See Appendix A for a complete reference for UIFC.

NOTE: See Chapter 4 for information on internationalization.

The utility functions provide pre-packaged functionality that is commonly needed by web-based UIFC applications. Utilities, such as conversions between strings and hashes, and widget allocations are greatly simplified by utility functions. See Appendix B for a description of the utility classes. For information on Styles, see The User Interface Style on page 3—9.

A Further Example

We've already shown some examples of adding a menu, so let's put all the pieces together and see how it looks. This example will be expanded in subsequent chapters to show how to internationalize your application, add handlers, and have it work with Sausalito s Active Monitor.

menu/helloMenu.xml

```
<item
    id="sample_helloworldmenu"
    label="Hello World App"
    description="This menu contains the Hello World application">
  <parent id="base_administration" order="100"/>
</item>
```

menu/hello.xml

```
<item
    id="sample_helloworld"
    label="Hello"
    description="This item says hello to the world"
```

```
    url="/sample/hello/helloWorld.php">
    <parent id="sample_helloworldmenu" order="0"/>
</item>
```

web/helloWorld.php

```
<html>
<body bgcolor="#ffffff">
   <h1> Hello, World! </h1>
</body>
</html>
```

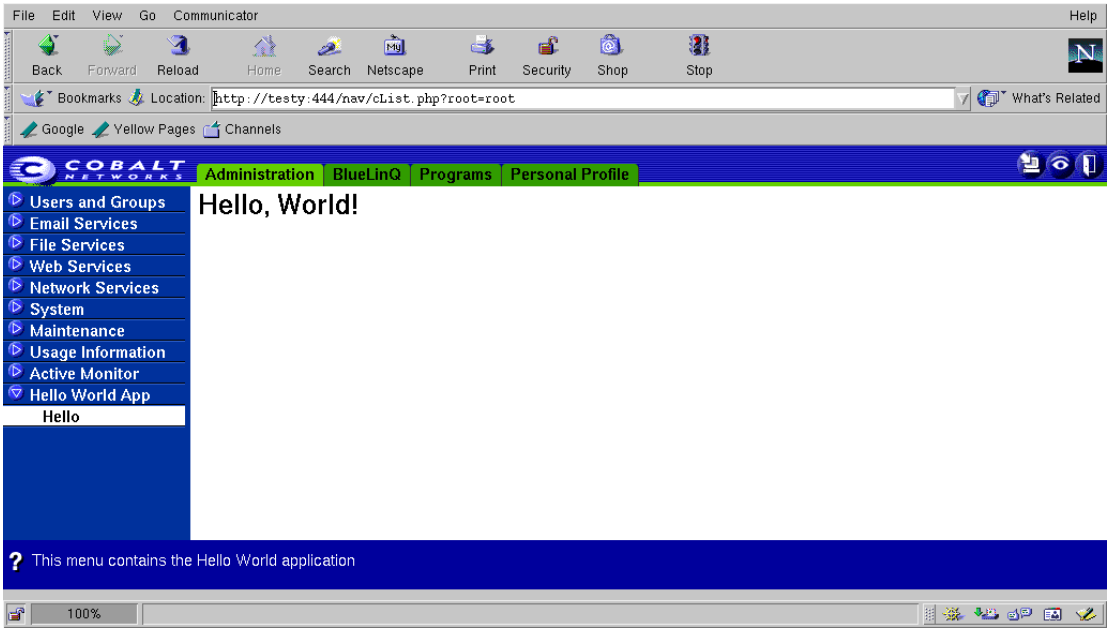
Putting all of these files together creates the web page shown in Figure 3—2 on page 3—9.Now, we can take advantage of the UI libraries. It might seem odd that the next example is, in fact, longer than the non-UIFC version, but for a use as trivial as this, the overhead of UIFC outweighs the benefits. When pages get longer and more complex, however, the benefits dwarf any overhead.

web/helloWorld.php

```
<?php
// PHP file to display "Hello, World"

include("ServerScriptHelper.php");

$servhelp = new ServerScriptHelper();
$factory = $servhelp->getHtmlComponentFactory("base-am");
$page = $factory->getPage();

print($page->toHeaderHtml());

$label = $factory->getLabel("Hello, World!", false);
print($label->toHtml());

print($page->toFooterHtml());
?>
```

Putting all these files into place results in the screen shown in Figure 3—2.

Figure 3—2 Hello World in the Sun Cobalt™ Menu



The User Interface Style

How Styles Work

The UI styles are defined in `Style` definition files. The `Style` definition file contains all the configurable items of the look and feel of UIFC s visual appearance. You can modify this file to change logos, background colors, text, and other features. See Appendix C, About Style .

Changing the User Interface Style

Style interacts with the UIFC that are described in Appendix A. Most of the widgets depend on `Style` to set background images and colors, font size and weight, and other parts of the visual elements.

Sausalito ships with one style file: `trueBlue.xml`. You can modify this file and save it as your own style file. You must give it a new name and create a new directory for your own style. The following example gives you high level instructions for creating a directory for your new style sheet, copying and modifying the style file.

IMPORTANT! You must make a copy of `trueBlue.xml`.

---

1. Change directories to `/usr/sausalito/ui/style`.
2. Copy the style file, `trueBlue.xml,` to `vendor_style.xml`.
3. Add any graphics or other files needed for your style file.

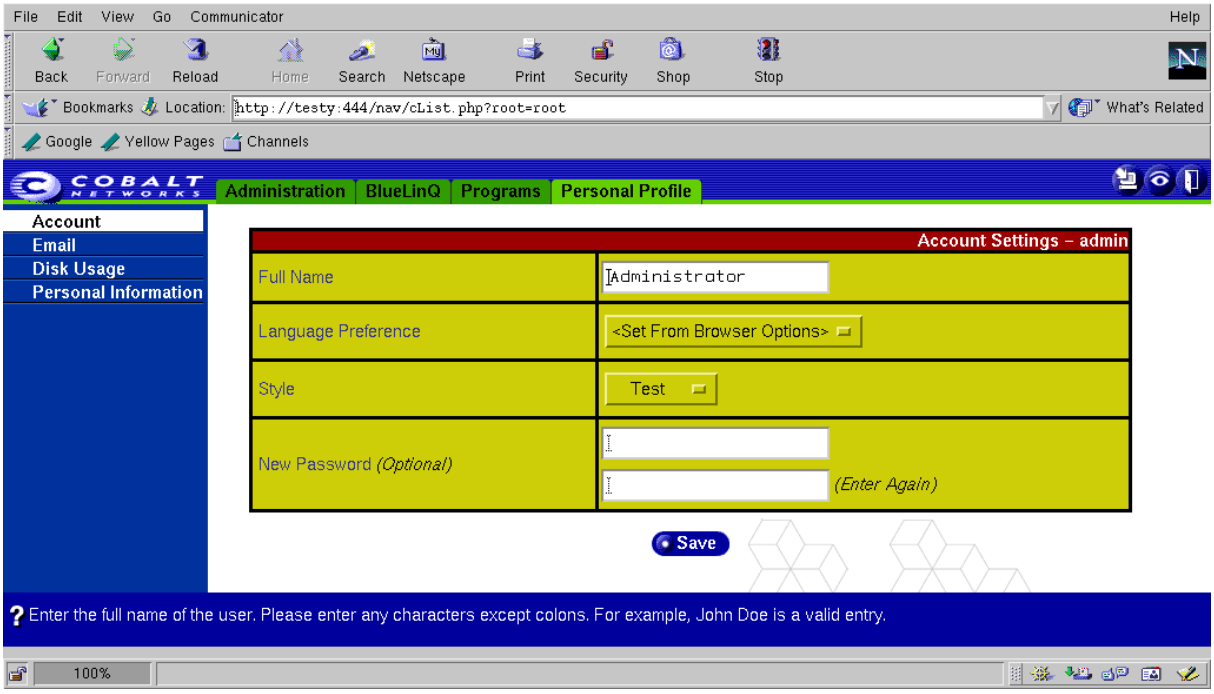
The following is an example of modifying the `trueBlue.xml` file. In this example, the following UI properties were changed, as shown in Figure 3—3 on page 3—11.

- `trueBlue.xml` was copied to `test.xml`; the word `test` is shown adjacent to Style in Figure 3—3 on page 3—11.
- The color value for the `aLinkColor` value was changed to `#0033CC`.
- The title alignment was changes from left to right by modifying the `tabAlign` value.
- The font size was changed by modifying `<property name="fontSize" value="12pt"/>` to `<property name="fontSize" value="18pt"/>`.

Similarly, changes were made to the background and divider colors of the table cells, and to the button font and colors, and to the alignment of the image.

Below is an example of a modified style file.

Figure 3—3 Modified Style File



Making Other Style Changes

You can make other style changes in addition to the ones shown in Figure 3—3 by making further modifications to your vendor_style file.

You can substitute your logo for Sun Cobalt™ s logo by searching for the line:

```
<property name="logo" value="/libImage/topLogo.gif"/>
```

and putting the `.gif` file for your logo in place of `topLogo.gif`.

Chapter 4

Using i18n and l10n in Sausalito

Chapter Contents

i18N: A World Tour

This chapter explains how to internationalize and localize Sausalito.

Terminology

This chapter uses two terms: internationalization, which is referred to as `i18n`, and localization, which is referred to as `l10n`.

Internationalization refers to the operation by which a set of programs are made aware of and are able to support multiple languages. This is a generalization process by which the programs are untied from calling only strings of a locale or other locale-specific habits, instead of connected to generic ways of doing the same. Program developers can use various techniques to internationalize their programs. GNU `gettext` offers one of these standards. For more information about `gettext`, see `http://www.gnu.org/manual/gettext/html_mono/gettext.html`.

Localization means the operation by which, in a set of programs already internationalized, the developer gives the program all needed information so that it can adapt itself to handle its input and output in a fashion that is correct for some native language and cultural habits. This is a particularisation process, by which language and cultural habits. This is a particularisation process, by which generic methods already implemented in an internationalized program are used in specific ways. The programming environment puts several functions to the programmers disposal which allow this runtime configuration. The formal description of specific set of cultural habits for some country, together with all associated translations targeted to the same native language, is called the *locale* for this language or country. Users achieve localization of programs by setting proper values to special environment variables, prior to executing those programs, identifying which locale should be used.

How Internationalization Works

The Sausalito architecture provides a simple-to-use interface to a database of localized strings used for internationalizing applications. This `i18N` interface is similar to the GNU `gettext` interface, and is, in fact, a higher-level wrapper than encapsulates GNU `gettext` functionality.

Like GNU `gettext`, the Sausalito `i18n` library allows developers to create their own databases of localized strings, and provides an interface for accessing that database from within applications. The Sausalito `i18n` wrapper library adds the following new functionality:

- Strings fetched from the library are subject to an interpolation process, in which user-supplied variables and even other internationalized strings can be automatically substituted into the localized string.
- Access to a set of routines for properly escaping the fetched strings for use in web applications, that is, for use in HTML documents or JavaScript programs.
- Automatic negotiation of the best possible locale, from a preference-ordered list of locales.

Using Domains, Tags, and Locales

The Sausalito `i18N` library manages a database of localized strings. Each application or module is granted it's own namespace within the database. This namespace is a called a *domain*. Within each domain, individual messages are identified a string known as the message tag.

When an application retrieves a message from the `i18N` database based on the message's domain and message tag, the `i18n` searches the database for a localized message that most closely matches the preferred locale.

Domains

A domain is a grouping for a similar set of resources, for example, the `sendmail` package can be a unique domain. In practical terms, localization strings are packaged by domain. Each domain defines the default language for its use in its own `prop` file. This file contains **only** a locale specification. The file is located in the same directory and locale property files except that its name is derived from the domain rather than the locale, for example, `cobalt.prop` for the Cobalt domain.

Developers retrieve message strings from the `i18N` database by specifying both the domain and the message identifier for each string.

Tags

A tag identifies a text string within a domain of strings for used in interpolation and `I18N`. The tag identifies the localized string. The localized string is identified by the `tag` specified in the function call and the `domain` specified within the `i18N` object. The locale used is the one negotiated during the creation of the `i18N` object. Developers retrieve message strings from the `i18N` database by specifying both the domain and the message id for each string using the "[[domain.msgid]]" tags.

Locale

Locales are specified by strings that start with an ISO-639 two-letter language code and an optional variant, all separated by underscore characters, for example, `en_US`.

In summary, the grammar for a locale identifier is:

```
locale_id := lang-code [ '_' country-code [ '_' variant-code ] ]
```

where `lang-code`, `country-code`, and `variant-code` are all alphanumeric codes defined in ISO-639.

Following are some example locale identifiers:

- `en`: Generic english
- `en_US`: English, American dialect
- `ja_JP_EUC`: Japanese, as spoken in Japan, the EUC variant

When the `i18n` library is initialized by an application, a comma-delimited list of locales is supplied to the `i18n` library. This list of locales indicates the various locales that the user can understand, in order of preference. The `i18n` library uses an intelligent algorithm to attempt to select the best available locale for each domain because not all domains support the same set of locales, for example: `en_US` or `en, ja`.

How Strings Are Added to the System

Adding new strings to the system in a three-step process:

1. A new `.po` file is created. This `.po` file defined all the message strings for one domain and one locale.

2. The `.po` file is compiled into an `.mo` file using the `msgfmt` tool.

3. The `.mo` file is placed in the appropriate directory beneath `/usr/share/locale/locale/LC_MESSAGES`.

Using Interpolation

When a string is fetched from the `i18n` library, it is subject to a process called interpolation. Interpolation allows user-supplied variables to be intelligently substituted into the string in various places. It also allows a string to contain references to other messages in the `i18n` database, with are expanded to full messages when interpolation occurs.

As a quick example, if the following string were stored in the `i18n` message string database:

```
"Hello, my name is [[VAR.name]]."
```

The `i18n_get` function is called such that the user supplied variable `name` was set to *Bob*, the following string would be returned by the library:

```
"Hello, my name is Bob."
```

Interpolation Rules

Every time a localized string is retrieved from the `I18N` database, it undergoes interpolation according to the rules defined below.

Rule 1. The string is subdivided into a list of tokens according to the following grammar:

```
string := token*
token := ( text | tag )
tag :=  [[  domain  .  tagname var*  ]]
var :=  ,  key  =  value.

text := escaped-string
domain := escaped-string
tagname := escaped-string
key := escaped-string
value := escaped-string
```

NOTE: The `tag` grammar interpolates the tag configuration in this format and substitutes variables into the string.

Rule 2. Strings are unescaped according to the following rules:

```
\n  -> newline
\b  -> backspace
\a
\f  -> formfeed
\n  -> newline
```

```
\r  -> return
\t  -> tab
\v  -> vertical newline
\(char) -> literal character
```

Rule 3. Tags are subject to the following expansion rules:

If the domain equals `VAR`, then variable expansion occurs. The variable specified in `tagname` is looked up in the current hash of variables. Its value is interpolated according to these rules and its value is substituted here.

If the domain is not equal to `VAR`, than the domain token is interpreted as the name of a `i18n` domain. The `tagname` token is interpreted as a message identifier, and the appropriate substring is fetched from the `i18n` database and interpolated.

Rule 4. The expanded unescaped tokens are reassembled into a single internationalized string.

The i18n Interface

Application developers use the following interface to fetch properly interpolated and escaped strings from the `i18n` database. Generally, the programmer first calls a constructor to create a new `i18n` object, perform a number of fetch operations, and then destroy the `i18n` object.

The `i18n` object performs its own memory management on strings that it returns. When the i18n object is destroyed, all memory allocated for various strings is freed automatically.

The `i18n` library is a C library, but Perl and PHP bindings are provided in addition to the C interface. These various interfaces to the `i18n` library are documented below.

The i18n C Language Interface

The function prototypes for the C language interface are in the following `include` file:

```
/usr/sausalito/include/cce/i18n.h
```

The link library for `i18n` is in these directories:

```
/usr/sausalito/lib/libi18n.a (library for static linking)
/usr/sausalito/lib/libi18n.so (library for dynamic linking)
```

The function interface for the C-language interface follows.

```
i18n_handle *i18n_new (char *domain, char *locales)
```

Summary: constructs a new `i18n` object, and returns a pointer to it.

Parameters

`domain:` identifies the default domain to use for operations where domain is omitted.

`locales:` a comma-delimited list of locale identifiers, listed in order of preference. This list of locales is used to choose the best locale for each domain when strings are retrieved from the database.

Returns: NULL for failure. Otherwise, returns a handle to a newly constructed `i18N` object.

```
void i18n_destroy (i18n_handle *handle)
```

Summary: destroys an `i18n` object, cleaning up all memory allocated by the `i18n` object.

Parameters

`handle:` the `i18n` object to be destroyed.

Returns: Nothing.

```
i18n_vars * i18n_vars_new ( void )
```

Summary: constructs a new object used to storing an associative array of variables for use by the various `i18n interpolate` and `get` functions.

Parameters

None

Returns: A pointer to a new `i18n_vars` object (a `GHashTable`).

```
void i18n_vars_add (i18n_vars *v, char *key, char *value)
```

Summary: adds a new key-value pair to the `i18n_vars` object. Copies of both the key and value are stored within the `i18n_vars` object. If this object is passed to a `i18n_interpolate` or similar function, it is used during interpolation to expand the VAR tags.

Parameters

`v`: a pointer to a valid i18n_vars object key -- a null-terminated string indicating the variable name

`value`: a null-terminated string indicating the value of the named variable.

Returns: Nothing.

```
void i18n_vars_destroy (i18n_vars *v)
```

Summary: destroys an i18n_vars object, and frees all memory associated with it.

Parameters

`v`: the pointer to the `i18n_vars` object to destroy

Returns: Nothing.

```
char *i18n_interpolate (i18n_handle *h, char *str,
i18n_vars *vars)
char *i18n_interpolate_html (i18n_handle *h, char *str, i18n_vars *vars)
char *i18n_interpolate_js (i18n_handle *h, char *str,
i18n_vars *vars)
```

Summary: These three functions provide direct access to the interpolation functionality within the `i18n` library. The `i18n_interpolate` function does not escape it's output at all.

The `i18n_interpolate_html` function performs an additional escaping expansion on the string it returns, escaping it appropriately for use in HTML content.

The `i18n_interpolate_js` function performs additional escaping, similar to the `i18n_interpolate_html` function, except that the string is escaped appropriately for use in JavaScript content.

Parameters

`h`: a pointer to a valid `i18N_handle` object.

`str`: a null-terminated string to subject to interpolation, as described above.

`vars`: a pointer to a valid i1`8n_vars` object. This object is used to find values for all variables needed during string interpolation.

Returns: A null-terminated string containing the results of interpolation on the string `str`. Optionally, this string can also have been escaped for use in HTML or JavaScript content.

```
char *i18n_get(i18n_handle *i, char *tag, char *domain,

i18n_vars *vars);

char *i18n_get_html(i18n_handle *i, char *tag, char *domain,

i18n_vars *vars);

char *i18n_get_js(i18n_handle *i, char *tag, char *domain,

i18n_vars *vars);
```

The `_get` functions are identical to the `_interpolate` functions, except that the message identified by domain and tag is fetched, and then interpolated.

```
char *i18n_strftime(i18n_handle *i, char *format, time_t time);

char *i18n_get_datetime(i18n_handle *i, time_t t);

char *i18n_get_date(i18n_handle *i, time_t t);

char *i18n_get_time(i18n_handle *i, time_t t);
```

Summary: these four functions get the time in the correct format for the current locale. Given a format that is identical to the one for `strftime` formats, the epochal time as found in `time_t` to the current locale settings

Parameters

`i18n`: the current `i18n` object.

`format`: the format to print the string in: `%x`, `%X`, and `%C` are useful.

t : The epochal time to format.

Returns: A pointer to a string formatted to the specified time

The i18n PHP Interface

Description: Constructor

```
Syntax: $i18n = new i18n (domain, languages)
```

Parameters

`domain`: sets the default domain to use for interpolation when domain is not explicitly specified.

`langs`: a comma-delimited list of supported locales specified in order of preference, for example, `jp, sh,` or `en`.

Returns: a new `i18n` object.

Object Methods

```
function i18N($domain = "", $langs = "")
```

Description: constructor

Parameters

`domain`: a string that describes the domain

`langs`: an optional string that contains a comma separated list of preferred locale. Most important locales appears first, that is, `en_US`, `en_AU, zh`, `de_DE`.

```
function get($tag, $domain = "", $vars = array())
```

Description: get a localized string

Parameters

`tag`: the tag of the string. Identical to the `msgid` string in the `.po` file

`domain`: the domain of the string in string. Identical to the `.po` or `.mo` file name without the extension. Optional. If not supplied, the one supplied to the `i18n` constructor is used

`vars:` a hash of variable key strings to value strings. Optional. If the hash contains `"name" => "Kevin"` and the string in question is `My name is [[VAR.name]]`, then `"My name is Kevin"` is returned.

Returns: a localized string if it is found or the tag otherwise.

```
function getJs($tag, $domain = "", $vars = array())
```

Description: get a localized string and encode it into JavaScript-friendly encoding

Parameters

`domain:` the domain of the string in string. Identical to the `.po` or `.mo` file name without the extension. Optional. If not supplied, the one supplied to the `i18N` constructor is used

`vars:` a hash of variable key strings to value strings. Optional. If the hash contains `"name" => "Kevin"` and the string in question is `"My name is [[VAR.name]]"`, then `"My name is Kevin"` is returned.

Returns: a JavaScript-friendly localized string if it is found or the tag otherwise.

```
function getHtml($tag, $domain = "", $vars = array())
```

Description: get a localized string and encode it into HTML friendly encoding

Parameters

tag: the tag of the string. Identical to the `msgid` string in the `.po` file

`domain:` the domain of the string in string. Identical to the `.po` or `.mo` file name without the extension. Optional. If not supplied, the one supplied to the `i18N` constructor is used.

`vars:` a hash of variable key strings to value strings. Optional. If the hash contains `"name" => "Kevin"` and the string in question is `"My name is [[VAR.name]]"`, then `"My name is Kevin"` is returned.

Returns: a HTML-friendly localized string if it is found or the tag, otherwise.

```
function interpolate($magicstr, $vars = array())
```

Description: get a localized string out of a fully qualified tag

Parameters

`magicstr`: the fully qualified tag of the format:
`"[[" . <domain> . "." . <tag> (. "," . <key> . "=" . <value>)* . "]]"`

`vars`: a hash of variable key strings to value strings. Optional.

Returns: a localized string or `magicstr` if interpolation failed.

```
function interpolateJs($magicstr, $vars = array())
```

Description: get a localized string out of a fully qualified tag and encode it into JavaScript-friendly encoding.

Parameters

`magicstr`: the fully qualified tag of the format:
`"[[" . <domain> . "." . <tag> (. "," . <key> . "=" . <value>)* . "]]"`

`vars`: a hash of variable key strings to value strings. Optional.

Returns: a JavaScript-friendly localized string or `magicstr` if interpolation failed.

```
function interpolateHtml($magicstr, $vars = array())
```

Description: get a localized string out of a fully qualified tag and encode it into HTML-friendly encoding

Parameters

`magicstr`: the fully qualified tag of format
"[[" . <domain> . "." . <tag> (. "," . <key> . "=" . <value>)* . "]]"

`vars`: a hash of variable key strings to value strings. Optional

Returns: a HTML-friendly localized string or `magicstr` if interpolation failed.

```
function getProperty($property, $domain = "", $lang = "")
```

Description: get a property value from the property file `/usr/share/locale/<locale>/<domain>.prop`. Properties are defined as <name>: <value>\n in the file. Each property is on its own line. Comments start with #.

Parameters

`property`: the name of the property in string

`domain`: the domain of the property in string. Optional. If not supplied, the one supplied to `i18N` constructor is used.

`langs`: an optional string that contains a comma separated list of preferred locale. Most important locales appears first, that is, `en_US, en_AU, zh, de_DE`. Optional. If not supplied, the one supplied to `i18N` constructor is used.

```
function getFile($file)
```

Description: get the path of the file of the most suitable locale, for example, if `/logo.gif` is supplied, locale `ja` is preferred, and/`logo.gif`, `/logo.gif.en` and `/logo.gif.ja` are available, `/logo.gif.ja` is returned.

Parameters

`file`: the full path of the file in question

Returns: the full path of the file of the most suitable locale.

```
function getAvailableLocales($domain = "")
```

Description: get a list of available locales for a domain or everything on the system

Parameters

`domain`: `i18n` domain in string. Optional

Returns: an array of locale strings.

```
function getLocales($domain = "")
```

Description: get a list of negotiated locales

Parameters

`domain`: `i18n` domain in string. Optional.

Returns: an array of locale strings, the first one being to most important, and so forth.

```
function strftime ($format = "", $time = 0)
```

Description: wrapper to `strftime()`

Parameters

`format`: the format parameter to `strftime()`

`time`: the epochal time

Returns: a `strftime()` formatted string

Internationalization Example

This is the code used to create this menu.

