InterWorx Plugin Developer Guide

by InterWorx LLC

Contents

Chapter 1

Introduction

1.1 Why Have Plugins?

The InterWorx Plugin system allows users to extend the functionality of the control panel in a wide variety of ways. With this guide and a bit of scripting experience, any user can leverage the complete InterWorx API to achieve endless possibilities.

A great deal of existing functionality in InterWorx can be leveraged to quickly bring any plugin concept to reality. Adding menu items to the interface, modifying existing input forms, and adding listeners to existing actions are all easily included, and substantially cut down on implementation and development time. Mostly importantly, plugins allow for the open-ended addition of features to customize your users' experience.

1.2 What Can Plugins Do?

Anything that can be scripted with the InterWorx API can be packaged into a plugin, which means nearly any action that can be performed within NodeWorx and SiteWorx can be performed using our plugin system. Plugins can modify existing behavior, add new features and behavior, or even disable existing behavior of the NodeWorx and/or SiteWorx control panel experience.