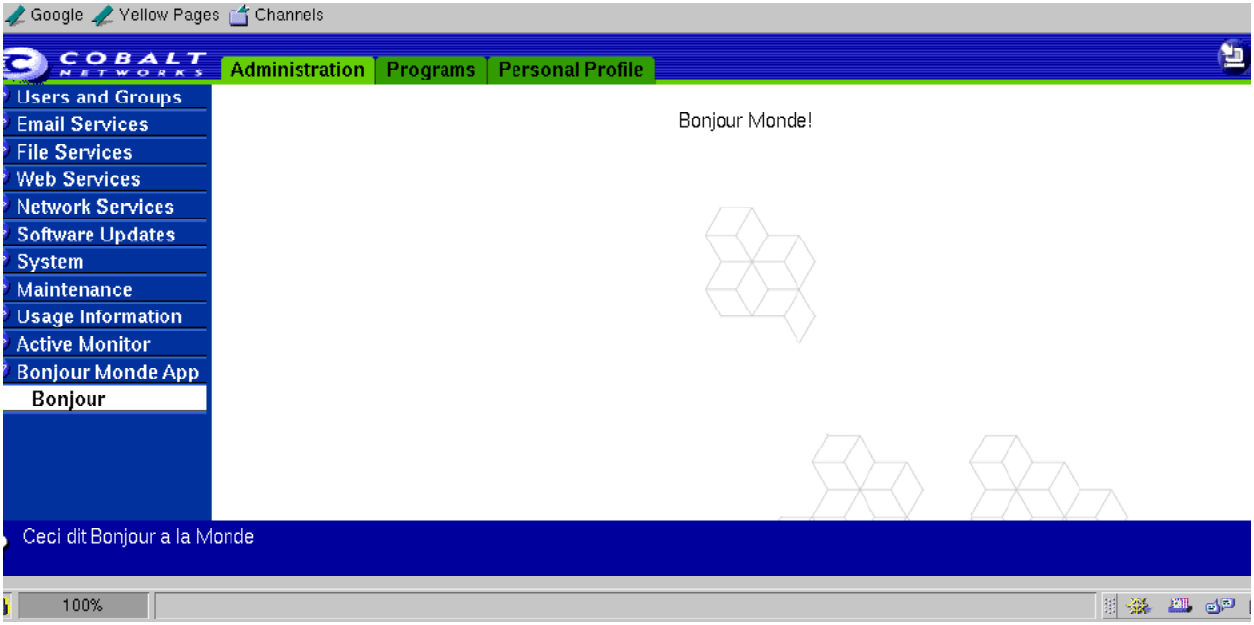
```
msgid "helloMenuItem"
msgstr "Bonjour"

msgid "helloMenuItem_help"
msgstr "Ceci dit Bonjour a la Monde"

msgid "helloMenu"
msgstr "Bonjour Monde App"

msgid "helloMenu_help"
msgstr "Ceci est l'application Bonjour Monde"

msgid "helloString"
msgstr "Bonjour Monde!"
Bon jour Monde!
```

This is the Makefile.

```
# Makefile for sample hello_world Sausalito application
```

```
VENDOR = sample
APP = hello

SRCS = en fr
I18NDIR = /usr/share/locale/

all:
        # nothing to do for all

install:
        for a in $(SRCS); do \
                DEST=$(I18NDIR)/$$a/LC_MESSAGES; \
                mkdir -p $$DEST; \
            msgfmt -e$$a/$(APP).po -e -o $$a/$(VENDOR)-$(APP).mo
; \
               install -o root -g root -m 644 $$a/*.mo $$DEST; \
        done
```

Figure 4—1 Internationalized Hello World example



Chapter 5

Introducing The Cobalt Configuration Engine

Chapter Contents

The Cobalt Configuration Engine (CCE)

If the user interface is the face of the Sausalito Architecture, the Cobalt Configuration Engine (CCE) is the brains. CCE is the agent by which the user interface affects changes on a system. It provides a unified interface to the task of configuring a system, and provides an abstraction layer between the user interface and the underlying system software.

CCE allows the development of a user interface that is truly flexible–it does not need to have intimate details about the underlying system. CCE is also designed to be extremely flexible, and allow developers to add new configuration options easily. Developers can extend CCE in the following ways:

1. Add configuration definitions to define new configurable applications (classes).
2. Add configuration information to extend the number of configurable options for an existing application (namespaces).
3. Add to the list of things that CCE does when configurable options change (handlers).

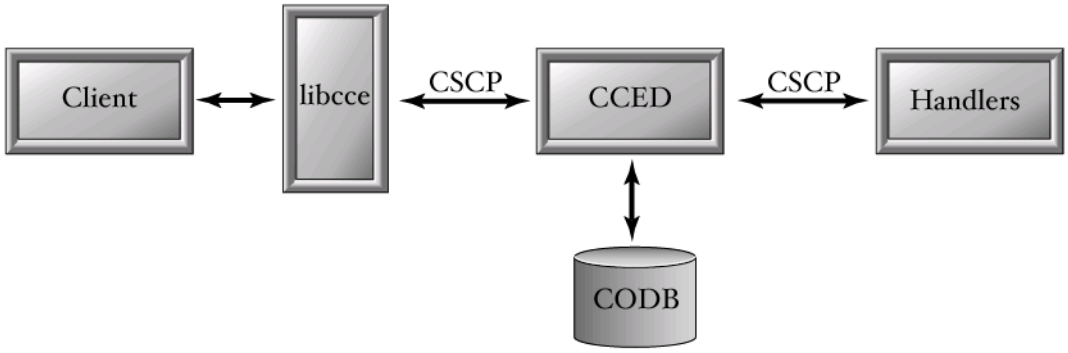
Basic Concepts

CCE is broken into several logical units for easier understanding. The major pieces of the CCE system are:

- The CCE daemon (`CCED`), which handles incoming connections, sessions, and signals.
- The Cobalt Object Database (`CODB`), which maintains the object store that reflects the current configuration of the system.
- The Cobalt System Configuration Protocol (`CSCP`), which is the protocol, or language, that CCE uses to communicate with clients.
- The CCE client library (`libcce`), which provides routines for clients to better access CCE via CSCP.
- The event handlers, which are the programs that make CCE changes take effect on the system itself.

CCED maintains the configuration state of the system in a set of objects representing the configurable applications, such as email and file sharing. These objects are stored internally by CODB. System configuration files are generated or modified by event handlers, which are triggered by a client making changes through CSCP. A client can be either a user interface, or a program written to interface with CCE.

Figure 5–1 CCE Block Diagram



How Data Flows Through CCE

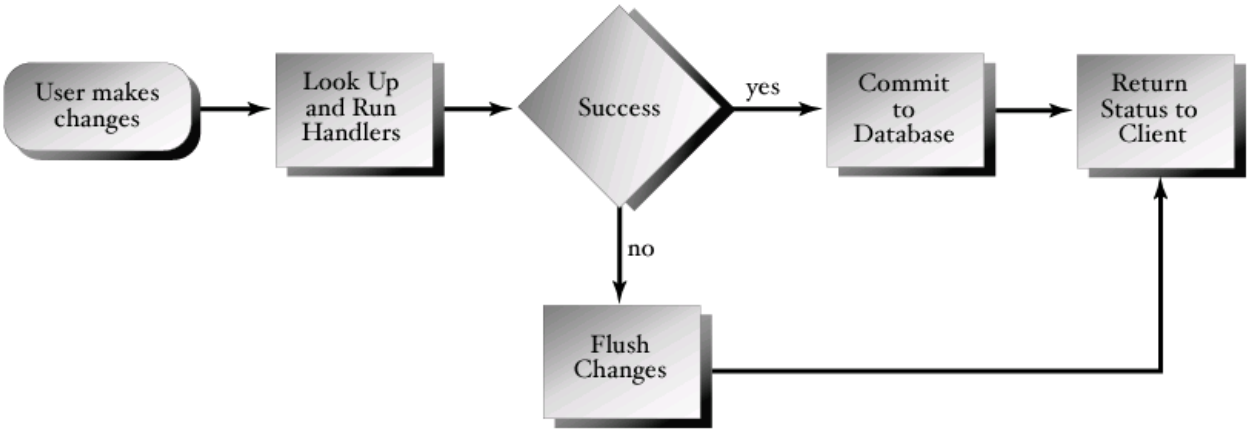
From start to finish, getting data to do the right things and go to the right places can seem complicated. The general flow of data through CCE is as follows:

- Packages register via configuration files for notification of when properties of objects change, or when objects are created or destroyed, which are commonly known as `events`.
- CCED listens for incoming clients.
- A client connects to CCED, which communicates using the CSCP protocol.
- The client gets or sets properties, or creates or destroys objects to configure the system.
- CCED determines which handlers need to run to actuate events from the client, and runs them.
- The handlers communicate with CCED, if needed, via CSCP.
- The handlers each do their work and exit, indicating their state of success. See “Method: bye( )” on page 5–34. If all handlers succeed, the changes are saved to the CODB. Otherwise, changes are ignored and discarded.

- CCED returns the status of the transaction to the client via CSCP.

Figure 5–2 illustrates the flow of CCE data.

Figure 5–2 CCE Process Flow



The CCE Daemon

The CCE daemon (CCED) is the server process that implements the core of CCE. CCED accepts incoming client connections on a UNIX domain socket, and initiates the CSCP protocol; see "The Cobalt System Configuration Protocol (CSCP)" on page 5–5. Each incoming connection is handled by a child process of the master CCED process, leaving the master process to handle new connections and signals. While active, the child process is responsible for running handlers, maintaining and updating the object database, and for communicating with the client. The master process also catches signals delivered to it, such as an interrupt signal, and distributes the signal to all the children, accordingly.

To preserve data integrity, all CSCP write operations for all clients are serialized. This does not affect the performance of the system, because there are not typically multiple simultaneous administrative sessions. For several reasons, including security, file system access, and handler access, CCED must run as `root`. Users must authenticate to CCED to

perform most tasks in order to protect the system: see “The AUTH Command” on page E–6. This authentication is passed through the Linux system of Pluggable Authentication Modules (PAM). For more information on security in CCE, see “Introducing Cobalt Security” on page 7–1.

CCED Command-Line Parameters

Usually, CCED does not need command-line parameters. However, for debugging handlers or CCE itself, it is sometimes useful to change certain aspects of CCED’s behavior. The following command-line parameters are available:

Table 5–1 CCED Command-Line Parameters

Parameters	Description
-c *directory*	set the handler configuration directory, /usr/sausalito/conf is the default
-d *number*	set the debug mask; 0 = no debugging (default), 0xffffffff = full debugging and profiling
-nd	do not run as a background daemon
-nf	do not fork child processes, handle only one client
-nh	do not run any handlers
-ro	read-only; do not save database changes; implies -nh
-st *seconds*	set the client `authkey` timeout, 1 hour (3600 seconds) by default
-V	verbose
-v	print version information and exit
-vv	print even more version information and exit

The Cobalt System Configuration Protocol (CSCP)

The Cobalt System Configuration Protocol (CSCP) is a simple protocol for communication between clients and the CCE, and between the CCE and event handlers. It is a text-based, newline delimited protocol, similar in form to FTP or HTTP. It is simple enough to be understood and debugged without the need for special tools.

To use CCE, you must use CSCP. The simplest way to use CSCP is with the command-line tool `/usr/sausalito/bin/cceclient`. This tool provides full access to CSCP directly, and is similar to using telnet to connect to TCP services. Wrapper libraries are provided for several popular languages to make CSCP easier to use from within programs.

For detailed information about the protocol specification, see Appendix E, “Cobalt System Configuration Protocol”.

The Cobalt Object Database (CODB)

The Cobalt Object Database (CODB), is similar in many respects to both traditional databases and object systems. It also differs in some significant ways. Every object stored within CODB has a unique identifier, its Object ID (OID), which CSCP uses to identify instances. Like traditional relational databases, CODB has a query language that allows the developer to access stored data. Unlike a traditional database, CCE uses CSCP, rather than SQL.

The traditional form of object manipulation is through object methods. These methods encapsulate and protect object data, stored in properties. CODB, by design, takes a different approach. The Sausalito system deals exclusively with properties. Unlike traditional object systems, there are no directly executing methods in Sausalito. Instead, Sausalito provides events and event handlers, which act as method code.

Schemas

The structure of objects within CODB is defined by *schemas* which are provided by third-party vendors, Schemas, in the form of schema definition files, provide the `class`, `property`, and `typedef` definitions necessary to impose order on the data within CODB. The syntax of a schema definition file is simple XML, and is very flexible.

How to Read XML Syntax Descriptions

Before proceeding, it is prudent to briefly cover the pieces that make a file XML. XML is a plain-text file format, similar to HTML (or their common ancestor SGML). XML files are parsed, and the data in them is stored in a manner that is useful to the controlling application.

Whitespace

Throughout XML files, most whitespace characters (spaces, tabs, and newlines) are ignored. The only exceptions to this rule are within quoted strings and within the content field of an element. In these cases, called significant whitespace, whitespace is preserved.

Symbols

To better represent the syntax used in this explanation, some symbols are necessary. Table 5–2 explains symbols herein.

Table 5–2 Symbols Used in Schemas

Symbol	Definition
SP	represents one whitespace character (space, tab or newline).
SQ	represents one single-quote (’).
DQ	represents one double-quote (").
asterisk (*)	represents zero or more occurrences of the previous expression.
plus (+)	represents one or more occurrences of the previous expression.

Elements and Content

All XML files consist of one or more `elements`. Each element has a case-insensitive name and a set of zero or more `attributes`. Elements can, but are not required to have `content`. Each element is begun by a an `opening tag` with the following form:

```
"<" SP* name SP* attribute* SP* ">"
```

The content field follows the opening tag. Content fields are free form, and all characters are retained, including whitespace. The content of one element can be, and frequently is, one or more child elements. This containership is arbitrarily deep, and is defined by the specific XML format being used. The content field is terminated by a `closing tag` of the following form:

```
"</" SP* name SP* ">"
```

Because the content field is optional, it is frequently empty. A second form of opening tag is allowed, which indicates the absence of a content field:

```
"<" SP* name SP* attribute* SP* "/>"
```

Attributes

As noted above, an element can have zero or more `attributes`. Attributes are always key-value pairs, and the value is always a quoted string. Attribute keys are always alphanumeric, and, like element names, are not case-sensitive. Attributes have the following form:

```
SP+ key SP* "=" SP* QU value QU
```

Comments

In addition to elements, XML files can include `comments`. Comments can be outside of any element, or in the content of any element. Comments begin with the string `<!--` and end with the string `-->`. Any text within a comment is ignored.

Escape Sequences

Because some characters, such as < and > are used by the XML language itself, it is necessary to use an alternate sequence of characters, called an escape sequence, to represent these reserved characters. The following escape sequences are recognized by XML:

Table 5–3 XML Escape Sequences

Literal Character	Escape Sequence
< (less-than)	<
> (greater-than)	>
& (ampersand)	&
’ (apostrophe)	'
" (quote)	"
(space)	

Sample XML

```
<!-- This is a sample XML file, illustrating syntax -->
<XMLElement NAME="Sample">
    <SubElement name="Sub Sample 1">
        This is &lt;content&gt; for a &quot;SubElement&quot;
    </SubElement>
```

```
    <SubElement Name = "Sub Sample 2" Color="green"></SubElement>

    <SubElement
        Name="Sub Sample 3" Note = "&amp;amp;"/>
</XMLElement>
```

Schema Syntax

Schema definition files can include any of the following elements:

- SCHEMA
- CLASS
- PROPERTY
- TYPEDEF

Syntax: SCHEMA

A `SCHEMA` is provided to identify a complete schema definition to the system. This element provides such information as schema name, vendor, version and any other information a vendor might find useful to store with their schema definition. All child elements of a schema are grouped together by the schema definition.

If no `SCHEMA` element is defined, or other top-level elements are defined, the non-schema-wrapped elements of the description file are assumed to be part of a schema with `NAME` set to the current filename (minus the `.schema` extension), and `VENDOR` and `VERSION` set to the empty string (""). Sun Cobalt™ recommends that every schema description file contain explicit SCHEMA elements, rather than rely on the default behavior.

```
Element name: "SCHEMA"

Required attributes: "NAME", "VENDOR", "VERSION"

Optional attributes: any

Required content: none

Optional content: "CLASS" or "TYPEDEF" elements
```

```
Valid Parents: none
```

Table 5–4 SCHEMA Attributes

Attribute	Description
NAME	The vendor-assigned name of the schema. This can be any string.
VENDOR	The unique name of the schema's vendor. This can be any string.
VERSION	The vendor-assigned version of the schema. This can be any string, but by convention is an integer or floating point number for example: "1" or "3.1415".

Syntax: CLASS

A CLASS is the formal definition of an object's structure. An object has all the properties of its CLASS, and only the properties of its CLASS.

```
Element name: "CLASS"

Required attributes: "NAME", "VERSION"

Optional attributes: "NAMESPACE"

Required content: none

Optional content: "PROPERTY" elements

Valid Parents: "SCHEMA"
```

Table 5–5 CLASS Attributes

Attribute	Description
NAME	The unique name of the class being defined, or the name of the class being extended. This must be a C-style symbol, that is, it must start with a letter or underscore (_), followed by any number of letters, digits, or underscores. NAME should, per convention, start with an upper-case letter, for example: "Foo".
VERSION	The version number of this class structure. This can be any string, but by convention is an integer or floating point number for example: "1" or "3.1415".
NAMESPACE	The optional name of the namespace being defined for the specified CLSASS. This follows the same rules as NAME, with the exception that NAMESPACE can be a blank string (""), or be unspecified.

Syntax: PROPERTY

A PROPERTY is a sub-element of a CLASS. A single PROPERTY defines a single datum. CLASSES get their utility from their PROPERTY elements.

```
Element name: "PROPERTY"

Required attributes: "NAME", "TYPE"

Optional attributes: "DEFAULT", "OPTIONAL", "ARRAY", "READACL",
     "WRITEACL"

Required content: none

Optional content: none

Valid Parents: "CLASS"
```

Table 5–6 PROPERTY Attributes

Attribute	Description
NAME	The name of the property. This must be a C-style symbol. See “Syntax: CLASS” on page 5–10 syntax for the NAME attribute. All properties beginning with an underscore (_) are reserved for future use.
TYPE	The data type of the property. This must be a valid TYPEDEF name. Type bindings are resolved after all schemas are loaded, so you can use a typedef before it is defined. A PROPERTY with a TYPE that does not exist will fail all data validation.
DEFAULT	The default value of the property, used when the property is unassigned. This can be any value that is valid for the specified TYPE. If DEFAULT is unspecified, the default value is an empty string (""), which might be valid for the PROPERTY.
OPTIONAL	Defines whether the property can be the empty string ("") in addition to a valid datum. This can be any string or unspecified. If unspecified or assigned the value "" or "0" (zero), the optional flag is set to false, otherwise, the optional flag is set to true.
ARRAY	Defines whether the property is an array of the specified TYPE. This can be any string or unspecified. If unspecified or assigned the value "" or "0" (zero), the array flag is set to false, otherwise, the array flag is set to true. When set to true, data for this property is assumed to be an unbounded array of data, of the type specified.

Table 5–6 PROPERTY Attributes

Attribute	Description
READACL	Defines the read access rule list for the property. This is a comma or space delimited list of security rules. If left blank or unspecified, the default value is ruleUser.
WRITEACL	Defines the write access rule list for the property. This is a comma or space delimited list of security rules. If left blank or unspecified, the default value is ruleAdmin.

Syntax: TYPEDEF

A TYPEDEF is a mechanism to build on the basic data typing provided by CCE. A TYPEDEF is a symbolic name given to a definition of a type, and is used by a PROPERTY to validate its data.

```
Element name: "TYPEDEF"

Required attributes: "NAME", "TYPE", "DATA"

Optional attributes: "ERRMSG"

Required content: none

Valid content: none

Valid Parents: "SCHEMA"
```

Table 5–7 TYPEDEF Attributes

Attribute	Definition
NAME	The symbolic name for the type. This must be a C-style symbol. See “Syntax: CLASS” on page 5–10 syntax for the NAME attribute.
TYPE	The validation class for the TYPEDEF. This must be either re or extern.
DATA	The TYPE appropriate data validator. For re TYPEs, it should be a valid regular expression. For extern TYPEs, it should be the path to an external program. The program should read the data from standard input, and return 0 if the data is valid, or non-zero if it is invalid.
ERRMSG	The error message returned by CCE when invalid data is written to an instance of this TYPEDEf. This can be any string, or unspecified.

Sample Schema Definition File

```
<SCHEMA
     NAME="Sample Schema"
     VENDOR="Sun Cobalt"
     VERSION="3.1415">

     <!-- Some classes, properties, namespaces, and types -->
     <CLASS name="SampleClass" version="12345">
         <PROPERTY name="name" type="sample_type" default="new"/>
     </CLASS>

     <CLASS name="SampleClass" namespace="Demo" version="6.02e23">
         <PROPERTY name="name" type="sample_type" default="123"/>
     </CLASS>

    <TYPEDEF name="sample_type" type="re" data="[A-Za-z0-9]*" />

    <CLASS name="SampleClass2" version="2.7183">
        <PROPERTY
            name="name"
            type="foo_type"
            default="new"
            optional="1"
            array=""
            readacl="ruleAdmin"    writeacl="ruleAdmin"
    />
    </CLASS>

    <TYPEDEF
        name="foo_type"
```

```
            type="re"   data="[A-Za-z0-9]*"
            errmsg="Yowie!  You can&apos;t do that with a foo_type!"
        />
</SCHEMA>
```

Handler Registration

The format of a CCE handler configuration file (`conf`) is very simple with two or three whitespace-delimited fields per line, and one or more lines per file. Each line has the following format:

```
event <whitespace> handler <whitespace> stage
```

Any line that begins with a hash (#), or is blank is ignored.

Events

The event field defines the circumstances upon which the handler is run. The event field follows the form:

```
class.property
```

This registers the specified handler to run whenever the specified class property is modified. To register a handler on an object's creation or destruction, use the special properties `_CREATE` or `_DESTROY`. To register a handler on the modification of any property of a class, use the special property `*` (asterisk).:

Table 5–8 Valid Events

Event	Condition
_CREATE	When an object of the specified class is created.
_DESTROY	When an object of the specified class is destroyed.
propertyname	When the specified property of the specified class is modify.
*	When any property of the specified class is modified.

Handlers

The handler field defines the type of handler, and the type-specific handler details. It has the form:

```
type:details
```

The details of the handler depend on the type specified. The following types are available:

- `exec`, which executes the file named in the details field.
- `perl`, which sends the Perl script named in the details field through a persistent Perl process, for improved Perl performance.
- `test`, which sends the contents of the details field to the standard output of `CCED`.

For `exec` and `perl` type handlers, which have a path name in the details field, some path expansion is performed. If the details field is a relative path (does not start with a `/`), the default handler path `/usr/sausalito/handlers/` is prepended to the details field.

Stages

All handlers are run in one of several stages, and can thereby ensure some relative ordering. The available stages are `VALIDATE`, `CONFIGURE`, `EXECUTE`, `TEST`, and `CLEANUP`. The stage field is optional, and if left off, is assumed to be `EXECUTE`. Since handlers within a stage are not guaranteed to run in any specific order, a single application can register multiple handlers in each stage. Each stage's name suggests what it can be used for. The `VALIDATE` stage, for example, should be used by handlers that do not make any changes, but instead verify that the requested event can be performed.

The final stage, `CLEANUP`, is meant for handlers which can not be undone. Handlers which register for this stage must not exit with a failure status, or the system can be left in an inconsistent and unrecoverable state.

File Naming

When searching for handler registration files, `CCED` will do a recursive search of the handler configuration directory. By default, this directory is `/usr/sausalito/conf`, but can be specified with the `-c` option to `CCED`, see "CCED Command-Line Parameters" on page 5–5. All files that end with `.conf` are parsed as handler registration files. The only exception to this is that any file or directory that begins with a dot (`.`) is ignored.

Sample Handler Registration File

```
# Register handlers for Class from AVendor
Class._CREATE   exec:/opt/AVendor/Class/Class_create       configure
Class._DESTROY  exec:/usr/sausalito/handlers/Class_destroy test
Class.*         exec:AVendor/Class/Class_mod
Class.property  perl:/usr/sausalito/bin/Class_prop.pl      validate
```

CCE Libraries

In order to make CSCP easier for programmers to use, code libraries are provided in several popular languages: C, Perl and PHP. These libraries insulate the users from the details of the CSCP protocol, and perform all the necessary conversions and transformations of data. The libraries are used both in the construction of the new user clients to the CCE server and to create handlers that extend the functionality of the CCE system.

All the libraries are similar in build and syntax, though there are minor variations among them to accommodate for language-specific feature sets. Each library builds functional wrappers around CSCP commands, as well as providing higher level functions for common tasks.

These libraries are under constant development, and will grow and evolve. As the libraries grow, some aspects of the library interfaces will inevitably become deprecated. All deprecated features will be retained for some time, for compatibility, but will eventually be removed from the libraries, so it is essential that new development not rely on deprecated features.

C

The C language library for CCE is, of course, available for C programmers. The C library can also be used as the basis for bindings in other languages, such as PHP, to access CSCP. This library uses some opaque type and several accessor functions for those types. If you are familiar with object-oriented programming in languages such as Java or C++, you can think of these as methods for a class.

Dependencies and Headers

The header files for `libcce` are found in `/usr/sausalito/include`, which should be specified as part of a client applications include search path. The only header file `libcce` applications need to include is `<cce/cce.h>`. This header includes anything else it needs. In addition to this header, `libcce` has one dependency, a library called `glib`. When compiling an application that links `libcce`, you will also need to link `glib`. To link `glib` and `libcce` successfully with your program, Sun Cobalt™ suggests adding the following options to your build process:

- When compiling, specify the following as part of your build string:

  ```
  -I/usr/sausalito/include `glib-config --cflags`
  ```

- When linking, specify the following as part of your link string:

  ```
  -L/usr/sausalito/lib `glib-config --libs` -lcce
  ```

Note that in both of the suggested strings, the quotations are back-quotes, not apostrophes.

Datatypes

Data structures are fundamental to using `libcce`. Several datatypes are defined by the CCE library.

Struct: cce_error_t

This is a transparent structure, used to hold information about an error condition. The data members of this structure are:

- `cscp_oid_t oid:` the oid for which an error occurred
- `char *key:` the property for which an error occurred (may be `NULL`)
- `char *message:` the error condition

Struct: cce_handle_t

This opaque structure holds all the information about a connection to CCE. Almost every `libcce` function needs a pointer to a `cce_handle_t` structure to be passed in.

Enum: cce_handler_ret

This datatype is an enumeration of values for handlers to indicate success or failure. Values for this enumeration are directly related to CSCP values for the `BYE` command. The following are valid values:

- CCE_SUCCESS
- CCE_FAIL
- CCE_DEFER

Struct: cce_props_t

This opaque structure holds all the properties of a CODB object. Because it is opaque, several accessor functions have been provided. See “Enum: cce_props_state_t”.

Enum: cce_props_state_t

This is an enumeration of state information for `cce_props_t` structures. Every `cce_props_t` structure can represents a CODB object in one of various states:

- CCE_NONE
- CCE_MODIFIED
- CCE_CREATED
- CCE_DESTROYED

This information can be very useful to event handlers.

Typedef: cscp_oid_t

This type represents a CSCP object ID (`OID`). It is used to uniquely identify CODB objects in all `libcce` functions. This is an integral type; do not make assumptions about whether it is signed or unsigned.

Functions

Functions within `libcce` can be broken down into three major categories: CSCP commands, accessor functions for datatypes, and utility functions. First let's examine CSCP command functions. For more information on CSCP see Appendix E, "Cobalt System Configuration Protocol". All of these functions, unless otherwise noted, are available to both handlers and client applications.

Function: cce_auth_cmnd( )

```
char *cce_auth_cmnd(cce_handle_t *handle, char *user, char *pass);
```

This function is used to authenticate to CCE. The provided username and password are evaluated and an authentication key is returned. This function is available to handlers that want to change their authentication state. Handlers, by default, begin their sessions authenticated with full privileges.

Parameters

`handle`: a pointer to the currently connected CCE handle

`user`: a pointer to the authenticating username string

`pass`: a pointer to the authenticating password string

Returns

A pointer to the authentication key string on success; `NULL` on failure

Function: cce_authkey_cmnd( )

```
int cce_authkey_cmnd(cce_handle_t *handle, char *user, char *key);
```

This function is used to reauthenticate to CCE. When calling `cce_auth_cmnd`, a pointer to an authentication key is returned. This function accepts that `authkey` in place of a password. Like `cce_auth_cmnd`, this function is not needed by most handlers.

Parameters

`handle`: a pointer to the currently connected CCE handle

`user`: a pointer to the authenticating username string

`key`: a pointer to the authentication key

Returns

A positive integer on success; `0` on failure

`Function: cce_bye_cmnd( )`

```
int cce_bye_cmnd(cce_handle_t *handle);
```

This function is used by a client application to terminate the CCE session. It terminates the CSCP connection and free any internal structures of the passed in `cce_handle_t`. This function is strictly for non-handler use. Note the presence of `cce_bye_handler_cmd` for use by handler applications.

Parameters

`handle`: a pointer to the currently connected CCE handle

Returns

A positive integer on success; `0` on failure.

`Function: cce_connect_cmnd( )`

```
int cce_connect_cmnd(cce_handle_t *handle, char *path);
```

This function initiates a CSCP connection to CCE. Once connected, the passed `cce_handle_t` is initialized. The path parameter is optional, and if passed `NULL`, defaults to `/usr/sausalito/cced.socket`. This function is strictly for non-handler use. Note the presence of `cce_connect_handler_cmnd` for use by handler applications.

Parameters

`handle`: a pointer to the currently connected CCE handle

`path`: the file path to the CCE UNIX Domain Socket (`NULL` = default)

Returns

A positive integer on success; `0` on failure.

`Function: cce_create_cmnd( )`

```
cscp_oid_t cce_create_cmnd(cce_handle_t *handle, char *class,
   cce_props_t *props);
```

This function is used to create a new CODB object of the specified class. The initialized values for the new object is taken from the passed in `cce_props_t`.

Parameters

`handle`: a pointer to the currently connected CCE handle

`class`: a pointer to the class name string

`props`: a pointer to a `cce_props_t` holding initial values for the new object, or `NULL` for no initialized values

Returns

a `cscp_oid_t` representing the new OID on success; 0 on failure

Function: cce_destroy_cmnd( )

```
int cce_destroy_cmnd(cce_handle_t *handle, cscp_oid_t oid);
```

This function attempts to destroy the CODB object specified by the passed in OID.

Parameters

`handle`: a pointer to the currently connected CCE handle

`oid`: the OID of the desired object

Returns

a positive integer on success; 0 on failure

Function: cce_endkey_cmnd( )

```
int cce_endkey_cmnd(cce_handle_t *handle);
```

This function requests that CCE immediately end the period of validity for the current session's authentication key.

Parameters

`handle`: a pointer to the currently connected CCE handle

Returns

a positive integer on success; 0 on failure

Function: cce_find_cmnd( )

```
GSList *cce_find_cmnd(cce_handle_t *handle, char *class,
   cce_props_t *props);
```

This function performs a search of CODB for an object or list of objects that match the specified criteria properties. The returned list is not guaranteed to be in any predictable order.

Parameters

`handle`: a pointer to the currently connected CCE handle

`class`: a pointer to the class name string

`props`: a pointer to the properties structure holding the search criteria

Returns

a pointer to a `GSList` of `cscp_oid_t` items; `NULL` if no objects match the specified criteria

Function: cce_find_sorted_cmnd( )

```
GSList *cce_find_sorted_cmnd(cce_handle_t *handle, char *class,
   cce_props_t *props, char *sortkey, int sorttype);
```

This function, like `cce_find_cmnd`, performs a search of the CODB. Unlike `cce_find_cmnd`, however, this function returns its data in a sorted order.

Parameters

`handle`: a pointer to the currently connected CCE handle

`class`: a pointer to the class name string

`props`: a pointer to the properties structure holding the search criteria

`sortkey`: a pointer to the property or namespace.property string upon which to sort the data

`sorttype`:: an integer representing which sort method to apply: 0 = alphanumeric sort, 1 = numeric sort

Returns

a pointer to a `GSList` of `cscp_oid_t` items; `NULL` if no objects match the specified criteria

Function: cce_get_cmnd( )

```
cce_props_t *cce_get_cmnd(cce_handle_t *handle, cscp_oid_t oid,
  char *namespace);
```

This function attempts to retrieve the contents of the specified CODB object.

Parameters

`handle`: a pointer to the currently connected CCE handle

`oid`: the object ID of the desired object

`namespace`: the namespace to retrieve, `NULL` for no namespace

Returns

a pointer to a `cce_props_t` structure holding the representation on the requested object on success; `NULL` on failure

Function: cce_names_class_cmnd( )

```
GSList *cce_names_class_cmnd(cce_handle_t *handle, char *class);
```

This function retrieves the list of namespaces available for a specified CODB class.

Parameters

`handle`: a pointer to the currently connected CCE handle

`class`: a pointer to the class name string

Returns

a pointer to a `GSList` of `char *` items; `NULL` if the class has no namespaces

Function: cce_names_oid_cmnd( )

```
GSList *cce_names_oid_cmnd(cce_handle_t *handle, cscp_oid_t oid);
```

Like `cce_names_class_cmnd`, this function gets the list of available namespaces for a class. However, this function accepts a particular OID, rather than a class name.

Parameters

`handle`: a pointer to the currently connected CCE handle

Returns

a pointer to a `GSList` of `char *` items; `NULL` if the object has no namespaces

Function: cce_set_cmnd( )

```
int cce_set_cmnd(cce_handle_t *handle, cscp_oid_t oid,
   char *namespace, cce_props_t *props);
```

This function attempts to set the specified properties in the specified CODB object.

Parameters

`handle`: a pointer to the currently connected CCE handle

`oid`: the object ID of the desired CODB object

`namespace`: a pointer to the namespace string, `NULL` for no namespace

`props`: a pointer to the property structure holding the new values for the object

Returns

a positive integer on success

0 on failure

Function: cce_whoami_cmnd( )

```
cscp_oid_t cce_whoami_cmnd(cce_handle_t *handle);
```

This function returns the object ID of the currently authenticated session.

Parameters

`handle`: a pointer to the currently connected CCE handle

Returns

the object ID of the user authenticated in the current session

Function: cce_bad_data_cmnd( )

```
int cce_bad_data_cmnd(cce_handle_t *handler, cscp_oid_t oid,
   char *namespace, char *key, char *reason);
```

This function allows a handler to report a problem with a specific piece of data to CCE.

Parameters

`handle`: a pointer to the currently connected CCE handle

`oid`: the object id in question

`namespace`: a pointer to the namespace string in question

`key`: a pointer to the property name string in question

`reason`: an string explanation of the problem

Returns

a positive integer on success

0 on failure

Function: cce_bye_handler_cmnd( )

```
int cce_bye_handler_cmnd(cce_handle_t *handle,
   cce_handler_ret status, char *message);
```

This function provides a handler-specific method of terminating a CCE connection. Handlers can exit with multiple success values (see “Enum: cce_handler_ret” on page 5–18). These values are important to CCE to know whether a handler has failed.

Parameters

`handle`: a pointer to the currently connected CCE handle

`status`: a symbolic exits status

`message`: an optional message for a failure exit status

Returns

a positive integer on success; 0 on failure

Function: cce_connect_handler_cmnd( )

```
cscp_oid_t cce_connect_handler_cmnd(cce_handle_t *handle);
```

This function, for use by handlers, is analogous to `cce_connect_handler_cmnd`. The reason for the distinction is in how handlers connect to CCE. This function will connect the CSCP file descriptor used by `libcce` to `stdin` and `stdout` of the calling program.

Parameters

`handle`: a pointer to the currently connected CCE handle

Returns

a positive integer on success; 0 on failure

Function: cce_last_errors_cmnd( )

```
GSList *cce_last_errors_cmnd(cce_handle_t *handle);
```

This function retrieves a list of errors and warnings for the most recent CCE command. While not truly a CSCP command, this function allows developers to access the error state of a CSCP connection.

Parameters

`handle`: a pointer to the currently connected CCE handle

Returns

a pointer to a GSList of CCE error structures on success; `NULL` on failure or if no errors or warnings are found

---

NOTE: In addition to the CSCP command functions, `libcce` provides several accessor functions for the various datatypes used throughout.

Function: cce_handle_new( )

```
cce_handle_t *cce_handle_new(void);
```

This function is used to create a new, disconnected `cce_handle_t` structure. To destroy a handle created by this function see `cce_handle_destroy` below.

Parameters

None

Returns

a pointer to a new CCE handle on success; `NULL` on failure

Function: cce_handle_destroy( )

```
void cce_handle_destroy(cce_handle_t *handle);
```

This function is the complement to `cce_handle_new`. It frees any resources allocated by the CCE handle. If the handle is connected, `cce_bye_cmnd` or `cce_bye_handler_cmnd` should be called first.

Parameters

`handle`: a pointer to the CCE handle to be destroyed

Returns

Nothing

Function: cce_props_count( )

```
int cce_props_count(cce_props_t *);
```

This function returns the number of properties stored in a CCE properties structure.

Parameters

`props`: a pointer to the CCE properties structure

Returns

An integer value representing the number of properties currently stored

Function: cce_props_destroy( )

```
void cce_props_destroy(cce_props_t *props);
```

This function is the complement to `cce_props_new`. It is used to release resources used by a properties structure.

Parameters

`props`: a pointer to the CCE properties structure

Returns

Nothing

Function: cce_props_get( )

```
char *cce_props_get(cce_props_t *props, char *key);
```

This function retrieves a pointer to a string, stored internally to the `cce_props_t`, representing the value of the property requested.

Parameters

`props`: a pointer to the CCE properties structure

`key`: the name of the desired property

Returns

a pointer to the value string on success; `NULL` on failure, or if the requested property is not found

Function: cce_props_get_new( )

```
char *cce_props_get_new(cce_props_t *props, char *key);
```

Because a CODB object, during a transaction, can have both changed (new) and old properties, `libcce` provides a way to access both. This function gets values only from the list of properties marked as changed (new).

Parameters

`props`: a pointer to the CCE properties structure

`key`: the name of the desired property

Returns

a pointer to the value string on success

NULL on failure, or if the requested property is not found

Function: cce_props_get_old( )

```
char * cce_props_get_old(cce_props_t *props, char *key);
```

Like cce_props_get_new, this function differentiates old and new data. This function only gets values from the unchanged (old) values list.

Parameters

props: a pointer to the CCE properties structure

key: the name of the desired property

Returns

a pointer to the value string on success

NULL on failure, or if the requested property is not found

Function: cce_props_new( )

```
cce_props_t *cce_props_new(void);
```

This function is used to allocate a new CCE properties structure.

Parameters

None

Returns

a pointer to a new CCE properties structure on success; NULL on failure

Function: cce_props_nextkey( )

```
char *cce_props_nextkey(cce_props_t *);
```

This function provides the ability to iterate over the items stored in a CCE properties structure. It retrieves the next property name in the internal queue.

Parameters

`props`: a pointer to the CCE properties structure

Returns

A pointer to the next property name string on success; `NULL` on failure

Function: cce_props_reinit( )

```
void cce_props_reinit(cce_props_t *);
```

The CCE properties structure has some iteration primitives (see “Function: cce_props_nextkey( )” on page 5–29). This function is used to re initialize the iteration primitives.

Parameters

`props`: a pointer to the CCE properties structure

Returns

Nothing

Function: cce_props_set( )

```
void cce_props_set(cce_props_t *props, char *key, char *value);
```

This functions sets the specified property to the specified value. If the property was previously set, the old value is overwritten by the new value.

Parameters

`props`: a pointer to the CCE properties structure

`key`: the property name to set

`value`: a pointer to a string value to store

Returns

Nothing

Function: cce_props_set_old( )

```
void cce_props_set_old(cce_props_t *, char *key, char *value);
```

This function, like `cce_props_set`, sets a property to a new value. Like `cce_props_get_old`, this function accesses only the property set that is marked as unchanged.

Parameters

`props`: a pointer to the CCE properties structure

`key`: the property name to set

`value`: a pointer to a string value to store

Returns

Nothing

Function: cce_props_state( )

```
cce_props_state_t cce_props_state(cce_props_t *);
```

This function retrieves the current state of the CCE properties structure. For details on the valid return values see “Enum: cce_props_state_t” on page 5–18.

Parameters

`props`: a pointer to the CCE properties structure

Returns

A `cce_props_state_t` which identifies the current state of the CCE properties structure

---

NOTE: The last set of functions provided by `libcce` are the utility functions. These are all helper functions to make various common activities simpler.

Function: cce_array_deserial( )

```
GSList *cce_array_deserial(char *str);
```

This function takes a pointer to a CCE internal array representation, and turns it into a linked list.

Parameters

- `str`: a pointer to a CCE internal array string

Returns

- a pointer to a GSList of strings on success
- `NULL` on failure

`Function: cce_list_destroy( )`

```
void cce_list_destroy(GSList *list);
```

This function is used to free a GSList of data. All elements and their data will be freed.

NOTE: **Do not** call this function with memory that is not dynamically allocated.

Parameters

`list`: a pointer to the GSList to be freed

Returns

Nothing

`Function: cscp_oid_from_string( )`

```
cscp_oid_t cscp_oid_from_string(char *string);
```

This function is used to read a string representation of a `cscp_oid_t` and convert it to a real `cscp_oid_t` representation.

Parameters

`string`: the string representation of the OID

Returns

the `cscp_oid_t` representation on success; 0 on failure

`Function: cscp_oid_to_string( )`

```
char *cscp_oid_to_string(cscp_oid_t oid);
```

This function is used to create a string representation of a `cscp_oid_t` from a real `cscp_oid_t` representation.

Parameters

`oid`: the OID

Returns

the string representation on success; `NULL` on failure

Perl

The Perl form of `libcce` implements an object-oriented interface. The same interface is used for communicating with CCED both in the context of a client, such as a user interface, and in the context of an event handler.

Module

The Perl library is located in `/usr/sausalito/perl/CCE.pm`. This directory should be included on the perl command line, for example:

```
#!/usr/bin/perl -I/usr/sausalito/perl
```

To include the CCE module in your perl program, add this line to the top of your program:

```
use CCE;
```

Creating a New Object

To begin using `libcce` in Perl, you must first create a CCE handle. This is analogous to the C library functionality (see “Function: cce_handle_new( )” on page 5–26). To create a new CCE object, use the following statement:

```
my $cce = new CCE;
```

Methods

All the functionality of the Perl `libcce` is provided as object methods of the CCE class.

`Method: auth( )`

```
$ok = $cce->auth($username, $password);
```

This method is used to authenticate to CCE.

Parameters

`$username`: the username for authentication

`$password`: the password for authentication

Returns

boolean success code

`Method: authkey( )`

```
$ok = $cce->authkey($username, $sessionid);
```

This method is used to resume a session, or reauthenticate to CCE.

Parameters

`$username`: the username for authentication

`$sessionid`: the sessionid for authentication

Returns

boolean success code

`Method: bye( )`

```
$ok = $cce->bye($status);
```

This method is used to end a CCE session. The optional status parameter is used by handlers to indicate one of the exit status conditions. See “Enum: cce_handler_ret” on page 5–18.

Parameters

`$status`: the exit status (handlers only)

Returns

boolean success code

Method: classes( )

```
($ok, @classlist) = $cce->classes();
```

This method retrieves a list of classes known by CCE.

Parameters

None

Returns

boolean success code

a list of classes

Method: connectuds( )

$ok = $cce->connectuds($filename);

This method is used exclusively by client applications (not handlers) to connect to CCE. If specified, the filename parameter will identify the UNIX domain socket to open, otherwise the socket `/usr/sausalito/cced.socket` is assumed.

Parameters

`$filename`: the (optional) name of a UNIX domain socket

Returns

boolean success code

Method: create( )

```
($ok, $badkeys, @info) = $cce->create($class, \%object);
```

This method is used to create a new CODB object. The provided object hash is used as the initial values for the new object. The OID of the new object can be retrieved by a call to the `oid` method; see “Method: oid( )” on page 5–41.

Parameters

`$class`: the class name to create

`\%object`: a reference to the hash of values with which to initialize the object

Returns

boolean success code, a hash reference of bad values, where the key is property name, and the value is the reason, or a list of informational and warning messages issued by CCE or handlers

Method: destroy( )

```
($ok, @info) = $cce->destroy($oid);
```

This method is used to destroy a CODB object.

Parameters

`$oid`: the oid to create

Returns

boolean success code or a list of informational and warning messages issued by CCE or handlers

Method: endkey( )

```
$ok = $cce->endkey();
```

This method is used to immediately end the validity of the current `authkey`.

Parameters

None

Returns

boolean success code

Method: find( )

```
@oidlist = $cce->find($class, \%criteria);
```

This method searches all instances of the specified class for a instances that match the specified criteria.

Parameters

`$class`: the class for which to search

`\%criteria`: a reference to a hash of property criteria

Returns

a list of OIDs that match the criteria

Method: findNSorted( )

```
@oidlist = $cce->findNSorted($class, $key, \%criteria);
```

This method searches all instances of the specified class for a instances that match the specified criteria. The resulting list is sorted numerically by the specified key property.

Parameters

`$class`: the class for which to search

`$key`: the property name upon which to sort

`\%criteria`: a reference to a hash of property criteria

Returns

a list of OIDs which match the criteria

Method: findSorted( )

```
@oidlist = $cce->findSorted($class, $key, \%criteria);
```

This method searches all instances of the specified class for a instances that match the specified criteria. The resulting list is sorted alphabetically by the specified key property.

Parameters

`$class`: the class for which to search

`$key`: the property name upon which to sort

`\%criteria`: a reference to a hash of property criteria

Returns

a list of OIDs which match the criteria

`Method: get( )`

```
($ok, $object, $old, $new) = $cce->get($oid, $namespace);
```

This method is used to fetch all of the attributes of a namespace or property from CODB. If the namespace is unspecified, the main namespace is retrieved.

Parameters

`$oid`: the desired OID

`$namespace`: the desired namespace name

Returns

a boolean success code, a reference to a hash of the requested properties, a reference to a hash of the previous values of any changed properties, if applicable, or a reference to a hash of the changed values of the property set, if applicable

`Method: names( )`

```
($ok, @namelist, @info) = $cce->names($oid);
```

This method is used to fetch the names of all of the namespaces for a particular OID.

Parameters

`$oid`: the desired OID

Returns

a boolean success code, a list of namespaces, or a list of informational and warning messages issued by CCE or handlers

`Method: set( )`

```
($ok, $badkeys, @info) = $cce->set($oid, $namespace, \%properties);
```

This method sets the properties in a CODB object or namespace to the properties specified. If the namespace is not specified, the main namespace is used.

Parameters

`$oid`: the desired OID

`$namespace`: the desired namespace name

`\%properties`: a reference to a hash of properties and values

Returns

a boolean success code, a list of namespaces, or a list of informational and warning messages issued by CCE or handlers

Method: whoami( )

```
($ok, $auth_oid) = $cce->whoami();
```

This method gets the OID of the currently authenticated user.

Parameters

None

Returns

a boolean success code or the authenticated OID

NOTE: Like the C library, there are a few functions that are applicable only to handlers.

Method: baddata( )

```
$ok = $cce->baddata($oid, $key, $msg);
```

This method is used exclusively by handlers to alert CCE and the calling client to a data error.

Parameters

`$oid`: the oid in question

`$key`: the property in question

`$msg`: the reason or explanation of the error

Returns

boolean success code

`Method: connectfd( )`

```
$ok = $cce->connectfd($readfd, $writefd);
```

This method is used exclusively by handlers to connect their input and output file descriptors to CCE. If the readfd or writefd parameters are omitted, `\*STDIN` and `\*STDOUT` are assumed, respectively.

Parameters

`$readfd`: the standard input file descriptor

`$writefd`: the standard output file descriptor

Returns

boolean success code

`Method: info( )`

```
$ok = $cce->info($msg);
```

This method is used exclusively by handlers to issue an informational message to CCE and the calling client.

Parameters

`$msg`: the message to pass

Returns

boolean success code

`Method: warn( )`

```
$ok = $cce->warn($msg);
```

This method is used exclusively by handlers to issue a warning message to CCE and the calling client.

Parameters

`$msg`: the message to pass

Returns

boolean success code

NOTE: Like the C version of `libcce`, the Perl library also provides several support and data-access methods.

Method: oid( )

```
$oid = $cce->oid();
```

This method is used to access the most recently created or found OID.

Parameters

None

Returns

the most recent OID

Method: event_oid( )

```
$oid = $cce->event_oid();
```

This method is used to access the OID which triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

the event-triggering OID

Method: event_namespace( )

```
$oid = $cce->event_namespace();
```

This method is used to access the namespace which triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

the event-triggering namespace

Method: event_property( )

```
$oid = $cce->event_property();
```

This method is used to access the property which triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

the event-triggering property

Method: event_object( )

```
$oid = $cce->event_object();
```

This method is used to access the hash of properties representing the object that triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

a reference to the event-triggering object hash

Method: event_old( )

```
$oid = $cce->event_old();
```

This method is used to access the hash of changed properties of the object that triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

a reference to the event-triggering changed properties hash

Method: event_new( )

```
$oid = $cce->event_new();
```

This method is used to access the hash of new properties of the object that triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

a reference to the event-triggering changed properties hash

Method: event_is_create( )

```
$oid = $cce->event_new();
```

This method is used to access the hash of created properties of the object that triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

a reference to the event-triggering changed properties hash

Method: event_is destroy( )

```
$oid = $cce->event_new();
```

This method is used to access the hash of destroyed properties of the object that triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

a reference to the event-triggering changed properties hash

`Method: event_is modify( )`

```
$oid = $cce->event_new();
```

This method is used to access the hash of modified properties of the object that triggered the current handler. It is meaningless in client applications.

Parameters

None

Returns

a reference to the event-triggering changed properties hash

Public Methods for CceClient (PHP)

Unfortunately, product schedules did not allow Sun Cobalt™ to complete this section. To see examples of this code, look at `/usr/Sausalito/ui/web`

Chapter 6

Making Sausalito-Aware Applications

Chapter Contents

Making Sausalito-Aware Applications

This chapter provides information on creating applications that run on the Qube 3. To create an application, you must create a module that includes all the components needed and structure it so that it can be easily installed by users, in a package file format (`.pkg`). This chapter lists the fields that you need to include so that the Qube 3 can display the appropriate information during the installation process. It also describes the appropriate directories, files, and resources for your application module.

About the Application Module

The application module is a self-contained bundle of files, directories, and resources required for a new capability. Depending on the type of module you are creating, you choose the appropriate level of integration. Some modules trigger both the user interface and the back end system; others are stand alone modules.

New modules can contain any or all of the following code:

1. User Interface (UI) modules
   - ¥ UI pages built using UIFC
   - ¥ Navigation nodes, such as adding buttons and menu items

   The Web mail service that is displayed on the Cobalt menu is an example of a service that is integrated only with the user interface and uses IMAP as its back-end system. The files for the user interface go into the `ui` directory; for more information about module directory layout, see Table 6—2 on page 6—5.

2. Internationalization modules
   - ¥ Internationalization resources to translate the user interface into other languages.
3. Back-end modules
   - ¥ CCE configuration files
   - ¥ CCE handlers

   Adding a user to the Qube 3 is an example of an instance that impacts only the back-end modules, where the existing user interface is used and the CCE configuration files and handlers are invoked.

4. Binary modules
   - ¥ Service binary and configuration files, for example, email modules have `SendMail` and `Majordomo` binaries and modify the configuration files for these binaries.
   - ¥ Databases that register users as they are created and notify event handlers about creating users. This type of module uses the existing user interface.

   These modules can be manually installed and completely unintegrated to the Cobalt User Interface (UI).

Naming Your Application Module

Developers must use unique vendor-specific names for modules to avoid name conflicts.

NOTE: Cobalt uses *base* in its module names, for example, `base-devel.mod`. Developers **must** differentiate their modules by naming the modules with a distinctive name, preferably a name that reflects their company or product, for example, *vendor_name*_module.

Building a New Service Module

A service module is a self-contained bundle of files or directories and resources required for a new capability, for example, an ecommerce product or a system backup product. New modules can contain any or all of the following:

- Navigation nodes `service.xml`
- User Interface (UI) pages built using UIFC `service.php`
- Internationalization resources `service.po`
- CCE configuration files `service.schema`, `service.conf`
- CCE handlers `serviceMod.pl`, `serviceMod.c`
- Service binaries and configuration `serviced`

NOTE: You can write handlers in any language. Cobalt provides bindings for C and Perl.

Cobalt enabling tools include:

- Standard directory structure document; see Figure 6—7 on page 6—22.
- Build tools to create loadable module files (scripts and a Makefile)

Making your Application into a Package

This section describes the skeleton module for Sausalito. By customizing the skeleton module for your needs, you can integrate seamlessly into the Cobalt configuration environment.

To build a service module:

1. Create handlers to interact with the Cobalt Configuration Engine (CCE). A configuration file goes in `glue/conf`; the actual handlers go in `glue/handlers`.
2. Create any user interface components, if necessary. These include web and menu page descriptors, which go in the `ui/web` and `ui/menu` directories, respectively.
3. Write any `locale` files; these go in the `locale` directory.
4. Look at `templates/spec.tmpl` and `templates/packing_list.tmpl`.

   NOTE: The default template to build RPM files is in `/usr/sausalito/devel/templates`. If you want to modify these templates, create a template directory in your module. Copy these files to your template directory and modify them as needed.

5. Look at the top-level `Makefile`. Adjust the variables to fit your situation.

The default build targets are `make all`, `make clean`, `make install`, and `make rpm`.

NOTE: A sample skeleton module is available in the Cobalt Developer web page. Go to `http://developer.cobalt.com/devnet/devtools.html` for the code sample and Readme file.

Here's some more information about the default `make` rules and expected file names:

Table 6—1 The top-level `Makefile` variables

Makefile Variables	Description
VENDOR	the vendor field for your module
VENDORNAME	the actual vendor name; this name can be the same as VENDOR
SERVICE	the name for the service
VERSION	version number

Table 6—1 The top-level `Makefile` variables

Makefile Variables	Description
RELEASE	release number
BUILDARCH	set to `noarch` if you don't want platform-specific RPMs generated.
XLOCALEPAT	set to a space-separated list of locale patterns to exclude
BUILDUI	packages all files in `ui/web` and `ui/menu`.
BUILDLOCALE	set to `yes` to build these components, create RPMs, and create a capstone RPM.
BUILDSRC	build the files is in the `src` directory.
BUILDGLUE	If `BUILDGLUE` is set to `yes`, packages all the handlers, object schemas, configuration files for event triggers, and conf files. If set to `no`, `BUILDGLUE` does no packaging.
DEFLOCALE	This locale is used for static HTML pages, for example, `en` or `ja`.

The BUILD variables determine which directories to include when calling the `clean`, `install`, and `rpm` targets.

The default `make` rules take the `BUILD?` variables and build up `ui`, `glue`, and `locale` RPMS, if requested. If any of these RPMS are generated, a *capstone* RPM is created as well. A capstone is a type of packing list for the RPMs.

Table 6—2 Module Directory Layout

Directories	Description
constructor	capstone constructors
destructor	capstone destructors
glue	handler and configuration modification code
ui	user interface and user interface menu code
locale	locale information and locale-specific UI pages
templates	user-modifiable template files used in packing list and RPM generation
src	`src` directory (optional)
RPMS	RPMS directory (optional)
SRPMS	source RPMS directory (optional)

The default `make` rules expect the following file layout:

1) `glue/conf/*`

   `glue/handlers/*`

2) `locale/localeX/$(SERVICE).po`

3) `ui/menu/*`

   `ui/web/*`

4) `constructor/*`

   `destructor/*`

The default `make` rules place these files in the following locations:

```
glue/conf/* -> $(CCEDIR)/conf/$(VENDOR)/$(SERVICE)/*
      glue/handlers/* -> $(CCEDIR)/handlers/$(VENDOR)/$(SERVICE)/*

locale/localeX/$(SERVICE).po ->
      /usr/share/locale/localeX/LC_MESSAGES/$(VENDOR)-$(SERVICE).mo

ui/menu/* -> $(CCEDIR)/ui/menu/$(VENDOR)/$(SERVICE)/*
      ui/web/* -> $(CCEDIR)/ui/web/$(VENDOR)/$(SERVICE)/*

constructors/* $(CCEDIR)/constructor/$(VENDOR)/$(SERVICE)/
      *destructors/* $(CCEDIR)/destructor/$(VENDOR)/$(SERVICE)/*
```

If your module does not contain compiled code, the above `make` rules should be all that you need for building a service module. Otherwise, you need to know about a couple additional `make` rules. First, `make` checks for Makefiles in the `glue`, `src`, and `ui` directories and uses them, if they are present. You must prepend the `PREFIX` environment variable on the install phase of the Makefile so that RPMs are properly generated.

In addition, the `make rpm` rule does not touch the `src` directory, so you must create any RPMs you want from separate specification files. `templates/packing_list.tmpl` should be updated to reflect any of these RPMs without version numbers. You should create a file with the same name as the RPM in the `rpms` directory to get the appropriate version included in the packing list.

Finally, you might need to edit `templates/rpmdefs.tmpl` to add additional build, install, and file targets for any files that you have. The `<begin [$%]VARIABLE>` sections in the `rpmdefs.tmpl` file correspond to the same `[VARIABLE_SECTION]` sections in `templates/spec.tmpl`. If you want to add something to `spec.tmpl` that is not dependent upon a single RPM, you can directly add it to `spec.tmpl`.

NOTE: If you have a VENDORNAME specified, `make` searches first in `{glue, locale, ui, src}/$(VENDORNAME)` for files before searching in the `glue, locale, ui,` and `src` directories.

Introducing Slush Barn, A Real-World Application

Here is an example of how to create a new Sausalito module. The goal of this example is to manage of barn of animals using a UIFC-enabled front-end while updating an XML file on the server. This example is included to help you better understand how the pieces of the Sausalito architecture work together to form a simplified means of creating web-based server administration tools.

The files created in making this module are listed in below. Although many files are needed for this module to work properly, each file is usually very short and serves its purpose. Details on the syntax and contents of each file type are listed throughout this manual.

NOTE: These code modules are given the vendor-specific name `slush` so that they are differentiated from Sausalito standard files. For a list of standard files, see Appendix D, Base Data Types .

- slush-barn.mod/Makefile
- slush-barn.mod/glue/conf/barn.conf
- slush-barn.mod/glue/handlers/Animal.pl
- slush-barn.mod/glue/schemas/animal.schema
- slush-barn.mod/glue/handlers/Animal.pl
- slush-barn.mod/ui/menu/barn.xml
- slush-barn.mod/ui/web/animal.php
- slush-barn.mod/ui/web/animalHandler.php
- slush-barn.mod/ui/web/slaughter.php
- slush-barn.mod/locale/en/animal.po

The data types are registered with CCE using a `typedef` and a class tag within the XML file `animal.schema`. The typedef called `animalType` defines a new type that might contain only the strings `Pig`, `Cow`, `Horse`, and `Chicken`. The class itself is called `BarnAnimal` and has two properties. The `name` property refers to the given name of that particular barn animal, and the `type` property is defined as being of type `animalType`. Each time an addition is made to our barn, a new instance of the class `BarnAnimal` is created and it s properties are filled with the data entered by the user.

Event handlers are also registered with CCE These event handlers are found within the `barn.conf` file. As defined, our handler (`Animal.pl`) is called upon any creation, change or destruction of a `BarnAnimal` instance. The handler in turn uses the data entered to create and update an XML file called `/etc/barn.conf`, but any types of service configuration can occur here. as the handler is run as root if triggered by the `admin` user.

The logic to the user interface is very simple. A listing of all the currently known animals is listed in the `barn.php` file using a `ScrollList` type. This type allows for multiple columns of data along with formatting rules making for a clear and distinct separation between the logic and presentation of the user interface. The `ModifyButtons` and the `AddButton` on this page link to a page called `animal.php`, which is generic in that it allows for both the creation and manipulation of animals within the barn. These actions are done in the page named `animalHandler.php`, which receives the `POST` of the `animal.php` page. To remove an animal from the barn, the `slaughter.php` page is called along with the `OID` of the object to be deleted.

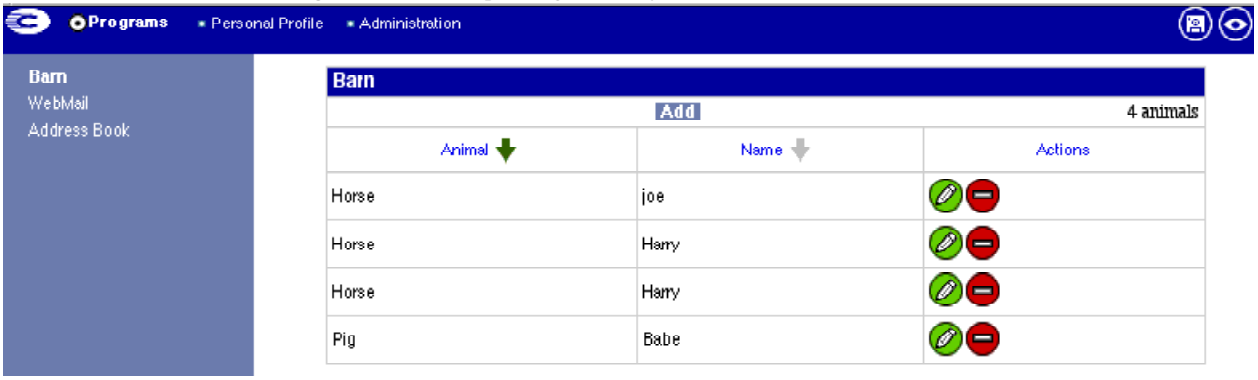
In order for our pages to be linked within our site, we need to create the XML tree node. This file is called `barn.xml` and contains information regarding that node. An ID is assigned to every node of the tree as a reference point when creating parent-child relationships.

In this example, you can manipulate objects in a barn.

NOTE: No animals were harmed in the making of this application.

The new page is shown in Figure 6—1.

Figure 6—1 Manipulating Barn Objects



How to Install your Package File on the Qube 3

There are two ways that packages can be installed on Qube:

- manually
- update server

Both these ways provide information about the package, that is, package meta-information, before the user installs the package. This meta-information includes fields with the package name, vendor, description, license, and whether package dependencies exists; these fields are described in Table 6—3 on page 6—12. This information is needed to properly display in the Qube UI details about the package before its installed. To provide this information, this information is included in the package list and the package information directories for each package.

Update servers alert you if they have new software for your Qube 3. When the Qube is alerted that there is a new version of software for the Qube, the update server and Qube have the following dialog:

1. The Qube 3 queries the server for information about new software. It provides details about the Qube including the packages installs, Qube identification, and so forth.
2. The update server replies with list of available packages with associated information, such as license and locale information. This informations corresponds to the `packing_list` and the contents of the `pkginfo` directory.

3. If an `InfoURL` field is specified, a popup window with the URL is displayed when you go to the install detail page. If an `InfoURL` field is not specified, a short description of the package is displayed.
4. Installation can be selected.

The events around the manual installation are as follows:

1. The user on the Qube enters the package location through either browser upload, URL download, or putting the file in `/home/packages`.
2. The Qube prepares the package for installation and displays the installation page. This informations corresponds to the `packing_list` and the contents of the `pkginfo` directory.
3. The contents of the installation page display a short description of the package that is to be installed.
4. Installation can be selected.

Installation Process

The following stages occur in the installation process:

- ¥ If the package requires the server to reboot, the user is prompted to reboot the machine.
- ¥ The install process looks first for a `splash` page If the `splash` page specifies the `pre-installation` option, it will look for an `index.cgi` or `index.php` page to call. It will pass in the following two variables a GET request to these files: `submiturl` and `cancelurl`.

NOTE: The `splash` page optionally specifies a pre-installation page, which allows developer to create a custom page for the package including license information. This page must be a CGI or PHP page that accepts `GET` requests.

- ¥ If the splash page doesn t exist and the license field does, **BlueLinQ** will present a standard license page containing the value of the license field.

NOTE: The Qube 3 software notification mechanism is called **BlueLinQ**.

- Once the user accepts the license (if there is a license), **BlueLinQ** checks package dependencies, and halts if there is a dependency error. If not, **BlueLinQ** runs the pre-installation scripts, install RPMS, and then runs the post-installation script. The scripts are located in the `scripts` directory of the package.

NOTE: **BlueLinQ** will install an RPM only if it is newer than any existing RPMs. If there is an existing RPM on the server, **BlueLinQ** increments the reference count each time you add a package with a RPM referenced in it. When you uninstall a package, the reference count is reduced. If the reference count for a package is less than one, **BlueLinQ** deletes the RPM.

Choices for the Installation Process

You can customize your installation. You can change the look and feel of install by opting to include:

- an `infoURL` field
- a splash page
- a generic license

The splash page must be a CGI or PHP file. The update process calls this CGI with the following URL variables set: `submitURL` and `cancelURL`.

Package Structure

The package file format is a `tar.gz` file. When you install a package file, **BlueLinQ** checks for the following items:

- whether the file is a `tar` file or a compressed `tar` file
- whether the file is signed

In packages for earlier Cobalt products, package files had the following elements:

- `packing_list`
- RPMs
- SRPMs

- `install_me` script

Packages for earlier Cobalt products had scripts that performed all installation tasks. Package dependency checking was done by the package itself. New packages have scripts that runs at specified times.The scripts deal with the following issues:

- pre-installation
- post-installation
- pre-uninstallation
- post-uninstallation

BlueLinQ runs these scripts as part of the installation. Package dependencies are based on vendor name, version number and package name. You can evaluate version number to determine if they are equal, less than, or greater than the target version. Sausalito currently checks a three-part field, for example, 1.0 or 1.1.2.

The new packing list format includes the following elements as shown in Table 6—3.

NOTE: All the information in the package list format is case-sensitive.

Table 6—3 Package List Format

Component	Description
[Package -- Version=1.0]	
`Vendor`	vendor name can include alphabetical characters, numbers, underscore (_), and the plus sign (+). Spaces and hyphens (-) are not permitted.
`VendorTag`	internationalizable vendor string
`Name`	*packagename* can include alphabetical characters, numbers, underscore (_), and the plus sign (+). Spaces and hyphens (-) are not permitted.
`NameTag:`	internationalizable package name string.
`Category`	category information can include alphabetical characters, numbers, underscore (_), and the plus sign (+). Spaces and hyphens (-) are not permitted.
`Location`	URL that specifies the package download location
`InfoURL`	additional information URL. Optional. Use this if you want to display a new site (as opposed to installing a package).

Table 6—3 Package List Format

Component	Description
`InfoURL` options	options that should be sent with to the URL, which can include serial number, product identifier (product), and vendor name (name).
`Version`	version of the package
`Version Tag`	Internationalizable version number.
`Size`	size in bytes (only used by the update server.)
`Product:` NOTE: use this field to specify as many products as you are including. Include one line for each package. You can use a regular expression to specify products, for example: (4000|4010|4100) WG.	Cobalt product requirements: for example, 4100WG or 4*nnn*WG. NOTE: 4000WG is the product number for the basic Qube 3. 4010WG is the product number for the Qube 3 with caching; 4100WG is the product number for the Qube 3 with caching and mirroring.
`PackageType`	specify `complete` or `update`
`Options`	`uninstallable, reboot, refreshui, refreshcce`
`LongDesc`	internationalizable long description
`ShortDesc`	internationalizable short description
`Copyright:`	internationalizable copyright string
`License`	internationalizable license information. Optional
`Splash`	pre-install, post-install, pre-uninstall, post-uninstall
`Depend` NOTE: Each dependency must be on its own line. See Package Dependency Model on page 6—16 for more information.	package dependencies. for example, `vendor:package.` The package won't show up in the new or updates pages if these dependencies aren't met. Here's what's expected: `vendor:package` vendor-package must exist. `vendor:package !` vendor-package must not exist. `vendor:package <=> version` vendor-package is less than, equal to, or greater than the specified version number. `vendor:package != version` vendor-package not equal to version.

Table 6—3 Package List Format

Component	Description
`VisibleDepend` NOTE: Each dependency must be on its own line. See Package Dependency Model on page 6—16 for more information.	just like Depend except that the package will show up in the new or updates lists even if dependencies aren't met.

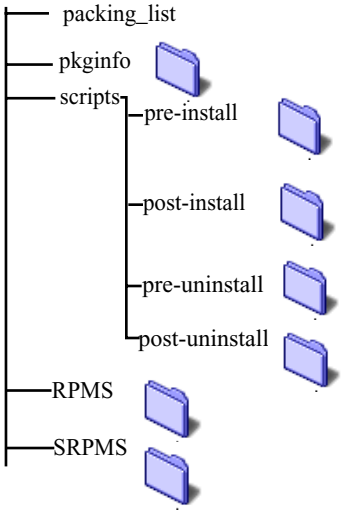
`Obsoletes`
NOTE: Each obsoletes must be on its own line. See Package Dependency Model on page 6—16 for more information.
	obsoletes vendor-packages
	`format:`
	`vendor:package`
	`vendor:package <=> version`

`RPM`	used only by the actual package
`SRPM`	used only by the actual package

NOTE: Internationalized strings are in the following format: `[[vendor]]`. If you are specifying strings within the `pkginfo` locale directory, then do not specify a domain. Sausalito specifies the domain for you. `pkginfo` locale strings cannot include locale tags within locale tags. You can include locale tags that refer to other domains.

Package files have the following structures. Figure 6—2 shows the package file structure.

Figure 6—2 Package File Structure



See Module File Hierarchy on page 6—22 for a more complete file hierarchy.

NOTE: The `packing_list` format for packages is very similar to the package part of the `package_list` update server packing list. You can use them interchangeably with the caveat that some fields are unused. For example, the update server information uses the `size` field. The packing list uses RPM, SRPM, and `fileName`.

The following features are only used by software update notification mechanism (**BlueLinQ**):

- `Size` (in bytes)
- `InfoURL`
- Location
- `PackageType`

The following fields are only used by actual package installation mechanism:

- `RPM`

- `SRPM`
- Options

Package Dependency Model

The dependency model allows you to restrict packages to particular Cobalt products, for example, the Qube 3. You can also include dependencies on other software packages. Finally, you can declare old packages obsolete.

The format for dependency requires that each dependency is on a separate line with a label denoting the type of dependency. Sausalito offers three types of dependency information:

- `Product:` Cobalt Product Dependency such that the package will install if other software products that are needed are not already installed. These are checked by product ID, for example `4000WG`. You can use a specific product, particular version, or you can use a Perl regular expression here.
- Package dependencies:
  - `Depend:` Normal package dependency based on the version number being less than (<), equal to (=), or greater than (>) the version number specified.
  - `VisibleDepend:` Visible dependency: same as `Depend` but is only useful for the software update mechanism. The packages that do not meet dependencies behave identically to the `Depend` in all other manners to new or update packages despite the fact that the package can t be installed.
- `Obsoletes:` Obsoletes packages name or name and optional version, less than (<), equal to (=), or greater than (>) the version number specified, which removes information about other packages of that name or version number specified.

Information for Installing Stand-alone Packages

The following are used in the actual package installation process but not in update server-supplied information. They are not used for the update server `pkginfo`.

- `RPM`
- `SRPM`
- `Options` (in a comma-separated list) include:
  - `reboot`
  - `refreshui`

- `refreshcce`
- `uninstallable`

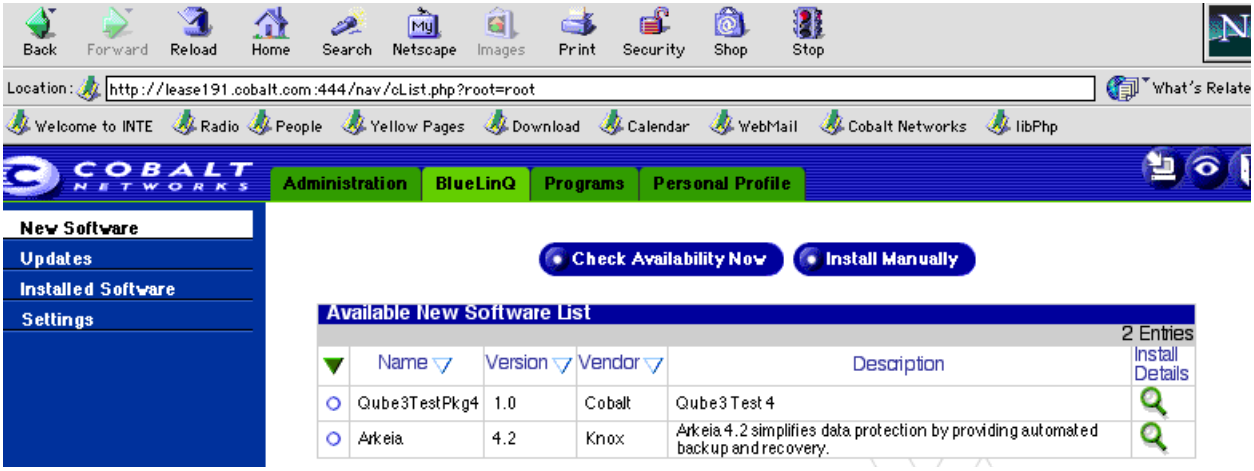
These fields are used to provide information and are included in the actual package as well as provided by the update servers:

- Package identification
  - `Name and nametag`
  - `Version and versionTag`
  - `Vendor and vendorTag`
- Description
  - `shortDesc`
  - `longDesc`
- License information
  - `License`
  - `Splash`
- Category

These fields are found only in update server package:

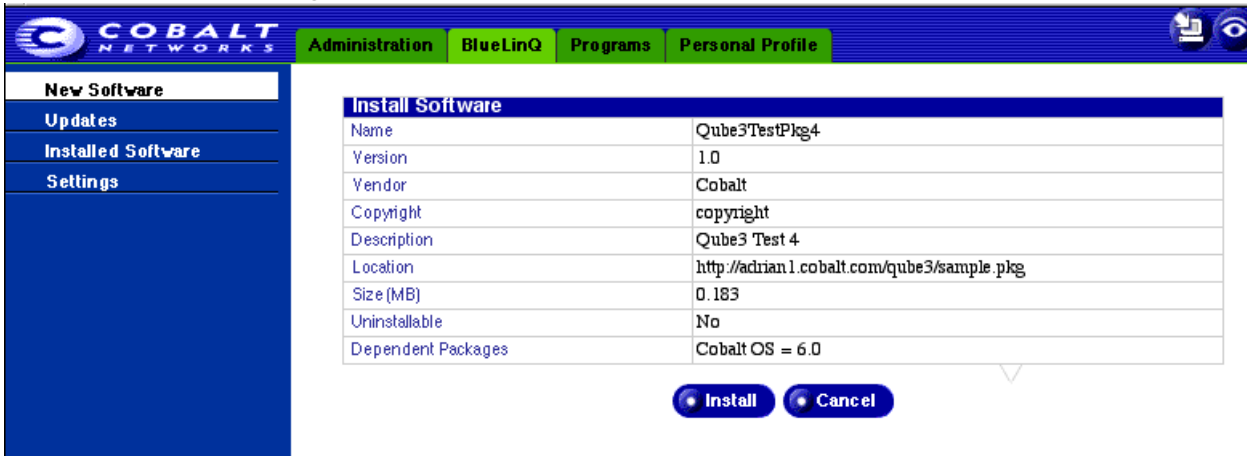
- `Size` (in bytes)
- `PackageType`: `complete` or `update`
- `Location`
- `InfoURL`: a pop-up window appears when the user clicks the magnifying glass

Figure 6—3 New Software Installed



If you click on the magnifying glass, you see the information shown in Figure 6—4, which corresponds to the information in Table 6—3 on page 6—12.

Figure 6—4 New Software Installation Details



Software Update Server

NOTE: If the `infoURL` file exists, it displays a popup window and will not install the actual package.

The **BlueLinQ** tab on the Qube 3 has an **Updates** menu. This page lists available software with the following information.

- ¥ Update server-provided information (name, vendor, locale, description)
- ¥ Pop-up information. `InfoURL` displays the URL to be passed the Qube s serial number
- ¥ The package checks for an `InfoURL`. If one exists, the page referenced by the `InfoURL` appears. If not, the package presents the license information, and installs after the user accepts the license agreement.

When users click on **Install Details**, the Qube 3:

- ¥ Displays the splash page if there is one or displays a license agreement in standardized license format.
- ¥ Begins installation

When the user begins installation, these events occur on the Qube 3:

- ¥ It checks for a signature and attempts to authenticate it, if one is present. If the signature cannot be authenticated, a message is displayed letting the user know that the signature check failed.
- ¥ It runs the preinstallation script.
- ¥ It installs the Redhat Package Modules (RPMs.

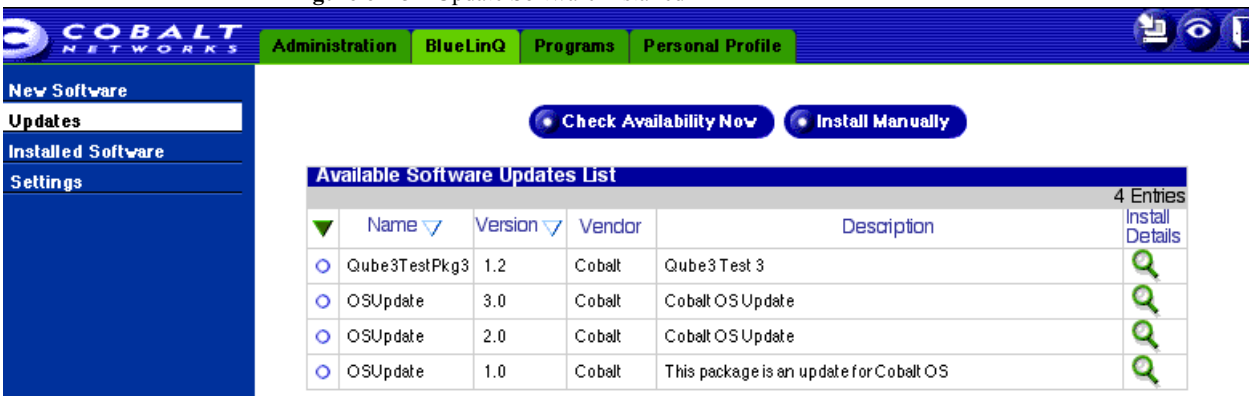
NOTE: Cobalt Networks uses Redhat Package Manager (RPM) files because applications are easy to manage if they are installed using RPM utilities. For details on creating *.rpm files (also known as redhat package module files), see *Maximum RPM*, by Marc Ewing and Erik Troan. *Maximum RPM* is the definitive technical reference for the RPM packaging system; it provides information on RPM's history, usage, and internals from both the user and packager perspectives. Also, see `http://www.redhat.com/` for the most up-to-date information about RPM technology.

- ¥ It runs the postinstallation scripts.

¥ It reboots or refreshes, if those options are set.

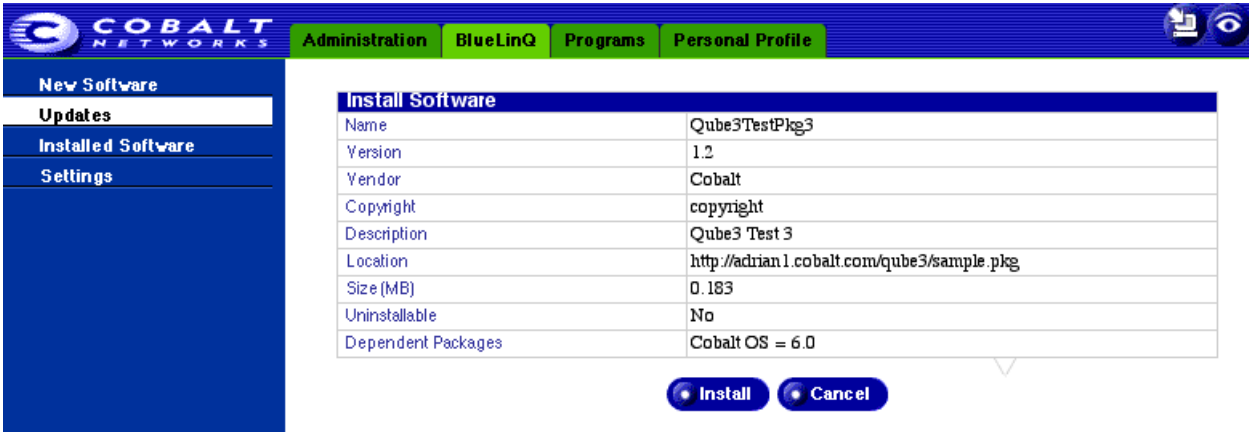
Figure 6—5 shows the Update Server page.

Figure 6—5 Update Software Installed



If you click on the magnifying glass, you see the information shown in Figure 6—6, shown in Figure 6—4, which corresponds to the information in Table 6—3 on page 6—12.

Figure 6—6 Update Software Installation Details

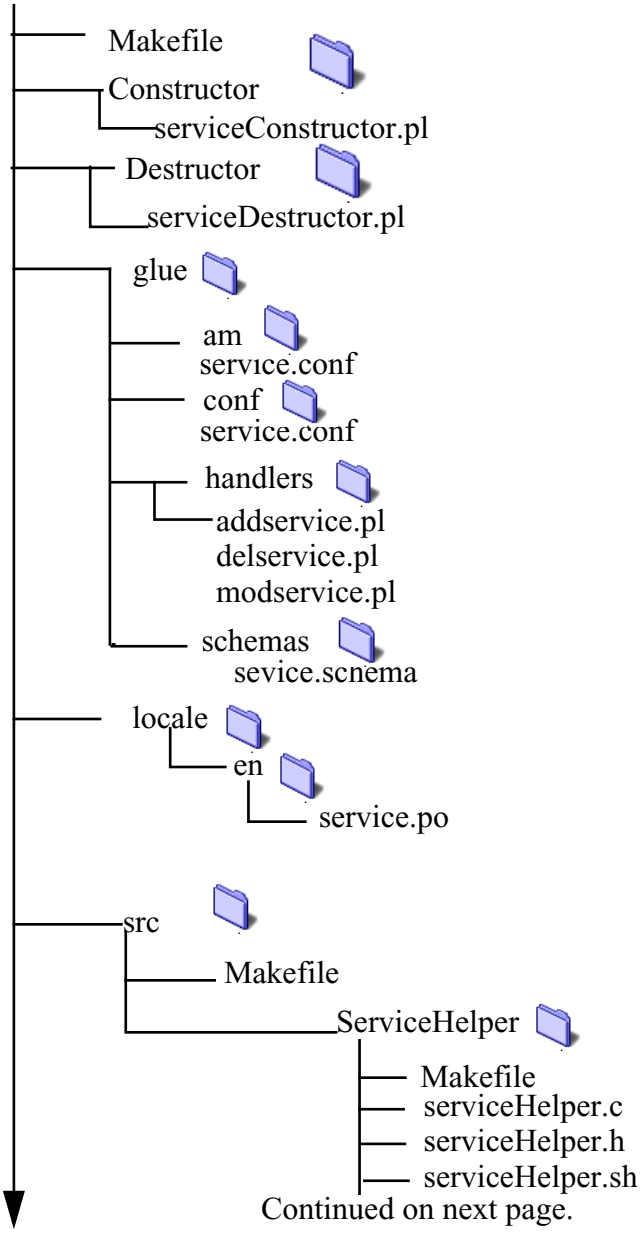


Development Details

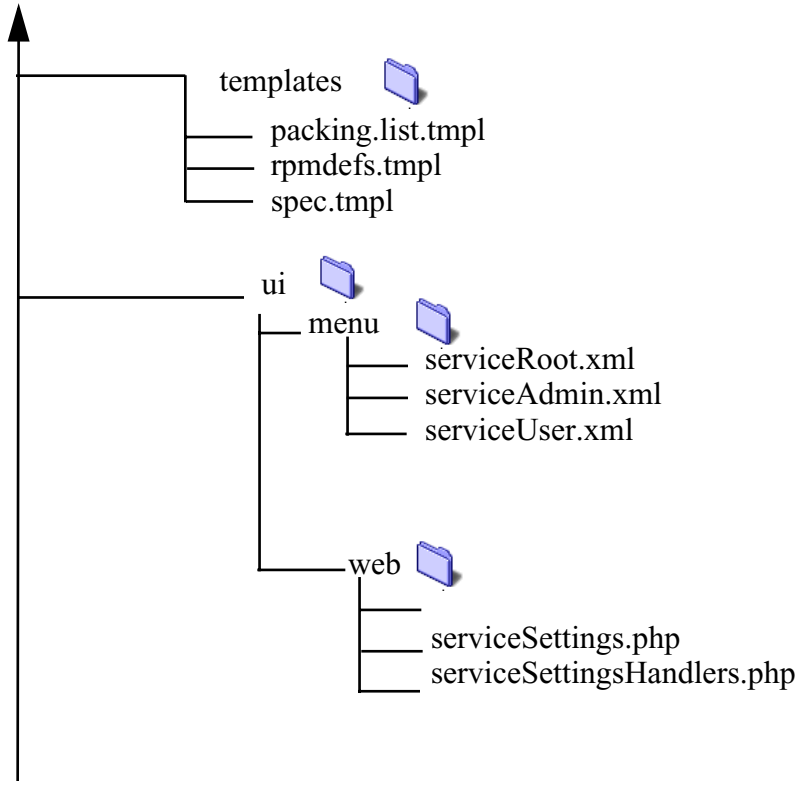
Modules expect the following auxiliary support from Sausalito development tools:

- `SAUSALITO/devel/module.mk` for all the `Makefile` rules.
- `SAUSALITO/bin/mod_rpmize` for the `RPM` specification file generator.

Figure 6—7 Module File Hierarchy



Continued on next page.



Appendix A

User Interface Foundation Classes

This appendix is a complete reference for all User Interface Foundation Classes (UIFC). The UIFC is a comprehensive set of class libraries for Cobalt's user interface components. Their functions include generation of HTML code for rendering and JavaScript code for error checking. “Utility Classes” on page B–1 describes classes that work in conjunction with the UIFC classes.

To use UIFC, you should have some basic knowledge about object-oriented design and programming as well as PHP, because UIFC is object-oriented and implemented in PHP.

The UIFC were designed to provide both user interface consistency and flexibility. `HtmlComponentFactory` is the first class you should look at. It is a factory that constructs UIFC class in the most commonly used way. See “HTMLComponent” on page A–22.

Each UIFC class is listed in this appendix in alphabetical order.

HTML Generation

UIFC contains classes of visual components. The classes have methods to generate the look and feel in HTML. For example, the `IpAddress` class generates HTML code that represents an IP address data type. In this way, a change in look and feel of a visual component within the whole user interface can be accomplished by modifying just one class.

Error Checking

Form fields in UIFC support the plug-in of JavaScript error checking code. This feature is useful for checking and reporting errors interactively. Not all form fields require error checking because their input set might be limited to valid data. For more information of error handling, see “Error” on page B–3.

Reusable Code

The class hierarchy of UIFC is designed to be reusable. It is easy to subclass a UIFC class and make a more specific visual component, for example: a class B IP address, can be made by subclassing the `IpAddress` class. `FormFieldBuilder` also generates HTML code.

Common Pitfalls

There are several things to avoid when using UIFC:

- The UIFC encompass many functions. You must pay special attention in extending UIFC classes to add new functionality. New functionality can introduce inconsistencies if the functionality does not occur in the existing user interface.
- Because UIFC is written in PHP and PHP does not have good support for object-oriented programing, UIFC users can directly refer to private variables and methods of UIFC classes. For good programming practice, do **not** do this because these functions could change in the future.
- Do **not** use UIFC to format free-flow text paragraphs. Pure HTML provides more formatting capabilities than UIFC. You can put HTML inside UIFC pages.
- The `toHeaderHtml()` method of Page object outputs HTTP headers. Do **no**t print anything before this method. As a common PHP catch, blank lines are printed. The following code provides a warning because there is a blank line above the method:

```
<?php
...
?>


<?php
print($page->toHeaderHtml());
?>
```

- Because PHP is interpreted and is basically typeless, it is very easy to pass in parameters of wrong types to functions or methods. This can generate runtime errors from UIFC classes that your code does not use directly.

AddButton

This class creates a labeled `button`. The application causes an `Add` item action when the button is clicked.

Extends:

The class `AddButton` extends `Button`.

Implements:

The `class button` implements `HTMLComponent`, `Stylish`, and `Collatable`.

See also:

`AddButton, BackButton, CancelButon, DetailButton, RemoveButton, SaveButton`

Public Methods

```
function AddButton($page, $action)
```

Description: constructor

Returns:

none

Parameters

`page`: the Page object in which this object resides

`action`: the string used within HREF attribute of the `A` tag

BackButton

This class creates a labeled `Back` button. The application causes a `Back` action when the button is clicked.

Extends:

The class `BackButton` extends `Button`.

Implements:

The `class button` implements `HTMLComponent`, `Stylish`, and `Collatable`.

See also:

`AddButton, BackButton, CancelButon RemoveButton, SaveButton`

Public Methods

Description: constructor

```
function BackButton($page, $action)
```

Parameters

`action`: the string used within HREF attribute of the `A` tag

`page`: the Page object in which this object resides

Bar

This class creates a vertical bar on the `page`.

Extends:

class `Bar` extends `FormField`.

Implements:

`HTMLComponent, Collatable, Stylish`

See also:

`FormField`

Public Methods

`function getLabel()`

Returns: a label in string

function setLabel($label)

Description: set label to replace the percentage shown by default

Parameters

`label:` a label in string

`function setVertical()`

Description: set bar to type vertical

Button

This class creates a labeled `Button`. The application causes a specified action when the button is clicked.

Extends:

`HtmlComponent`

Implements:

The `class button` implements `HTMLComponent`, `Stylish`, and `Collatable`.

See also:

`AddButton, BackButton, CancelButon, DetailButton, RemoveButton, MultiButton, SaveButton`

Public Methods

`function Button($page, $action, $label, $labelDisabled = "")`

Description: Constructor

Parameters

`page`: the Page object in which this object lives

`action:` the string used within HREF attribute of the A tag

`label`: a Label object for the normal state

`labelDisabled`: a Label object for the disabled state. Optional. If it is not supplied, it is the same as the label parameter.

```
function getAction()
```

Description: get the action to perform when the button is pressed

Returns: an action

```
function setAction($action)
```

Description: set the action to perform when the button is pressed

```
function isDisabled()
```

Return: `$isDisabled`

Parameters

`disabled`: true if the button is disabled; false otherwise

```
function setDisabled($isDisabled)
function getLabel()
function getLabelDisabled()
function setLabel($label, $labelDisabled = "")
```

Description: set the label for the button

Parameters

`label`: label object for the normal state

`labelDisabled`: a label object for the disabled state. Optional. If not supplied, it is the same as the label parameter.

```
function toHtml($style = "")
```

Returns: $style, $page

CancelButton

class `CancelButton` extends `Button`.

Description: constructor

This class creates a cancelbutton. The application causes an action to be cancelled when the button is clicked.

Extends:

none

Implements:

The `class button` implements `HTMLComponent`, `stylish`, and `collatable`.

See also:

AddButton, BackButton, CancelButon, DetailButton, RemoveButton, SaveButton

Public Methods

```
function CancelButton($page, $action)
```

Parameters

`page`: the Page object this object lives in

`action`: the string used within HREF attribute of the A tag

CompositeFormField

The class `CompositeFormField` extends `FormField`.

Public Methods

`function CompositeFormField()`

Description: constructor

`function getDelimiter()`

Description: get the delimiter to separate form fields

`function setDelimiter($delimiter)`

Description: set the delimiter to separate form fields

Parameters

`delimiter`: a delimiter in string

`function getFormFields()`

Description: get form fields added to this object

Returns: an array of **FormField** objects

`function addFormField($formField)`

Description: add a form field to this object

Parameter

`formField`: a FormField object

Returns: nothing

CountryName

The class `CountryName` extends `FormField`.

Public Methods

`function CountryName($page, $id, $value)`

Description: constructor

Parameters

`page:` the Page object this form field lives in

`id:` the identifier of this object

`value:` the group of country names defined in ISO 3166

DetailButton

This class creates a `DetailButton`. The application causes a `DetailButton` action when the button is clicked.

Extends:

The class `DetailButton` extends `Button`.

Implements:

The `class button` implements `HTMLComponent`, `Stylish`, and `Collatable`.

See also:

`AddButton, BackButton, CancelButton, DetailButton, ModfiyButton, RemoveButton, SaveButton`

Public Methods

```
function DetailButton($page, $action)
```

Description: constructor

Parameters:

`page`: the Page object this object lives in

`action`: the string used within HREF attribute of the A tag

DomainName

The class `DomainName` extends `FormField`.

DomainNameList

The class `DomainNameList` extends `FormField`.

EmailAddress

The class `EmailAddress` extends `FormField`.

EmailAddressList

The class `EmailAddressList` extends `FormField`.

Public Methods

```
function EmailAddressList ($page, $id, $value, $invalidMessage,
        $emptyMessage)
```

Description: superclass constructor

```
function setImport( $on, $javascriptFunction = "")
```

Description: set the import feature of the list so that email addresses can be imported from the address book

Parameters:

`on`: true to enable import, false to disable

`javascriptFunction`: JavaScript code that is being run during import

```
function setFormat( $format = "BLOCK")
```

Parameters:

Format to have the `EmailAddressList` show up in either `BLOCK` mode in which email addresses are one per line using a `TextBlock` field or in a `SINGLELINE` mode where multiple email addresses can be entered comma-separated in a `First Lastname <abc@abc.net>` format. Using `SINGLELINE` returns the email addresses only in the `$id` variable as done in the `BLOCK` mode, but also returns the unformatted data in the `$id_full` variable.

FileUpLoad

The class `FileUpload` extends `FormField`.

Public methods

```
function FileUpload($page, $id, $value, $maxFileSize = "",
        $invalidMessage, $emptyMessage = "")
```

Description: constructor

Parameters

`page:` the Page object this form field lives in

`id:` the identifier of this object

`value:` the path

`maxFileSize:` the maximum file size allowed to upload in bytes. Optional

`invalidMessage:` the message to be shown upon invalid input. Optional

`emptyMessage:` the message to be shown upon empty input if the field is not optional. This message is optional.

```
function setMaxFileSize($maxFileSize)
```

Description: set the maximum file size allowed to upload

Parameters

`maxFileSize`: bytes in integer

Form

This class represents a HTML `form`.

Applicability

This class is used where a HTML form is needed.

Usage

Each `Page` contains a `Form` object that is accessible by the `getForm()` method of the `Page` object. Form objects have `getId()` methods to get its ID, which is used as the NAME attribute of the HTML FORM tag. Each form has a JavaScript `onsubmit()` handler associated with it. Because JavaScript function `form.submit()` does not call the `onsubmit()` handler; you must explicitly call `onsubmit()` if you submit the form through JavaScript. If no action is supplied, environment variable REQUEST_URI is used as action. Otherwise, JavaScript variable `isActionAvailable` for the Form object is set to true.

Public Methods

```
function Form($page, $action = "")
```

Description: constructor

Parameters

`page`: the Page object this object lives in

`action`: the ACTION attribute of the FORM tag. Optional. If not supplied, it is set to environment variable REQUEST_URI

```
function getAction()
```

Description: get the ACTION attribute

Parameters

`action:` the `action` attribute of the FORM tag

Also see: `setAction()`

`function setAction($action)`

Description: set the `ACTION` attribute

Parameters

`action`: the `ACTION` attribute of the `FORM` tag

Also see: `getAction()`

`function getTarget()`

Description: get the `target` attribute

Returns: the `target` attribute of the `form` tag

Also see: `setTarget()`

`function setTarget($target)`

Description: set the `target` attribute

Returns: the `target` attribute of the `form` tag

Also see: `getTarget()`

`function getId()`

Description: get the ID of the form. It is also the NAME attribute

Returns: a string

Also see: setId()

`function setId($id)`

Description: set the ID of the form. It is also the NAME attribute

Returns: a string

Also see: getId()

`function getSubmitAction()`

Description: get the form action that is used to submit the form

Returns*:* a string

`function $handlerName()`

`function toFooterHtml($style = "")`

Description: translate the footer of the form into HTML representation

Parameters

`style`: a Style object that defines the style of the representation. Optional. If not supplied, the default style is used.

Returns: HTML in string.

FormField

The class `FormField` extends `HtmlComponent`.

NOTE: You can not put HTML into FormField values.

Public Methods

```
function FormField($page, $id, $value = "", $invalidMessage = "",
        $emptyMessage = "")
```

Description: constructor

Parameters

`page`: a Page object in which this form field resides

`id`: the identifier of this form field. Used in the `NAME` attribute of input fields

`value`: the default value of this form field. Depending on what this form field is, the value can be different

`invalidMessage`: message to be shown upon invalid input. Optional

`emptyMessage`: message to be shown upon empty input if the field is not optional. This message is optional

`function getAccess()`

Description: get the access property

Returns: a string

Also see: `setAccess()`

`function setAccess($access)`

Description: set the access property

Parameter

Access can be "" for hidden, "r" for read-only, "w" for write-only and "rw" for both read and write.

Returns: true if succeed, false if failed

Also see: `getAccess()`

`function getCollatableValue()`

`function getEmptyMessage()`

Description: set the message to display when the form field is empty while it should not

Parameter

emptyMessage: a string

Also see: `setEmptyMessage()`

`function setEmptyMessage($emptyMessage)`

Description: set the message to display when the form field is empty while it should not

Parameter

`emptyMessage:` a string

Also see: `setEmptyMessage()`

`function getId()`

Description: set the unique ID of the form field. It is used to identify the form field when the form containing the field is submitted, this ID is a variable name in the submit only alphanumeric characters and underscores are supported

`function setId($id)`

Parameter

`id`: a string

`function getInvalidMessage()`

Description: set the message to display when the form field is invalid

Parameter

`invalidMessage:` a string

`function setInvalidMessage($invalidMessage)`

Description: set the message to display when the form field is invalid

Parameter

`invalidMessage:` a string

Also see: `getInvalidMessage()`

`function isOptional()`

Description: get the optional flag

Returns: true if this form field is optional, false otherwise

Also see: `setOptional()`

`function setOptional($optional)`

Description: set the optional flag; it indicates if the form field can be empty.

Parameters

`optional:` true if the field is optional, false otherwise

Also see: setOptional()

`function getValue()`

Description: get the value

Returns: the value of different types depending on which concrete subclass of form field this is

Also see: `setValue()`

`function setValue($value)`

Description: set the value; depending on the concrete type of the form field (e.g., `IpAddress`); this value can be of different type.

Parameters

`value:` any variable

Also see: `getValue()`

FormFieldBuilder

This class helps to build form field components.

Applicability

Any form field can use this class to build components.

Public methods

```
function makeCheckboxField($id, $value, $access, $checked,
        $onClick = "")
```

Description: make a checkbox field

Parameters

`id`: the identifier of the field

`value`: the value of the HTML input field

`access`: "" for hidden, "r" for read-only, "w" for write-only and "rw" for read and write

`checked`: if it has a value checked, false otherwise

`onClick`: the onClick attribute of the field

Returns: HTML that represents the field

```
function makeFileUploadField($id, $access, $size, $maxLength,
$onChange)
```

Description: make a file upload field

Parameters

`id`: the identifier of the field

`access`: "" for hidden, "r" for read-only, "w" for write-only and "rw" for read and write

`size`: the length of the field

`maxLength`: maximum number of characters that can be entered into the field

`onChange`: the `onChange` attribute of the field

Returns: HTML that represents the field

```
function makeHiddenField($id, $value = "")
```

Description: make a hidden field

Parameters

`id`: the identifier of the field

`value`: the value of the HTML input field

Returns: HTML that represents the field

```
function makeJavaScript($formField, $changeHandler,
$submitHandler)
```

Description: make javascript for form fields

Parameters

`formField`: the form field to generate javascript for

`changeHandler`: the JavaScript function that is called when the form field change

`submitHandler`: the JavaScript function that is called when the form field submits

Returns: HTML that represents the field

```
function makePasswordField($id, $value, $access, $size,
$onChange)
```

Description: make a password field

`id`: the identifier of the field

`access`: "" for hidden, "r" for read-only, "w" for write-only and "rw" for read and write

`size`: the length of the field

`onChange`: the `onChange` attribute of the field

`Returns`: HTML that represents the field

```
function makeRadioField($id, $value, $access, $checked)
```

Description: make a radio field

`id`: the identifier of the field

`value`: the value of the HTML input field

`access`: "" for hidden, "r" for read-only, "w" for write-only and "rw" for read and write parameters checked: true if it is checked, false otherwise

Returns: HTML that represents the field

```
function makeSelectField($id, $access, $size, $width,
$isMultiple, $formId, $onChange = "", $labels = array(),
$values = array(), $selectedIndexes = array())
```

Description: make a select field

Parameters

id: the identifier of the field

`access`: "" for hidden, "r" for read-only, "w" for write-only and "rw" for read and write

`size`: the SIZE attribute of the HTML SELECT tag

`width`: the minimum width. Select field width is static in Netscape, dynamic in IE

`isMultiple`: true if multiple items can be selected, false otherwise

`formId`: the ID of the form this field lives in

`onChange`: the `onChange` attribute of the field. Optional.

`labels`: an array of labels in string. Optional. Must have same length with values

`values`: an array of values in string. Optional. Must have same length with labels

`selectedIndexes`: an array of indexes of labels for the selected

Returns: HTML that represents the field

```
function makeTextField($id, $value, $access, $size, $maxLength,
        $onChange)
```

Description: make a text field

Parameters

`id:` the identifier of the field

`value:` the value of the HTML input field

access: "" for hidden, "r" for read-only, "w" for write-only and "rw" for read and write

`size`: the length of the field

`maxLength`: maximum number of characters that can be entered into the field

`onChange`: the `onChange` attribute of the field

Returns: HTML that represents the field

```
function makeTextAreaField($id, $value, $access, $rows, $columns,
        $onChange, $wrap = "")
```

Description: make a text area field

Parameters

`id`: the identifier of the field

`value`: the value of the HTML input field

`access`: "" for hidden, "r" for read-only, "w" for write-only and "rw" for read and write

`rows`: the number of rows

`columns`: the number of columns

`onChange`: the `onChange` attribute of the field

Returns: HTML that represents the field

`function makeTextListField($id, $values, $access, $formId, $rows, $columns)`

Description: make a text list field

Parameters

`id:` the identifier of the field

`values:` an array of values in string

`access:` "" for hidden, "r" for read-only, "w" for write-only and "rw" for read and write

`formId:` the identifier of the form this field lives in

`rows:` the number of rows

`columns:` the number of columns

Returns: HTML that represents the field

FullName

The class `FullName` extends `FormField`.

GroupName

The class `GroupName` extends `FormField`.

HTMLComponent

The class `HtmlComponent` extends `Stylish`.

Implements

This class implements `Collatable`.

Description: constructor

```
function HtmlComponent($page)
```

Parameters

`page`: the Page object in which this `HTMLComponent` resides.

```
function setPage($page)
```

Description: set Page object in which this `HTMLComponent` resides.

Parameters

`page:` a Page object

```
function toHtml($style = "")
```

Description: translate into a HTML representation

Parameters

`style`: the style of the representation in a Style object

Returns: HTML

ImageButton

class `ImageButton` extends `Button`.

Public methods

```
function ImageButton($page, $action, $image, $lbl, $desc)
```

Description: constructor

Parameters

`page`: the Page object this object lives in

`action`: the string used within HREF attribute of the `A` tag

ImageLabel

The class `ImageLabel` extends `Label`.

Public Methods

```
function ImageLabel($page, $image, $label, $description = "")
```

Description: constructor

Parameters

page: the Page object this object lives in

image: an URL of an image

label: a label string

description: a description string

```
function getImage()
```

Description: get the image used as the label

Returns: an URL of an image function `setImage($image)`

`function setImage($image)`

Description: set the image used as the label

Parameters

`image`: an URL of an image

`function setImage($image)`

IntRange

The class `IntRange` extends `FormField`.

Public Methods

`function IntRange($page, $stylist, $id, $value, $invalidMessage, $emptyMessage = "")`

Description: constructor

Parameters

`page`: the Page object this form field lives in

`stylist`: a Stylist object that defines the style

`id`: the identifier of this object

`value`: the default value

`invalidMessage`: message to be shown upon invalid input. Optional

`emptyMessage`: message to be shown upon empty input if the field is not optional. This message is optional.

`function isConfirm()`

Description: superclass constructor

`function setConfirm($isConfirm)`

Description: set the config flag

Parameter

`isConfirm`: if true, a confirm field is shown

`function setConfirm($isConfirm)`

Description: set the config flag

Parameter

`isConfirm`: if true, a confirm field is shown

Integer

The class `Integer` extends `FormField.`

Public Methods

```
function Integer($page, $stylist, $id, $value, $invalidMessage,
        $emptyMessage = "")
```

Description: constructor

Parameters

`page`: the Page object this form field lives in

`stylist`: a Stylist object that defines the style

`id`: the identifier of this object

`value`: the default value

`invalidMessage`: message to be shown upon invalid input. Optional

`emptyMessage`: message to be shown upon empty input, if the field is not optional. This message is optional.

`function getMax()`

Description: get the maximum valid value

Returns: an integer

Also see: `setMax()`

`function setMax($max)`

Description: set the maximum valid value

Returns: an integer

Also see: `getMax()`

`function getMin()`

Description: get the minimum valid value

Returns: an integer

Also see: `setMain()`

`function setMin($min)`

Description: set the minimum valid value

Returns: an integer

Also see: `getMain()`

`function toHtml($style = "")`

IpAddressList

The class `IpAddressList` extends `FormField`.

Label

The class `Label` extends `HtmlComponent`.

Implements

`Label` implements `Collatable`.

Public Methods

`function Label($page, $label, $description = "")`

Description: constructor

Parameters

`page`: the Page object this object lives in

`label`: a label string

`function getDescription()`

Description: get the description of the label

Returns: a string

Also see: `setDescription()`

`function setDescription()`

Description: get the description of the label

Returns: a string

Also see: `getDescription()`

`function getLabel()`

Description: get the label string of the label

Returns: a string

Also see: `setLabel()`

`function setLabel($label)`

Description: set the label string of the label

Returns: a string

Also see: `getLabel()`

Locale

The class `Locale` extends `FormField`.

Public Methods

`function getPossibleLocales()`

Description: get the list of possible locales

Returns: an array of locale strings

Also see: `setPossibleLocales()`

`function setPossibleLocales($possibleLocales)`

Description: set the list of possible locales

Parameters

`possibleLocales`: an array of locale strings; `browser` is also a possible special locale string case

Also see: getPossibleLocales()

Parameters

`possibleLocales:` an array of locale strings; `browser` is also a possible special locale string case.

MacAddress

The class `MacAddress` extends `FormField`.

MailListName

The class `MailListName` extends `FormField`.

ModifyButton

This class creates a `ModifyButton`. The application causes a `ModifyButton` action when the button is clicked.

Extends:

The class `DetailButton` extends `Button`.

Implements:

The `class button` implements `HTMLComponent`, `Stylish`, and `Collatable`.

See also:

`AddButton, BackButton, CancelButton, DetailButton, MultiButton, RemoveButton, SaveButton`

Public methods

```
function ModifyButton($page, $action)
```

Description: constructor

Parameters

`page`: the Page object this object lives in

`action`: the string used within HREF attribute of the `A` tag

MultiButton

This class represents a button with multiple actions. Users can perform one of those actions by selecting it.

Applicability

Anywhere a related set of actions are provided for the users to select and the selected one is being performed.

Usage

Instantiate a MultiButton by specifying a text. This text is like the label of the button. Use addAction() to add actions to the button. Finally, use toHtml() to get a HTML representation of the button to present.

Extends

The class `MultiButton` extends `FormField`.

Implements

The `class MultiButton` implements `HTMLComponent`, `stylish`, and `collatable`. It also implements `action[]`, `actionText[]`, and `text`.

See also:

`AddButton, BackButton, CancelButton, DetailButton, ModifyButton RemoveButton, SaveButton`

Public Methods

```
function MultiButton($page, $text = "", $id = "")
```

Description: constructor

Parameters

`page`: the Page object this object lives in

`text`: a label text in string. Optional

```
function getActionText($action)
```

Description: superclass constructor

```
function getActions()
```

Description: get all the text of the button

Returns: an array of text strings

Also see: `addAction(), getActions()`

`function addAction($action, $text)`

Description: add an action to the button

Parameters

`action`: the string used within HREF attribute of the A tag

`text`: a label text in string

`function getSelectedIndex()`

Description: get the index of the selected action

Returns: an integer

Also see: `setSelectedIndex()`

`function setSelectedIndex($selectedIndex)`

Description: set the index of the selected action

Returns: an integer

Also see: `getSelectedIndex()`

`function getText()`

Description: get the default text of the button

Returns: a string

Also see: `setText()`

`function setText($text)`

Description: set the default text of the button

Returns: a string

Also see: `getText()`

MultiChoice

This class represents a widget that allows users to choose one or more options. It can render itself as different forms such as pull-down menus or checkboxes for different option types, which can be a single option or many options.

Applicability:

Use `Multichoice` where options need to be selected.

Usage

Instantiate an object and add options, for example, `Option` class, to it. Each option can contain form field objects. For example, a multiple choice for payment method can have cash and credit card options; the credit card option can have a credit card number field associated with it. Although this class selects the best form to render automatically, users can use the set`FullSize()` to force this class to use a more readable but consume more space form. The set`Multiple()` methods can be uses to make multiple options selectable at once. When multiple is set, this submitted value of this form field is an array encoded in a string by array packer.

Extends

The class `MultiChoice` extends `FormField`.

Public Methods

```
function MultiChoice($page, $id)
```

Description: constructor

Parameters

`page`: the Page object this form field lives in

`id`: the identifier of this object

```
function getOptions()
```

Description: get all options added

Returns: an array of Option objects

Also see: `addOption()`

`function addOption($option, $selected)`

Description: add an option; options are not selected by default when they are added.

Parameters

`option`: an Option object

`function setFullSize($fullSize)`

Description: set the full size mode

Parameters

`fullSize`: true to make the object rendered as more readable, but less compact; false otherwise.

Returns: nothing.

`function setMultiple($multiple)`

Description: set the multiple mode

Parameters

`multiple`: true if multiple choices can be selected at the same time; false otherwise

`function setSelected($index, $isSelected = true)`

Description: select a option

Parameters

`index`: an integer index of the option

`isSelected`: true for selected, false otherwise. Optional and true by default.

Returns: nothing

`function setValue($value)`

MultiFileUpload

The class `MultiFileUpload` extends `FormField`.

Public Methods

```
function MultiFileUpload($page, $id, $value, $maxFileSize = false,
        $invalidMessage = "", $emptyMessage = "")
```

Description: constructor

Parameters

`page`: the Page object this form field lives in

`id`: the identifier of this object

`value`: the path

`maxFileSize`: the maximum file size allowed to upload in bytes. Optional

`invalidMessage`: message to be shown upon invalid input. Optional

`emptyMessage`: message to be shown upon empty input if the field is not optional. This message is optional

```
function getMaxFileSize()
```

Description: get the maximum file size allowed to upload

Returns: `maxFileSize`: bytes in an integer

Parameters

`maxFileSize`: bytes in integer

Also see: `setMaxFileSize()`

```
function setMaxFileSize($maxFileSize)
```

Description: set the maximum file size allowed to upload

Returns: `maxFileSize`: bytes in an integer

Parameters

`maxFileSize`: bytes in integer

Also see: `getMaxFileSize()`

NetAddress

The class `NetAddress` extends `FormField`.

NetAddressList

The class `NetAddressList` extends `FormField`.

Option

This class represents an option for the `MultiChoice` class.

Applicability:

Use `option` where `MultiChoice` is used.

Public methods

```
function Option($label, $value, $isSelected = false)
```

Description: constructor

Parameters

`label`: a Label object

`value`: the value of this option

`isSelected`: true if selected, false otherwise. The default is optional and false.

`function getLabel()`

Description: gets the label

Returns: a Label object

Also see: `setLabel()`

`function setLabel($label)`

Description: set the label

Parameter

`label`: a Label object

Also see: `getLabel()`

`function isSelected()`

Description: see if the option is selected

Returns: true if selected, false otherwise

Also see: `setSelected()`

`function setSelected($isSelected)`

Description: select or unselect the option

Parameter

`isSelected`: true to select, false to unselect

Also see: `isSelected()`

`function getValue()`

Description: get the value

Parameters

`value`: a string

Also see: `setValue()`

`function setValue($value)`

Description: set the value

Parameters

`value`: a string

Also see: `getValue()`

`function getFormFields()`

Description: get all the form fields of the block

Returns: an array of FormField objects

`function getFormFieldLabel($formField)`

Description: get the label for a form field

Parameters

`formField`: a FormField object

Returns: a Label object

`function addFormField($formField, $label = "")`

Description: add a form field to this option so this option can associate with another form field

Parameters

`formField:` a FormField object

`label`: a Label object. Optional

Page

This class represents a page on the user interface. It also encapsulates all information about the page. For example, a Stylist object and an I18n object resides in each Page object.

Applicability:

This class is applicable to every page on the user interface that uses UIFC.

Usage

All UIFC pages must have one and only one page object. All `toHtml()` calls of any `HtmlComponent` must reside within the t`oHeaderHtml()` and `toFooterHtml()` calls of the page object. Otherwise, undefined result can happen.

Public Methods

`function Page($stylist, $i18n, $formAction)`

Description: constructor

Parameters

`stylist:` a Stylist object that defines the style

`i18n:` an I18n object for internationalization

`formAction:` the action of the Form object for this Page. Optional

`function getForm()`

Description: get the form embedded in the page

Returns: a Form object

`function getI18n()`

Description: get the `I18n` object used to internationalize this page

Returns: an `I18n` object

Also see: `setI18n()`

`function setOnLoad($js)`

Description: set JavaScript to be performed when the page loads

Parameters

`js`: a string of JavaScript code

`function getStylist()`

Description: get the stylist that stylize the page

Returns: a Stylist object

Also see: `setStylist()`

`function setStylist($stylist)`

Description: set the stylist that stylize the page

Parameters

`stylist`: a Stylist object

Also see: `getStylist()`

`function getSubmitAction()`

Description: get the submit action that submits the form in this page

Returns: a string

`function getSubmitTarget()`

Description: get the target of the embedded form to submit to

Returns: a string

Also see: `setSubmitTarget()`

`function setSubmitTarget($target)`

Description: set the target of the embedded form to submit to

Returns: a string

Also see: `getSubmitTarget()`

`function toHeaderHtml($style = "")`

Description: translate the header of the page into HTML representation

Parameters

style: a Style object that defines the style of the representation. Optional. If not supplied, default style is used

Returns: HTML in string

```
function toFooterHtml($style = "")
```

Description: translate the footer of the page into HTML representation

Parameters

`style`: a Style object that defines the style of the representation. Optional. If not supplied, default style is used

Returns: HTML in string

PagedBlock

`PagedBlock` represents a block that have multiple pages with each of them having their own form fields. The states of form fields on different pages are automatically maintained.

Applicability

Use this class to separate functionally cohesive, but context distant information. For example, use it to group *basic* information into one page and *advanced* information in another. Do **not** use this class simply for navigation purposes, use the navigation system instead.

Usage

To use this class for just one page, create a `PagedBlock` object and add form fields without specifying any page IDs. To support multiple pages, after constructing an object, add pages to it. Afterwards, add form fields to the pages. The page to display can be selected by using `setSelectedId()`, but this is optional. The page to display is maintained automagically based on user interaction. Changed form field values are passed back to the pages as `$formFieldId`. After submission, `$pageId` for visited pages are set to true. Use `getStartMark()` and `getEndMark()` to put HTML code outside the scope of PHP into the context of pages.

The class `PagedBlock` extends `HtmlComponent`. `PagedBlock` implements `page` and `toHTML()`.

Public Methods

`function PagedBlock($page, $id, $label)`

Description: constructor

Parameters

`page`: the Page object this block is in

`id`: an unique ID of the block in string

`label`: a Label object for the block title

`function getButtons()`

Description: get all buttons added to the block

Returns: an array of Button objects

Also see: `addButton()`

`function addButton($button)`

Description: add a button to the list

Parameters

`button:` a Button object

Also see: `getButton()`

`function getEndMark($pageId)`

Description: get the mark for marking the end of a HTML section specifically for a page. This is useful for adding page specific HTML

Parameters

`pageId:` the ID of the page in string

Returns: the mark in string

Also see: `getStartMark()`

`function getFormFields()`

Description: get all the form fields of the block

Parameters

`pageId`: the ID of the page the form field is in. Optional if there is only one page.

Returns: an array of FormField objects

Also see: `addFormField()`

`function addFormField($formField, $label = "", $pageId = "")`

Description: add a form field to this block

Parameters

`formField`: a FormField object

`label:` a label object. Optional. Hidden form fields are not shown and therefore do not need labels

`pageId`: the ID of the page the form field is in; optional if there is only one page

Returns: nothing

Also see: `getFormField()`

`function getDividers()`

Description: get all dividers added to the block

Returns: an array of Label objects

Also see: `addDivider()`

`function addDivider($label = "", $pageId = "")`

Description: add a divider

Parameter

`label`: a label object. Optional.

`pageId`: the ID of the page the form field is in; optional if there is only one page.

`function getFormFieldLabel($formField)`

Description: get the label for a form field

Parameter

`formField`: a FormField object

Returns: a Label object

Also see: `getDividers()`

`function getFormFieldPageId($formField)`

Description: get the page ID of a form field

Parameter

formField: a FormField object

Returns: page ID in string

`function getLabel()`

Description: get the label of the block

Returns: a Label object

Also see: `setLabel()`

`function setLabel($label)`

Description: set the label of the block

Parameter

`label`: a Label object

Also see: `getLabel()`

`function getId()`

Description: get the ID of the block

Parameters

`Id`: a string

Returns: a string

Also see: `setId()`

`function setId($id)`

Description: set the ID of the block

Parameters

`Id`: a string

Returns: a string

Also see: `getId()`

`function getPageIds()`

Description: get all the page IDs

Returns: an array of IDs in string

Also see: `addPage()`

`function getPageLabel($pageId)`

Description: get the label of a page

Parameters

`pageId`: the ID of the page

Returns: a Label object

`function addPage($pageId, $label)`

Description: add a page into the paged block

Parameters

`pageId`: the ID of the page in string

`label`: a Label object for the page

`function getSelectedId()`

Description: get the ID of the selected page

Returns: a string

Also see: `setSelectedId()`

`function setSelectedId($selectedId)`

Description: set the ID of the selected page

Parameters

`selectedId`: a ID string

`function getStartMark($pageId)`

Description: get the mark for marking the start of a HTML section specifically for a page

Parameters

`pageId`: the ID of the page in string

Returns: the mark in string

Password

The class `Password` extends `FormField`.

Public Methods

Description: constructor

```
function Password($page, $stylist, $id, $value, $invalidMessage,
        $emptyMessage = "")
```

Parameters

`page`: the Page object this form field lives in

`stylist`: a Stylist object that defines the style

`id`: the identifier of this object

`value`: the default value

`invalidMessage`: message to be shown upon invalid input. Optional

`emptyMessage`: message to be shown upon empty input if the field is not optional. This message is optional

`function isConfirm()`

Description: see if the confirm field is shown

Return: if true, a confirm field is shown

Also see: `setConfirm()`

`function setConfirm($isConfirm)`

Description: set the configuration flag

Parameter

`isConfirm`: if true, a confirm field is shown

Also see: `isConfirm()`

RemoveButton

The class `RemoveButton` extends `Button`.

Public Methods

`function RemoveButton($page, $action)`

Description: constructor

Parameters

`page`: the Page object this object lives in

`action`: the string used within HREF attribute of the A tag

SaveButton

The class `SaveButton` extends `Button`.

Public Methods

```
function SaveButton($page, $action)
```

Description: constructor

Parameters

`page`: the Page object this object lives in

`action`: the string used within HREF attribute of the A tag

ScrollList

The class `ScrollList` extends `HtmlComponent`. The class represents a list of similar elements to be displayed on pages. This class automatically maintains the number of pages and determine which one to display.

Applicability

User this class when a list of similar elements needs to be represented. Do not use this class for list of different elements.

Usage

This class simply constructs a `ScrollList` object with a list of entry labels specified. You can add entries using the `addEntry()` method.

NOTE: Remember to keep the number of elements of each entry the same as the number of entry labels.

Public Methods

`function ScrollList($page, $id, $label, $entryLabels, $sortables = array())`

Description: constructor

Parameters

`page`: the Page object this object lives in

`id`: the identifier in string

`label`: a label object for the list

`entryLabels`: an array Label object for the entries

`sortables`: an array of indexes of the sortable components. Optional.

`function getAlignments()`

Description: get the horizontal alignments of items in entries

Returns: an array of alignment strings.

Also see: `setAlignments()`.

`function setAlignments($alignments)`

Description: set the horizontal alignments of items in entries

Parameters

`alignments`: an array of alignment strings, for example, "", `left`, `center`, or `right`. "" and empty array element means left. First alignment string for the first item in entries, second alignment string for the second item in entries and so forth

Also see: `getAlignments()`

`function getColumnWidths()`

Description: get the column widths for items in entries

Returns: an array of widths

Also see: `setColumnWidths()`

`function setColumnWidths($columnWidths)`

Description: set the widths of label and form field

Returns: an array of widths in integer (pixel) or string (for example, "60%"). The first element is for label and the second element is for form field.

Also see: `getColumnWidths()`

Parameters

`widths`: an array of widths in numbers (for example, 100), percentage strings (for example, 25%) , " . ", or empty elements, which means no defined width.

Also see: getColumnWidths()

`function addButton($button)`

Description: add a button to the list

Parameters

`button`: a Button object

Also see: `getButtons()`

`function setSelectAll($selectAll = true)`

Description: when `select all` is on and entries can be selected, a widget is available on the list to select or unselect all entries at once.

Parameters

`selectAll`: a boolean

Also see: `isSelectAll(), addEntry()`

`function isSelectAll()`

Description: get the select all flag

Returns: `true` if `select all` is enabled; `false` otherwise.

Also see: `addEntry()`

`function setEmptyMessage($msg = "")`

Description: set the message to be displayed when the list is empty

Parameters

`msg`: an `I18n` tag of the form `[[domain.messageId]]` for interpolation

`function getDuplicateLimit()`

Description: get the upper limit of duplicate buttons at the end of the list

Returns: an integer

Also see: `setDuplicateLimit()`

`function setDuplicateLimit($duplicateLimit)`

Description: the upper limit of duplicate buttons at the end of the list

Parameter

`duplicateLimit`: the limit in integer

Also see: `getDuplicateLimit()`

```
function addEntry($entry, $entryId = "", $entrySelected = false,
$entryIndex = -1)
```

Description: add an entry to the list

Parameters

`entry`: an array of objects that consist the entry

`entryId`: an unique ID for the entry. Optional. If supplied, the entry can be selected

`entrySelected`: true if the entry is selected, false otherwise.Optional.

`entryNumber`: the index of the entry on the list. Optional. If not supplied, the entry is appended to the end of the list

`function getEntryNum()`

Description: get the number of entries in the list

Returns: an integer

Also see: `setEntryNum()`, `addEntry()`

`function setEntryNum($entryNum)`

Description: tell the list how many entries are there in the list. This is useful when you use `addEntry()` only to add a section of the list, so you need to tell the list how many entries are really there

Parameters

`entryNum:` an integer

Also see: getEntryNum(), addEntry()

`function setEntryCountTags($singular, $plural)`

Description: set the `i18n` message tags used in entry count. Message tags have the format of `[[<domain>.<messageId>]]`

Parameters

`singular`: a string message tag used when only one entry is listed

`plural`: a string message tag used when many or zero are listed

`function getEntries()`

Description: get all the entries added to the list

Returns: an array of entries. Each entry is an array of `HtmlComponent` objects

Also see: `addEntry()`

`function getEntryLabels()`

Description: get the labels for each item of the entries

Returns: an array of Label objects

Also see: `setEntryLabels()`

`function setEntryLabels($entryLabels)`

Description: set the labels for each item of the entries

Parameters

`entryLabels`: an array of Label objects

Also see: `getEntryLabels()`

`function getId() {`

Description: get the ID of the block

Returns: an ID string

Also see: `setId()`

`function setId($id)`

Description: set the ID of the block

Parameters

`id`: an ID string

Also see: `getId()`

`function getLabel()`

Description: get the label of the block

Returns: a Label object

Also see: `setLabel()`

`function setLabel($label)`

Description: set the label of the block

Parameters

`label`: a Label object

Also see: getLabel()

`function getLength()`

Description: get the maximum length of pages on the list.

Returns: an integer

Also see: `setLength()`

`function setLength($length)`

Description: set the maximum length of pages on the list. For example, if length is set to 10, and there are 25 entries, the list is presented in 3 pages of 10, 10 and 5 entries.

Parameters

`length`: an integer

Also see: `getLength()`

`function getPageIndex()`

Description: get the index of the page the list is presenting

Returns: an integer

Also see: `setPageIndex(), setLength()`

`function setPageIndex($pageIndex)`

Description: set the index of the page the list is presenting

Parameters

`pageIndex`: an integer

Also see: `getPageIndex()`, `setLength()`

`function isSortEnabled()`

Description: see if sorting is done by the list

`Returns`: a boolean

Also see: `setSortEnabled()`

`function setSortEnabled($sortEnabled)`

Description: enable or disable sorting done by the list. This method is useful if entries supplied are already sorted.

Parameters

`sortEnabled`: a boolean

Also see: getSortEnabled()

`function getSortables()`

Description: get the sortable components of the entries

Returns: an array of indexes of the sortable components

Also see: `setSortables()`

`function setSortables($sortables)`

Description: set the sortable components of the entries

Parameters

`sortables`: an array of indexes of the sortable components

Also see: `getSortables()`

`function getSortedIndex()`

Description: get the index of the components that are sorted

Returns: an integer

Also see: `setSortedIndex()`

`function setSortedIndex($sortedIndex)`

Description: set the index of the components that are sorted. This method always overrides user selection. Use `setDefaultSortedIndex()` if overriding is not desired

Parameters

`sortedIndex`: an integer. If -1, no sorting is done

Also see: `getSortedIndex()`

`function setDefaultSortedIndex($sortedIndex)`

Description: set the index of the components that are sorted. If user has made selections, this method will not override it

Parameters

`sortedIndex`: an integer. If -1, no sorting is done

`function getSortOrder()`

Description: get the order of sorting

Returns: `ascending` or `descending`

Also see: `setSortOrder()`

`function setSortOrder($sortOrder = "ascending")`

Description: set the order of sorting

Parameters

sortOrder: `ascending` or `descending`. Optional and ascending by default

Also see: getSortOrder()

`function sortEntries(&$entries)`

Description: the method to sort the entries when displaying the list

Parameters

`entries`: the array of entries to sort

`function toHtml($style = "")`

Description: turn the object into HTML form

Parameters

`style`: the style to show in (optional)

Returns: HTML that represents the object or "" if `pageIndex` is out of range

SetSelector

The class `SetSelector` extends `FormField`.

Public methods

`function SetSelector($page, $id, $value, $entries, $emptyMessage)`

Description: constructor

Parameters

`page`: the Page object that this object lives in

`id`: the identifier of the object

`value`: an ampersand (&) separated list for the value set

`entries`: an ampersand (&) separated list for the entry set

`emptyMessage`: message to be shown upon empty input

`function getEntriesLabel()`

Description: get the label of the entry set

Returns: a Label object

Also see: `setEntriesLabel()`

`function setEntriesLabel($entriesLabel = "")`

Description: set the label of the entry set

Parameters

`entriesLabel`: a Label object

Also see: `getEntriesLabel()`

`function getValueLabel()`

Description: get the label of the value set

Returns: a Label object

Also see: `setValueLabel()`

`function setValueLabel($valueLabel = "")`

Description: set the label of the value set

Parameters

`valueLabel`: a Label object

Also see: `getValueLabel()`

`function getEntries()`

Description: get the entry set to choose from

Returns: an ampersand-separated list for the entry set

Also see: `setEntries()`

`function setEntries($entries)`

Description: set the entry set to choose from

Parameters

`entries`: an ampersand-separated list for the entry set

SnmpCommunity

The class `SnmpCommunity` extends `FormField`.

Public methods

`function toHtml($style = "")`

StatusSignal

The class `StatusSignal` extends `HtmlComponent`.

Public Methods

`function StatusSignal($page, $status, $url = "")`

Description: constructor

Parameters

`page`: the Page object this object lives in

`status`: `none, normal, problem, severeProblem, new, disabled, noMonitor, replied, old, success, failure, pending`

`url`: the url to which to link (optional)

`function getCollatableValue()`

`function getStatus()`

Description: get the status

Returns: a string

Also see: `setStatus()`

`function setStatus($status)`

Description: set the status

Parameters

status: a string. Possible values are `noMonitor`, `disabled`, `none`, `normal`, `problem`, `severeProblem`, `new`, `replied`, `old`, `success`, `failure`, `pending`

Also see: `getStatus()`

`function setUrl($url)`

Description: set the URL to link to

Parameters

`url`: the url to which to link

`function setDescribed($described)`

Description: describe the signal to users if set to true

Parameters

`described`: true if described, false otherwise

Also see: `isDescribed()`

`function isDescribed()`

Description: see if the signal is described to users

Returns: true if described, false otherwise

Also see: `setDescribed()`

NOTE: For information on the Style class, see the Style appendix.

Stylish

The class `Stylish` gets the default style; subclasses should always override this style.

Public Methods

`function getDefaultStyle($stylist)`

Parameters

`stylist:` a Stylist object

Returns: a Style object

Stylist

The class `Stylist` gets a list of all the style resources that are available.

Public Methods

`function getAllResources($localePreference)`

Description: get a list of all the style resources available

Parameters

`localePreference`: a comma-separated list of preferred locale

Returns: a hash of style resource `id` to name

```
function setResource($styleResource, $locale)
```

Description: set the style resource

Parameters

`styleResource`: an ID in string that identifies the style resource

`locale`: a locale string for style localization

```
function setStyle($style)
```

Description: set a style object to the stylist

```
function getStyle($styleId, $styleVariant = "")
```

Description: get a style object with the specified `id` and `variant`. If no style of the `id` and `variant` can be found, only the `id` is used. If no style of the `id` can be found, an empty style is returned.

Parameters

`styleId`: the identifier of the style in string

`styleVariant`: the variant of the style in string

Returns: a Style object with properties if the style can be found; empty Style object otherwise

```
function _Stylist_getResourceId($file, $localePreference)
```

Description: get the style resource ID from a file

Parameters

`file`: path of the file in string

`localePreference`: a comma-separated list of preferred locale

Returns: a style resource ID in string if succeed or false otherwise

```
function _Stylist_load($styleResource, $locale)
```

Description: loads in a style from `styleDir` defined in the configuration file

Parameters

`styleResource`: an identifier string

```
Style <styleDir>/<styleResource>.xml is loaded
```

`locale`: a locale string for style localization; it returns a hash containing all the style information or empty hash if failed, including key `id` contains the `id` in string, key `variant` contains the variant in string, and key `property` contains properties in a hash

```
function _Stylist_startElementHandler($parser, $name,
$attributes)
```

```
function _Stylist_resourceElementHandler($parser, $name,
$attributes)
```

```
function _Stylist_styleStartHandler($attributes)
```

```
function _Stylist_propertyStartHandler($attributes)
```

```
function _Stylist_parseXmlFile($file, $startElementHandler)
```

TextBlock

The class `TextBlock` extends `FormField`.

Public Methods

```
function TextBlock($page, $id, $value = "", $emptyMessage = "")
```

Description: constructor

Parameters

`page`: the Page object that this object lives in

`id`: the identifier of the object

`value`: a text string. Optional

`emptyMessage`: message to be shown upon empty input if the field is not optional. This message is optional

`function getHeight()`

Description: get the height or number of rows

Returns: an integer

Also see: `setHeight()`

`function setHeight($height)`

Description: set the height or number of rows

Returns: an integer

Also see: `getHeight()`

`function getWidth()`

Description: get the width or number of columns

Returns: an integer

Also see: `getWidth()`

`function setWidth($width)`

Description: set the width or number of columns

Returns: an integer

Also see: `getWidth()`

`function setWrap($val = false)`

Description: set to or not to wrap text

Parameter

`val`: true to wrap, false otherwise

Also see: `isWrap()`

`function isWrap()`

Description: see if text should be wrapped or not

Returns: true to wrap, false otherwise

Also see: `setWrap()`

TextField

The class `TextField` extends `FormField`.

Public Methods

```
function TextField($page, $id, $value, $invalidMessage,
$emptyMessage)
```

Description: constructor

Parameters

`page`: the Page object this form field lives in

`id`: the identifier of this object

`value`: the default value

`invalidMessage`: message to be shown upon invalid input. Optional

`emptyMessage`: message to be shown upon empty input if the field is not optional. This message is optional

```
function setSize($size)
```

Description: set the size or number of columns

Parameters

`size`: an integer

```
function setMaxLength($len)
```

Description: set the maximum length or characters the field can take

Parameters

`len`: an integer

TextList

The class `TextList` extends `FormField`.

TimeStamp

The class `TimeStamp` extends `FormField`.

```
function TimeStamp($page, $id, $value)
```

Description: constructor

Parameters

`page`: the Page object this form field lives in

`id`: the identifier of this object

`value`: the number of seconds since Epoch

```
function getFormat()
```

Description: get the format of the time stamp

Parameters

`format`: can be `date`, `time`, or `datetime`

Also see: `getformat()`

```
function setFormat($format)
```

Description: set the format of the time stamp

Parameters

`format`: can be `date`, `time`, or `datetime`

Also see: `getformat()`

TimeZone

The class `TimeZone` extends `FormField`.

Public Methods

```
function TimeZone($page, $id, $value)
```

Description: constructor

Parameters

`page`: the Page object this form field lives in

`id`: the identifier of this object

UninstallButton

The class `UninstallButton` extends `Button`.

Public Methods

```
function UninstallButton($page, $action)
```

Description: constructor

Parameters

`page`: the Page object this object lives in

`action`: the string used within HREF attribute of the A tag

Url

The class `Url` extends `FormField`.

Public Methods

`function Url($page, $id, $value, $label = "", $target = "", $invalidMessage = "", $emptyMessage = "")`

Description: constructor

Parameter

`page`: the Page object this form field lives in

`id`: the identifier of this object

`value`: the URL

`label`: a label in string. Optional

`target`: the `TARGET` attribute of the A tag. Optional

`invalidMessage`: message to be shown upon invalid input. Optional

`emptyMessage`: message to be shown upon empty input if the field is not optional. This message is optional.

`function getCollatableValue()`

`function getLabel()`

Description: get the label

Returns: a label in string

Also see: setLabel()

`function setLabel($label)`

Description: set the label

Parameters

`label`: a label in string

Also see: `getLabel()`

`function getTarget()`

Description: get the target

Returns: the `TARGET` attribute of the A tag

Also see: `setTarget()`

```
function setTarget($target)
```

Description: set the target

Parameters

`target`: the `TARGET` attribute of the A tag

Also see: `getTarget()`

UrlList

The class `UrlList` extends `FormField`.

Public Methods

```
function UrlList($page, $id, $value, $labels = array(), $targets =
        array(), $invalidMessage, $emptyMessage)
```

Description: constructor

Parameters

`page`: the Page object this form field lives in

`id`: the identifier of this object

`value`: an URL encoded list of URLs

`labels`: an array of label strings. Optional

`targets`: an array of target attributes for the A tag in strings. Optional

`invalidMessage`: message to be shown upon invalid input. Optional

`emptyMessage`: message to be shown upon empty input if the field is not optional. This message is optional

`function getLabels()`

Description: get the labels

Returns: an array of label strings

Also see: `setLabels()`

`function setLabels($labels)`

Description: set the labels

Parameters

`labels`: an array of label strings

Also see: `getLabels()`

`function getTargets()`

Description: set the targets attributes

Parameters

`labels:` an array of label strings

Also see: `getTargets()`

`function setTargets($targets)`

Description: set the labels

Parameters

`labels`: an array of label strings

UserName

The class UserName extends `FormField`.

UserNameList

The class `UserNameList` extends `FormField`.

VerticalCompositeFormField

The class `VerticalCompositeFormField` extends `CompositeFormField`.

Appendix B

Utility Classes

This appendix describes three utility classes:

- `ArrayPacker`
- `Error`
- `ServerScriptHelper`

These classes work in conjunction with the UIFC classes to help you create User Interface pages.

ArrayPacker

`ArrayPacker` provides a library of functions for packing and unpacking arrays or hashes to or from strings. The functions use CCE preferred array packing format, which is URL-encoded elements delimited by ampersands (&). For example, an array of `first`, `seco&d`, and `_third` is packed into `&first&seco%26d&_third&`.

Applicability

This class can be used anywhere where arrays or hashes need to be got from or put into CCE.

```
function arrayToString($array)
```

Description: converts an array to a string

Paramater

`array`: an array of strings

Returns: the packed array in string

`function stringToArray($string)`

Description: convert a string to an array

Parameter

`string`: a packed array in string

Returns: an array of strings

`function isInArrayString($needle, $hayStack)`

Description: to see if a string is in an array

Parameters

`needle`: the string to find

`hayStack`: a packed array in string

Returns: true if string found, false otherwise

`function hashToString($array)`

Description: convert a hash (associative array) to a string, for example, `["foo"] = "bar", [1] = "one" => "&foo=bar&1=one&"`

Parameters

`array`: a hash

Returns: a packed hash in string

`function stringToHash($string)`

Description: convert a string to a hash (associative array), for example, `"&foo=bar&1=one&" => ["foo"] = "bar", [1] = "one"`

Parameters

`string`: a packed hash in string

Returns: a hash

Error

This class represents an error.

`function Error($message, $vars = array())`

This method sets the error message with the `message` and `vars` parameters.

Description: constructor

Parameters

`message`: an internationalizable string, that is, it can have `[[domain.id]]` tags.

`vars`: a hash of variable names to values for localizing the string

`function getMessage()`

Description: get the error message

Returns: an internationalizable string

Parameters

`message`: an internationalizable string, that is, it can have `[[domain.id]]` tags.

`vars`: a hash of variable names to values for localizing the string

See also: `setMessage()`

Optional Methods

`function setMessage($message, $vars = array())`

Returns: none

`function getVars()`

Description: get the hash for string localization

Returns: `vars`: a hash of variable names to values for localizing the message string. Optional

See also: `setMessage()`

```
function setVar($key, $val)
```

Description: adding a variable to the string localization hash

Parameters

`key`: the key of the variable in string

`val`: the value of the variable in string

See also: getVars()

ServerScriptHelper

This class is designed to facilitate the development of server-side scripts. It is a library of commonly used functions.

Applicability

This class is applicable to server-side scripts that use session, UIFC, `I18n`, and CCE.

Usage

This class construct a new `ServerScriptHelper` at the start of every server-side script. It automatically gets session information, identifies the logged-in user, and connects to CCE to find out more information about the user. The `get` method can be used to get information about the script.

NOTE: Always call `destructor()` at the end of the scripts.

Public Methods

```
function ServerScriptHelper($sessionId = "", $loginName = "")
```

Description: constructor

Parameters

`sessionId`: the session `id` in string. Optional. If not supplied, the global `$sessionId` is used

`loginName`: the login name of the user in string. Optional. If not supplied, the global `$loginName` is used

`function redirect()`

`function destructor()`

Description: destructor

`function getFile($filename)`

Description: Returns the contents of a file using the Unix permissions granted to the current CCE user.

Parameters

`filename`: The filename of the file to be opened

Returns: the contents of the file

`function popen($cmd)`

Description: opens a read-only stream wrapped by CCE

Parameters

`program`: A string containing the program to execute, including the path and any arguments

Returns: a file handle to be read from

`function shell($cmd, &$output)`

Description: allows one to execute a program as the currently logged in user

Parameters

program: A string containing program to execute, including path and any arguments output variable that picks up the output sent by the program

Returns: 0 an success, error number on error

`function fork($cmd)`

Description: allows one to fork a program as the currently logged in user.

NOTE: No interaction between the called program and the caller can be made.

Parameters

program: A string containing program to execute, including path and any arguments

Returns: 0 an success, error number on error

`function getAccessRights()`

Descriptions: get an array of access rights

Returns: an array of access rights in strings

`function getCceClient()`

Description: get a connected and authenticated CceClient

Returns: a `CceClient` object

`function getHtmlComponentFactory($i18nDomain, $formAction = "")`

Description: get a `HtmlComponentFactory` object to construct `HtmlComponents`

Parameters

`i18nDomain`: the `I18n` domain used for construction

`formAction`: the action of the form in which `HtmlComponents` reside

Returns: a `HtmlComponentFactory` object

`function toErrorJavascript($errors)`

Description: represent errors in JavaScript

Parameters

`errors`: an array of error objects

Returns: JavaScript if error occurred or "" otherwise

`function getI18n($domain = "", $httpAcceptLanguage = "")`

Description: get the right `I18n` object

Parameters

`domain`: the domain of the I18n object. Optional

`httpAcceptLanguage`: the HTTP_ACCEPT_LANGUAGE header. Optional. If not supplied, global `$HTTP_ACCEPT_LANGUAGE` is used

Returns: an `I18n` object

`function getLocalePreference($httpAcceptLanguage = "")`

Description: gets the preferred locale specified by the logged -in user if `browser` is preferred, locale from `HTTP_ACCEPT_LANGUAGE` is used. If no locale is preferred, use the `defaultLocale` specified in `ui.cfg`.

Parameters

`httpAcceptLanguage`: the `HTTP_ACCEPT_LANGUAGE` header. Optional. Global `HTTP_ACCEPT_LANGUAGE` is used if a value is not supplied.

Returns: a list of locales in string separated by commas.

`function getLoginName()`

Description: get the name of the logged-in user

Returns: login name in string

`function getStylePreference()`

Description: gets the style preferred by the logged in user; if user has no preference or if the preference is not available, use any style available on the system

Returns: style ID in string

`function getStylist()`

Description: get the `Stylist` who gives right styles according to the style preference of the logged-in user

Returns: a `Stylist` object

`function toHandlerHtml($returnUrl = "", $errors = array())`

Description: gets the HTML page to be printed out by UI page handlers

Parameters

`returnUrl`: the URL the handler returns to. Optional

`errors`: an array of Error objects for errors occurred within the handler. Optional

`function getCListStyleJavascript()`

Description: get JavaScript to set style for collapsible list

Returns: JavaScript in string

`function getFlowControlStyleJavascript()`

Description: get JavaScript to set style for flow navigation

Returns: JavaScript in string

`function getInfoStyleJavascript()`

Description: get JavaScript to set style for info

Returns: JavaScript in string

`getTabStyleJavascript()`

Description: get JavaScript to set style for tab

Returns: JavaScript in string

`function getTitleStyleJavascript()`

Description: get JavaScript to set style for title

Returns: JavaScript in string

Appendix C I

About Style

This Appendix provides a comprehensive description of the `Style` file. See How Styles Work on page 3—9 for an overview.

Style Files

Style files are XML files located in `/usr/sausalito/ui/style/`. Each of these files contains all the information about a certain style resource. These XML files can contain `styleResource`, `style`, and `property` elements.

An example of a style file is `goodlooking.xml`:

```
<styleResource name="Good Looking">
  <style id="Block">
    <property name="backgroundColor" value="#FFFFFF"/>
  </style>

  <style id="Label">
    <property name="color" value="#FFFFFF"/>
  </style>
</styleResource>
```

Style files must be enclosed by a `styleResource` element. This element can have these attributes:

```
name ::= internationalizable string
```

`name` is the name of the style resource. The interpolate function of `I18n` module is used to internationalize this string.

Within `styleResource`, there are `style` elements. Each `style` element describes one style. The attributes of this element are:

```
id ::= [a-zA-Z0-9_\-]+
```

This is the identifier of the style.

```
variant ::= [a-zA-Z0-9_\-]+
```

NOTE: The `variant` attribute is optional. It acts as a secondary identifier of the style. Each style in the same style file must have a unique `id` and `variant`.

Within `style` elements, there are `property` elements. Each of these elements describes a property of the style. The attributes are:

```
name ::= [a-zA-Z0-9_\-]+
```

Each property is identified by a name.

```
target ::= [a-zA-Z0-9_\-]+
```

The `target` attribute is optional. It acts as a secondary identifier and specifies the target to which property applies. Properties within a style element must not have the same names and targets.

```
value ::= string
```

where *string* is the value of the property.

Supported Styles

Property Types

Different properties have different value types. These are commonly used types for the properties:

Boolean

```
String "true" or "false".
```

Color

```
RGB format (for example, #RRGGBB) or names (for example, green).
```

Positive integer

```
Positive integers including 0.
```

URL

```
A URL.
```

Common Properties

Common properties are properties used commonly in many different styles.

backgroundColor

Description: the background color of the page.

NOTE: Do not use with property `backgroundImage`.

Value type

```
Color
```

backgroundImage

Description: the background image of the page.

NOTE: Do not be use with property `backgroundColor`.

Value type

```
URL
```

borderThickness

Description: the pixel thickness of border.

Value type

```
Positive integer
```

color

Description: the color of text.

Value type

```
Color
```

fontFamily

Description: the family of the font that is used.

Value type

The value type is the same as CSS-1 font-family definition. Generic families are cursive,fantasy, monospace, sans-serif and serif.

fontSize

Description: the size of the font.

Value type

The value type is the same as CSS-1 font-size definition, for example: 12 point, large or 120%.

fontStyle

Description: the style of the font.

Value type

The value type is the same as CSS-1 font-style definition, for example: normal or italic.

fontWeight

Description: the fontWeight is the weight (boldness) of the font.

Value type

The value type is the same as CSS-1 font-weight definition, for example, bold or 900.

textDecoration

Description: Decoration of text.

Value type

The value type is the same as CSS-1 text-decoration definition. For example, blink, line-through, none or underline.

width

Description: Pixel width.

Value type

Positive integer

Styles

Bar

Description: In UIFC, the `Bar` class that represents a bar chart.

Common properties

`color`

`fontFamily`

`fontSize`

`fontStyle`

`fontWeight`

`textDecoration`

Unique properties

none

emptyImage

Description: Image for the empty portion of the bar.

Value type

`URL`

Possible targets

`none`

endImage

Description: Image for the end portion of the bar.

Value type

`URL`

Possible targets

`none`

filledImage

Description: Image for the filled portion of the bar.

Value type

`URL`

Possible targets

`none`

startImage

Description: Image for the start portion of the bar.

Value type

`URL`

Possible targets

`none`

Button

Description: `Button` is a class in the UIFC; see Button on page A—5. The `Button` class represents a clickable button.

Common properties

`backgroundColor`

`backgroundImage`

`color`

`fontFamily`

`fontSize`

`fontStyle`

`fontWeight`

`textDecoration`

CancelButton

Description: `CancelButton` class that represent a cancel button; see also CancelButton on page A—7.

Common properties

`backgroundColor`

`backgroundImage`

`color`

`fontFamily`

`fontSize`

`fontStyle`

`fontWeight`

`textDecoration`

Label

Description: `Label` class represent a text label with description; see also Label on page A—26.

Common properties

`backgroundColor`

`backgroundImage`

`color`

```
fontFamily
fontSize
fontStyle
fontWeight
textDecoration
```

ModifyButton

Description: `ModifyButton` class that represent a button for the modify action.; see also ModifyButton on page A—29.

Unique properties

```
modifyIcon
  Icon for the button.
```

Value type

```
URL
```

Possible targets

```
none
```

MultiChoice

Description: `MultiChoice` class that represent a widget for selecting choices. It has the following options:

- `Label` represents labels of choice.
- `formFieldLabel` represents labels of `formfields` if the choice has them.
- `subscript` represents subscripts used in `MultiChoice` class, such as `optional`.

See also MultiChoice on page A—32.

Common properties

`color, fontFamily, fontSize, fontStyle, fontWeight and textDecoration`

Possible targets:

`choiceLabel, formFieldLabel, subscript`

Page

Description: `page` class that represents a user interface page. See also Page on page A—37.

Common properties

`backgroundColor`

`backgroundImage`

`color`

`fontFamily`

`fontSize`

`fontStyle`

`fontWeight`

`textDecoration`

Unique properties

`center`

`center` defines if all the content of the page should be centered.

Value type

`Boolean`

Possible targets

`none`

PagedBlock

Description:`PagedBlock` class that represents blocks that group form fields together.; see also PagedBlock on page A—40. It has the following attributes:

- `dividerCell` represents the cells that act as dividers.
- `dividerLabel` represents labels in divider cells.
- `form`
- `FieldCell` represents cells in which form fields reside.
- `labelCell` represents cells in which form field labels reside.
- `labelLabel` represents labels in the form field label cells.
- subscript represents possible subscripts used in `PagedBlock` class. such as `optional.`
- `tabSelected` represents the selected tab.
- `tab`
- `Unselected` represents tabs that are not selected.
- `titleCell` represents the cell in which `titleLabe`l resides.
- `titleLabel` represents the label for the title.

Common properties

`backgroundColor and backgroundImage (Possible targets: dividerCell, formFieldCell, labelCell, tabSelected, tabUnselected, titleCell) borderThickness, color, fontFamily, fontSize, fontStyle, fontWeight and textDecoration`

Possible targets:

`dividerLabel, labelLabel, subscript, tabSelected,`

`tabUnselected, titleLabel) width`

Unique properties

`borderColor`

`The color of the block border.`

Value type

`Color`

Possible targets

`none`

dividerHeight

Description: the pixel height of block dividers. If there is content within the divider and it is taller than this value, the divider is expanded to be greater than this value to fit the content.

Value type

`Positive integer`

Possible targets

`none`

icon

Description: the icon image to indicate if the tab is selected or not.

Value type

`URL`

Possible targets

`tabSelected, tabUnselected`

Password

Description: `Password` class that represents a password. It has `subscript. subscript` represents subscripts used in the `Password` class such as repeat.

Common properties

`color, fontFamily, fontSize, fontStyle, fontWeight and textDecoration`

Possible targets

subscript

RemoveButton

Description: `RemoveButton` class that represent a button for the remove action; see also RemoveButton on page A—46.

Unique properties

none

removeIcon

Description: Icon for the button.

Value type

`URL`

Possible targets

`none`

SaveButton

Description: `SaveButton` class that represent a button for the save action; see SaveButton on page A—47.

Common properties

`backgroundColor`

`backgroundImage`

color

fontFamily

fontSize

fontStyle

fontWeight

textDecoration

SetSelector

Description: SetSelector class that represent a widget to select a subset out of a full set.; see SetSelector on page A—55.

Unique properties!

addIcon

The icon for the add button to add entries to the set.

Value type

URL

Possible targets

none

addIconGray

Description: the icon for the add button to add entries to the set in grayed out state.

Value type

URL

Possible targets

none

removeIcon

Description: the icon for the remove button to add entries to the set.

Value type

`URL`

Possible targets

`none`

removeIconGray

Description: the icon for the remove button to add entries to the set in grayed out state.

Value type

`URL`

Possible targets

`none`

ScrollList

Description: `ScrollList` class that represents a scrollable list; see ScrollList on page A—47. It has:

`entryCell`

`labelCell`

`labelLabel`

`titleCell`

`titleLabel`

`entryCell` represents cells in which entries reside. `labelCell` represents cells in which labels reside. `labelLabel` represents labels in label cells. `titleCell` represents the cell in which the title reside. `titleLabel` represents the title label.

Common properties

`backgroundColor and backgroundImage`

Possible targets:

`entryCell, labelCell, titleCell)`

borderThickness

color, fontFamily, fontSize, fontStyle, fontWeight and textDecoration

Possible targets:

entryCell, labelLabel, titleLabel

Unique properties

none

borderColor

The color of the scroll list border.

Value type

URL

Possible targets

`none`

sortAscendingIcon

The icon for the button to sort entries in ascending order. Used in unsorted columns.

Value type

`URL`

Possible targets

none

sortDescendingIcon

The icon for the button to sort entries in descending order used in unsorted columns.

Value type

URL

Possible targets

none

sortedAscendingIcon

The icon for the button to sort entries in ascending order, used in the sorted column.

Value type

URL

Possible targets

none

sortedDescendingIcon

The icon for the button to sort entries in descending order, used in the sorted column.

Value type

URL

Possible targets

none

StatusSignal

Description: `StatusSignal` class that represents a status signal. See also StatusSignal on page A—57.

Unique properties

none

failureIcon

The icon the indicate a failure state.

Value type

`URL`

Possible targets

`none`

newIcon

`The icon the indicate a new state.`

Value type

`URL`

Possible targets

`none`

noneIcon

The icon the indicate a none state.

Value type

URL

Possible targets

none

normalIcon

The icon the indicate a normal state.

Value type

URL

Possible targets

none

oldIcon

The icon the indicate an old state.

Value type

URL

Possible targets

none

problemIcon

The icon the indicate a problem state.

Value type

URL

Possible targets

none

repliedIcon

The icon the indicate a replied state.

Value type

URL

Possible targets

none

severeProblemIcon

The icon the indicate a severe problem state.

Value type

URL

Possible targets

none

successIcon

The icon the indicate a success state.

Value type

URL

Possible targets

none

cListNavigation

Description: this is used for the collapsible list navigation system.

Unique properties

`collapsibleListWidth`

`The pixel width of the collapsible list widget.`

Value type

`Positive Integer`

Possible targets

`none`

infoHeight

`The pixel height of the information widget.`

Value type

`Positive Integer`

Possible targets

`none`

tabHeight

`The pixel height of the tab widget.`

Value type

`Positive Integer`

Possible targets

`none`

collapsibleList

Description: used for the collapsible list widget in the collapsible list navigation system.

Common properties

`backgroundColor and backgroundImage`

Possible targets:

`list, page`

borderThickness

`color, fontFamily, fontSize, fontStyle, fontWeight and textDecoration`

Possible targets:

`selected, unselected width`

Unique properties

`none`

collapsed Icon

The icon to indicate an item with children is collapsed.

Value type

`URL`

Possible targets

`none`

expandedIcon

The icon to indicate an item with children is expanded.

Value type

`URL`

Possible targets

`none`

selectedIcon

The icon to indicate an item without children is selected.

Value type

`URL`

Possible targets

`none`

unselectedIcon

Description: the icon to indicate an item without children is unselected.

Value type

`URL`

Possible targets

`none`

info

Description: the information widget used in several navigation systems.

Common properties

`backgroundColor and backgroundImage`

Possible targets

`error`

`help`

`color`

`fontFamily`

`fontSize`

`fontStyle`

`fontWeight`

`textDecoration`

Possible targets:

`error, help`

Unique properties

none

downIcon

Description: the icon for the button for going down.

Value type

`URL`

Possible targets

`error, help`

downIconGray

Description: the icon for the button for going down in grayed out state.

Value type

`URL`

Possible targets

`error, help`

typeIcon

Description: the icon to indicate the type of the information.

Value type

`URL`

Possible targets

`error, help`

upIcon

Description: the icon for the button for going up.

Value type

URL

Possible targets

error, help

upIconGray

Description: the icon for the button for going up in grayed out state.

Value type

URL

Possible targets

error, help

tab

Description: the tab widget is in the collapsible list navigation system.

Common properties

backgroundColor

backgroundImage

color

fontFamily

fontSize

fontStyle

fontWeight

textDecoration

Possible targets:

selected, unselected)

Unique properties

logo

Description: the logo to be shown next to the tabs.

Value type

URL

Possible targets

none

selectedImageLeft

Description: the image put on the left of the selected tab item.

Value type

URL

Possible targets

none

selectedImageRight

Description: the image put on the right of the selected tab item.

Value type

URL

Possible targets

none

unselectedImageLeft

Description: the image put on the left of unselected tab items.

Value type

URL

Possible targets

none

unselectedImageRight

Description: the image put on the right of unselected tab items.

Value type

`URL`

Possible targets

`none`

Appendix D

Base Data Types

This section contains base data type definitions for the Sausalito architecture.

CAUTION! Do not reuse or redefine the base types listed above. If you modify the definitions of the base types, it can cause a data collision where it might not be clear which data type definition is used. If you need to extend the data type definitions, append your vendor name to them, for example, `vendor_use.emailaddress`.

Scalar

`Scalar` is any data.

```
<typedef name="scalar"          type="re"       data="^.*$"/>
```

Word

`Word` is any non-whitespace data.

```
<typedef name="word"            type="re"       data="^[^
\t\n\r\v\f]+$"/>
```

Alphanum

`Alphanum` is any alphanumeric data.

```
<typedef name="alphanum"        type="re"       data="^[A-Za-z0-
9]+$"/>
```

Alphanum_plus

An `alphanum_plus` is alphanumeric data plus an approved subset of punctuation.

```
<typedef
        name="alphanum_plus"
        type="re"
        data="^[A-Za-z0-9._-]+$"
/>
```

Int

A `Int` is a signed integer.

```
<typedef name="int"             type="re"
data="^(\-?[1-9][0-9]*)|(0)$"/>
```

Uint

A `Uint` is an unsigned integer.

```
<typedef name="uint"            type="re"
data="^([1-9][0-9]*)|(0)$"/>
```

Boolean

A `Boolean` is empty or `0` for `FALSE`; any data for `TRUE`.

```
<typedef name="boolean"         type="re"       data=".*"/>
```

Ipaddr

```
<typedef name="ipaddr"          type="re"
```

```
data="^(([0-9])|([1-9][0-9])|(1[0-9][0-9])|2[0-4][0-9]|25[0-5])\
.(([0-9])|([1-9][0-9])|(1[0-9][0-9])|2[0-4][0-9]|25[0-5])\.(([0-
9])|([1-9][0-9])|(1[0-9][0-9])|2[0-4][0-9]|25[0-5])\.(([0-9])|([1-
9][0-9])|(1[0-9][0-9])|2[0-4][0-9]|25[0-5])$"
/>
```

Network

`A network` defines a network number, such as `10.9.0.0/16`.

```
<typedef name="network"                  type="re"


data="^(([0-9])|([1-9][0-9])|(1[0-9][0-9])|2[0-4][0-9]|25[0-5])\
.(([0-9])|([1-9][0-9])|(1[0-9][0-9])|2[0-4][0-9]|25[0-5])\.(([0-
9])|([1-9][0-9])|(1[0-9][0-9])|2[0-4][0-9]|25[0-5])\.(([0-9])|([1-
9][0-9])|(1[0-9][0-9])|2[0-4][0-9]|25[0-5])/([1-9]|[12][0-9]|3[0-
2])$"
/>
```

Email Address

`The email_address` is the address of the email user, for example, fred@cobalt.com.

```
<typedef
  name="email_address"
  type="re"
  data="^[a-zA-Z\-\_\d\.]+\@[a-zA-Z\-\_\d\.]+$"
/>
```

Netmask

`A netmask` can be either a number from 1 to 32 or a dot-quaded IP mask.

```
<typedef name="netmask" type="re"
```

```
data="^(([1-9])|([12][0-9])|(3[0-
2])|((0|128|192|224|240|248|252|254|255)\.0\.0\.0)|255\
.((0|128|192|224|240|248|252|254|255)\.0\.0)|255\.255\
.((0|128|192|224|240|248|252|254|255)\.0)|255\.255\.255\
.((0|128|192|224|240|248|252|254|255)))$"
/>
```

Fqdn

An `fqdn` is the fully qualified domain name, for example, `www.cobalt.com`

```
<typedef name="fqdn" type="re"
  data="^([A-Za-z0-9][A-Za-z0-9\-]*\.)+[A-Za-z]{2,3}$"
/>
```

Hostname

`A hostname` is defined as follows:

```
<typedef name="hostname"        type="re"
  data="^[A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*$"
/>
```

Domainname

A domainname is defined as follows:

```
<typedef name="domainname"      type="re"
  data="^(localdomain)|(([A-Za-z0-9][A-Za-z0-9\-]*\.)+[A-Za-
z]{2,3})$"
/>


<typedef
        name="password"
        type="re"
```

```
                data="^[^\001-\037\177]{3,16}$"
        />
```

Appendix E

Cobalt System Configuration Protocol

Chapter Contents

This appendix describes the details of the Cobalt System Configuration Protocol (CSCP). For an overview of how CSCP works with the rest of Sausalito, see Chapter 5, Introducing The Cobalt Configuration Engine . CSCP is enables communicaiton between a client application and CCE or when CCE communicates with a handler.

When a CSCP session begins, the server starts the connection by transmitting a CSCP header to the client. This header is described below in lazy-BNF notation.

```
Header ::= Identifier-Line ObjectID-Line? Ready-Line

Identifier-Line ::= "100 CSCP/" version nl

ObjectID-Line ::= "101 EVENT " object-id "." (namespace ".")?
property

Ready-Line ::= "200 Ready" nl
```

A handler is triggered because of some change in an object. The `ObjectID-Line` tells you the name of the `ObjectID` and the namespace.

NOTE: `ObjectID-Line` and `Attribute-Line` are only meaningful in the context of CCED communicating with an event handler.

Example Headers

When CCE connects to a client or a handler, the header is sent.This is an example header that a UI client would expect to see when connecting to CCED:

```
100 CSCP/1.0
200 Ready
```

These are example headers that an event handler would expect to see when CCED connects to the handler:

```
100 CSCP/1.0
101 Event 5._CREATE
200 OK

100 CSCP/1.0
101 Event 27.Foo.enable
200 OK

100 CSCP/1.0
101 Event 93..enable
200 OK
```

Messages

This section explains patterns repeatedly occur in CSCP. All lines sent by the server consist of a numeric code and a set of arguments. The first digit (hundreds place) of the code defines whther the message is informational, a warning, a success or a failure.

```
100-199 = Informational
200-299 = Success
300-399 = Warning
400-499 = Failure
900-999 = System issued message (can be sent at any time)
```

A response is made up of any number of 100 or 300 lines, finishing with a single 200 or 400 line. Response codes are shared between different commands. However, messages with the same code always share the same syntax, regardless of the command the message is responding to.

The lowest 30 codes of each `100` and `300` block and the lowest 10 codes of each `200` and `400` block is reserved for common messages. Codes outside those blocks are allocated as needed.

The following is a more detailed breakdown of allocations:

```
"100 CSCP/" version
"101 EVENT oid.event"
"102 DATA " key " = " val
"103 DATA " key " = " val (uncommitted)
"104 OBJECT " oid
"105 NAMESPACE " namespace
"106 INFO " msg
"107 CREATED"
"108 DESTROYED"
"109 SESSIONID " session-id-string
"110 CLASS " classname
111-119 : reserved
120-129 : reserved for protocol headers
130-199 : allocated for commands

"200 READY"
"201 OK"
"202 GOODBYE"
203-209 : reserved
210-299 : allocated for commands

"300 UNKNOWN OBJECT " oid
"301 UNKNOWN CLASS " class
"302 BAD DATA " oid " " key " " value
```

```
"303 UNKNOWN NAMESPACE " namespace
"304 PERMISSION DENIED" reason
"305 WARN " msg
"306 ERROR " msg
"307 OUT OF MEMORY"
308-329 : reserved
330-399 : allocated for commands

"400 NOT READY"
"401 FAIL"
"402 BAD COMMAND"
"403 BAD PARAMETERS"
404-410 : reserved
420-499 : allocated for commands

"998 SHUTTING DOWN"
"999 ENGINE ON FIRE"
```

CSCP Command Summary

This is the total set of CSCP commands:

Table 6—4 CSCP commands

Command	Description
AUTH	Authenticates as a user, to get that user's access privileges (starts a new session)
AUTHKEY	Authenticates to an already existing session
ENDKEY	Expire the current `sessionid`
WHOAMI	Returns the `OID` of the currently authenticated user
BYE	Closes the connection
COMMIT	Triggers any postponed handler activity

Table 6—4 CSCP commands

Command	Description
CREATE	Creates a new object of a certain class
DESTROY	Destroys an object
FIND	Finds all objects that match a given criteria
GET	Gets all properties of a certain object.
NAMES	Lists `namespaces` associated with a class (for example, services).
CLASSES	Lists all classes
SET	Sets the properties of a certain object.

These additional commands are available in handler mode, that is, when the `CCED` is communicating with a handler:

Table 6—5 Additional CSCPcommands for handler mode

Command	Description
BADDATA	Reports that an unrecognized attribute or value was passed
INFO	Report a piece of information
WARN	Report a warning or error

See Chapter 5 for the Perl, C, and PHP libraries of CSCP commands.

Common Syntax Definitions

Syntax for commands is described in lazy-BNF notation, that is, it is similar to BNF, but is human readable.

```
sp ::= [ \t]+ (any number of whitespace characters)
nl ::= '\n'
CLASSNAME ::= "SITE" | "USER" | "GROUP" | "MAILLIST"
alphanumeric_string ::= [A-Za-z0-9_]+
quoted_string ::= "\"" [^\"]* "\""
stringvalue ::= quoted_string | alphanumeric_string
KEY ::= stringvalue
```

```
VALUE ::= stringvalue
OID ::= stringvalue
```

CSCP Commands

The AUTH Command

The `auth` command authenticates the client to have the permissions of the specified user. To re-AUTH to the default (anonymous) user, specify `username` and password as blank strings ("").

Syntax:

```
"AUTH" sp USERNAME sp PASSWORD nl
```

`USERNAME` is the user's username.

`PASSWORD` is the user's password (unencrypted).

Return values: `109, 201, 401`

The AUTHKEY Command

The `authkey` commands authenticates to an existing session, assuming that sessions'`sid` and privileges.

Syntax:

```
"AUTHKEY" sp USERNAME sp SESSION-KEY nl
```

`SESSION-KEY` is an alphanumeric string that uniquely identifies a session-user pair.

Return values: `109, 201, 401`

The ENDKEY Command

The `endkey` command alerts the server to immediately expire the current `sessionid`, and not allow it as a parameter to `authkey`.

Syntax:

```
"ENDKEY" nl
```

Return values: `201`

The CREATE Command

The `create` command takes a class name and a list of attributes, and creates a new object of that type.

Syntax:

```
"CREATE" sp CLASSNAME ( sp KEY sp "=" sp VALUE )* nl
```

Informational responses:`104`

Warning responses: `301, 302, 303, 304`

Return values: `201, 401`

The DESTROY Command

The `destroy` command takes an oid and destroys the object.

Syntax:

```
"DESTROY" sp oid
```

Informational responses: `300, 304`

Return values: `201, 401`

The SET Command

The `set` command modifies the attributes of an existing object.

Syntax:

```
"SET" sp OID ("." NAMESPACE)? ( sp KEY sp? "=" sp? VALUE )* nl
```

Warning responses:`300, 302, 303, 304`

Return values: `201, 401`

The GET Command

The `get` command returns all of the current attributes for the specified object. In the face of transactions (such as a handler s view of the ODB), `get` will return both the previous state and the current state, in that order. There are two `info` messages to denote that an object was just created (has no previous state) or just destroyed (has no current state).

Syntax:

```
"GET" sp OID ("." NAMESPACE)? nl
```

Informational responses:102, 103, 107, 108

Warning responses: 300, 303

Return values:201, 401

The COMMIT Command

The `commit` command triggers any deferred activity.

Syntax:

"COMMIT" nl

Informational responses: `106`

Warning responses:`305`

Return values: 201, 401

The NAMES Command

The `names` command returns a list of all defined namespaces for a class.

Syntax:

```
"NAMES" sp (OID|CLASSNAME) nl
```

Informational responses: `105`

Warning responses: `300, 301`

Return values: `201, 401`

The CLASSES Command

The `CLASSES` command returns a list of all defined classes.

Syntax:

"CLASSES" nl

Informational responses: 110

Return values: 201

The FIND Command

The `find` command searches through object space to find all object of a given class that match a criteria.

Syntax:

"FIND" sp CLASSNAME (sp ("SORT"|"SORTNUM") sp SORTKEY)?

(sp KEY sp? "=" sp? VALUE)* nl

`Find` searches within the set of objects that belong to class `CLASSNAME`. `Find` finds all of the objects of that class whose properties match the properties set forth in the `KEY-VALUE` list. `KEY` may be of the form `PROPERTY` or `NAMESPACE.PROPERTY`.

If the `SORT` option is specified, the objects are returned in order, sorted alphanumeric ally from lowest to highest according to the value of the `SORTKEY` property of each object. `SORTKEY` may be of the form `PROPERTY` or `NAMESPACE.PROPERTY`.

If the `SORTNUM` option is specified, the objects are returned in order as with the `SORT` option, except that the objects are sorted in numeric order, that is, 9 < 10, as opposed to alphanumeric order, that is, 9 > 10, because 9 comes after 1 .

SORTNUM is capable of handling the sorting of integers (11), floating point numbers (12.54), and version numbers (v1.5.3.27). Version numbers are special numbers that must start with the letter v . They differ from floating point numbers in the sense that every group of digits within the version numbers is compared like an integer. For example:0.15 is less than 0.2 (floating point numbers), but v0.15 is greater than v0.2 (version numbers).

Informational responses: 104

Warning responses:301

Return values: 201, 401

The WHOAMI Command

Syntax:

"WHOAMI" nl

If the session is currently authenticated, whoami returns the OID of the user object that the connection is currently authenticated as. If the connection is not authenticated, or is authenticated as "" (anonymous), the OID returned will be -1.

Informational responses: 104

Return values: 201

The BYE Command

The Bye-Condition field is optional, and is ignored unless CCEd is talking to an event handler, that is, in handler mode.

In a handler context, if the "Bye-Condition" is omitted (or if the handler exits without issuing a "BYE" command), the handler is assumed to have failed (for example, as if the handler had issued the command "BYE FAIL").

Syntax:

```
"BYE" Bye-Condition? nl
Bye-Condition ::= ( Bye-Success | Bye-Failure | Bye-Defer )
Bye-Success ::= "SUCCESS"
```

```
Bye-Failure ::= "FAIL"
Bye-Defer   ::= "DEFER"
```

Return values: 202

CSCP Handler Extensions

The BADDATA Command

The `BADDATA` command is used by a handler to report that one of the attributes or data in the current operation isn't valid for the specified class and namespace.

Reporting bad data is left to the discretion of the handler. Handlers can choose to not flag BADDATA errors if they want to facilitate future extensions to a namespace.

Syntax:

```
"BADDATA" sp OID sp KEY sp VALUE nl
```

Return value: `201`

The INFO Command

The `INFO` command is used by a handler to report some piece of info for use by the front-end. The parameter `MESSAGE` is a single string formatted as listed below. This format allows easy parsing by internationalization software at higher levels.

Syntax:

```
"INFO" sp MESSAGE nl
```

MESSAGE ::= domain ":" tag (<sp>+ var1name <sp>* "=" <sp>* var1val)*

Variable names must follow all the same guidelines as a property name, and variable values must be alphanumeric or a properly quoted and escaped string.

Return value: `201`

The WARN Command

The `WARN` command is used by a handler to report some piece of information for use by the front-end. See `INFO` command for information on `MESSAGE` parameter format.

Syntax:

```
"WARN" sp MESSAGE nl
```

Return value: `201`

Built-in Properties of Objects

Using the `get` command, a hash is returned from the Object Database (ODB). In addition to ordinary properties, it also has these magic properties inserted in it:

OID	The unique identifier number for the object
CLASS	The class of the object
NAMESPACE	The namespace of the the subset of properties retrieved

Appendix F

CCE Class Definitions

Chapter Contents

NOTE: In a later draft, properties will be labeled optional and required.

CAUTION! The class definition properties are subject to change. Check this section for updates.

Programming Conventions

The class definitions use the following conventions:

- All class names have the first character capitalized. for example, `System`. If they have more than one word, the first character of all words is capitalized, for example, `MailList`.
- Namespace names follow the same rule as class names.
- All property names start with an all lowercase first word. If a property name has more than one word, the first characters of the second word onwards are capitalized. For example, `gateway` and `stylePreference` are valid property names.

CCE Class Definitions

System

`System` stores all system-wide configuration settings. There should be exactly one `System` object in every functional system.

`Table C–1 Network`

Properties	Definition
`hostname`	the name of the host (first half of the Fully Qualified Domain Name)
`domainname`	the domain name of the host (second half of the Fully Qualified Domain Name)
`gateway`	IP address of the default gateway
`DNS`	colon (:) delimited list of DNS server IP addresses
`notify_email`	Address to whom to email emergency reports
`time_region`	Used by UI to select timezones

Table C–1 Network

Properties	Definition
time_country	Used by UI to select timezones
time_zone	Time zone to use
reboot	Set to true to enable rebooting the machine, and is cleared whenever CCE is restarted.
halt	Set to true to halt the machine. Cleared when restarted

Network

Network stores settings relevant to the basic (non-virtual) TCP/IP network interfaces.

Table C–2 Network

Properties	Definition
device	Usually either eth0 or eth1
ipaddr	IP address for this interface
netmask	netmask for this interface
Media Address Control	MAC address of this interface
enabled	True to bring the interface up, false to take it down
bootproto	Either DHCP, none, or LCD

Route

route used to add additional gateways for some routes.

Table C–3 route

Properties	Definition
target	the destination subnet or host to perform routing
netmask	netmask of target subnet
gateway	IP address of gateway for this subnet
device	device (defaults to the device gateway is within)

Workgroup

`workgroup` stores all workgroup-specific settings.

`Table C–4 workgroup`

Properties	Definition
`enabled`	determines if the workgroup enabled (Boolean)
`members`	colon-delimited list of usernames who are members of this group
`name`	the unique name of this workgroup (alphanumeric)
`quota`	disk space quota for this workgroup (integer)

Workgroup Defaults

`workgroup defaults` stores workgroup defaults

Table C–5 `workgroup defaults`

Properties	Definition
`quota`	allowed disk space (in megabytes)

User

`User` stores all user-specific settings.

Table C–6

Properties	Definition
`enabled`	Used to enable or disable the users account
`fullName`	The full comment name of the user
`localePreference`	Used exclusively by the UI
`name`	The unique name of this user (alphanumeric)
`password`	The user's plaintext password
`sortName`	The string to use when sorting users
`shell`	Path to the user's shell
`site`	The name of the site to which the user belongs

Table C–6

Properties	Definition
`siteAdministrator`	`flag`: is the user a site administrator?
`stylePreference`	Used exclusively by the UI
`systemAdministrator`	`flag`: is the user a system administrator?

UserDefaults

`UserDefaults` stores user defaults.

Table C–7 User Defaults

Properties	Definition
`quota`	allowed disk space (megabytes)
`userNameGenMode`	The mode for user name generation. It can be `firstInitLast`, `first` or `last`.

MailList

`MailList` represents a mailing list.

Table C–8 MailList

Properties	Definition
`name`	alphanumeric name of the mailing list
`password`	password for authenticating mail-admin commands
`postPolicy`	Rules to restrict who can post to the list
`moderated`	indicates that only moderators can post
`any`	anybody can post
`members`	only members can post
`subPolicy`	Rules to restrict who can subscribe to the list
`closed`	only the admin or moderaters can subscribe users
`open`	anybody can subscribe

Table C–8 MailList

Properties	Definition
`confirm`	anybody can *subscribe?*, confirmation required for subscriptions
`local_recips ","`	delimited list of local usernames to receive mail
`remote_recips ","`	delimited list of remote usernames
`moderator`	The list moderator
`group`	What group is this mailing list associated with (for quota purposes)
`site`	This field should always be empty for Qube3
`enabled`	Is list active? (Boolean value; default is true)

User.Email

`User.Email` determines email-specific properties

Table C–9

Properties	Definition
`enabled`	Determines if email is enabled
`aliases`	List of email aliases for this user
`forward`	Address to forward this user's email
`vacation`	Is user on vacation? What's the message?
`url`	url of UI for configuring email properties
`apop`	Whether APOP is active for this user

System.Email

`System.Email` lists System specific email properties

Table C–10

Properties	Definition
`acceptFor`	`Array[Host|Domain]` for which to accept mail.
`denied`	`Array[Host|Domain]` for which to deny mail.
`relayFor`	`Array[Host|Domain]` to relay for

Table C–10

Properties	Definition
`routes`	`Hash[Domain:Domain]` redirection
`masqDomain`	Domain to masaquerade as
`smartRelay`	Host to forward mail to
`deliveryMode`	Alter('interactive','background','queue') Controls how email is delivered.
`privacy`	Boolean representing high email security (Expn/Vrfy).
`maxMessagesSize`	Integer (In megabytes) largest size email to accept and send. 0 to disable.
`queueTime`	Alter ('daily','half-daily'... and others) Controls how often mail is queued per `cron` intervals.)

System.FTP

System.FTP determines FTP settings.

Table C–11

Properties	Definition
`enabled`	determines whether FTP is enabled
`anon_en`	determines whether anonymous FTP is enabled
`maxusers`	number of simultaneous users who can be logged in
`quota`	quota for anonymous files
`url`	URl of UI for configuring FTP properties

System.Snmp

SNMP settings are settings for Simple Network Mail Protocol.

Table C–12 SNMP

Properties	Definition
`enabled`	determines whether SNMP server is enabled.
`readCommunity`	read-only SNMP community
`readWriteCommunity`	the read and write SNMP community

DhcpParam

DhcpParam are parameters for DHCP clients.

Table C–13DhcpParam

Properties	Definition
enabled	DHCPd on or off flag
domainname	the domain name of the host (second half of the fully qualified domain name (FQDN))
gateway	IP address of the default gateway
dns	colon-delimited list of DNS server IP addresses.
netmask	netmask for this network
lease	the maximum lease time in seconds

DhcpStatic

DhcpStatic configures static address assignments.

Table C–14DhcpStatic

Properties	Definition
ipaddr	IP address to assign to this mac address
mac	Media Access Control (MAC) address to get the above IP Address

DhcpDynamic

DhcpDynamic provides configuration for dynamic address range assignments

Table C–15DhcpDynamic

Properties	Definition
ipaddrlo	the beginning of the range
ipaddrhi	the end of the range

Index

Symbols

A

B

C

D

E

T

U

V

W

X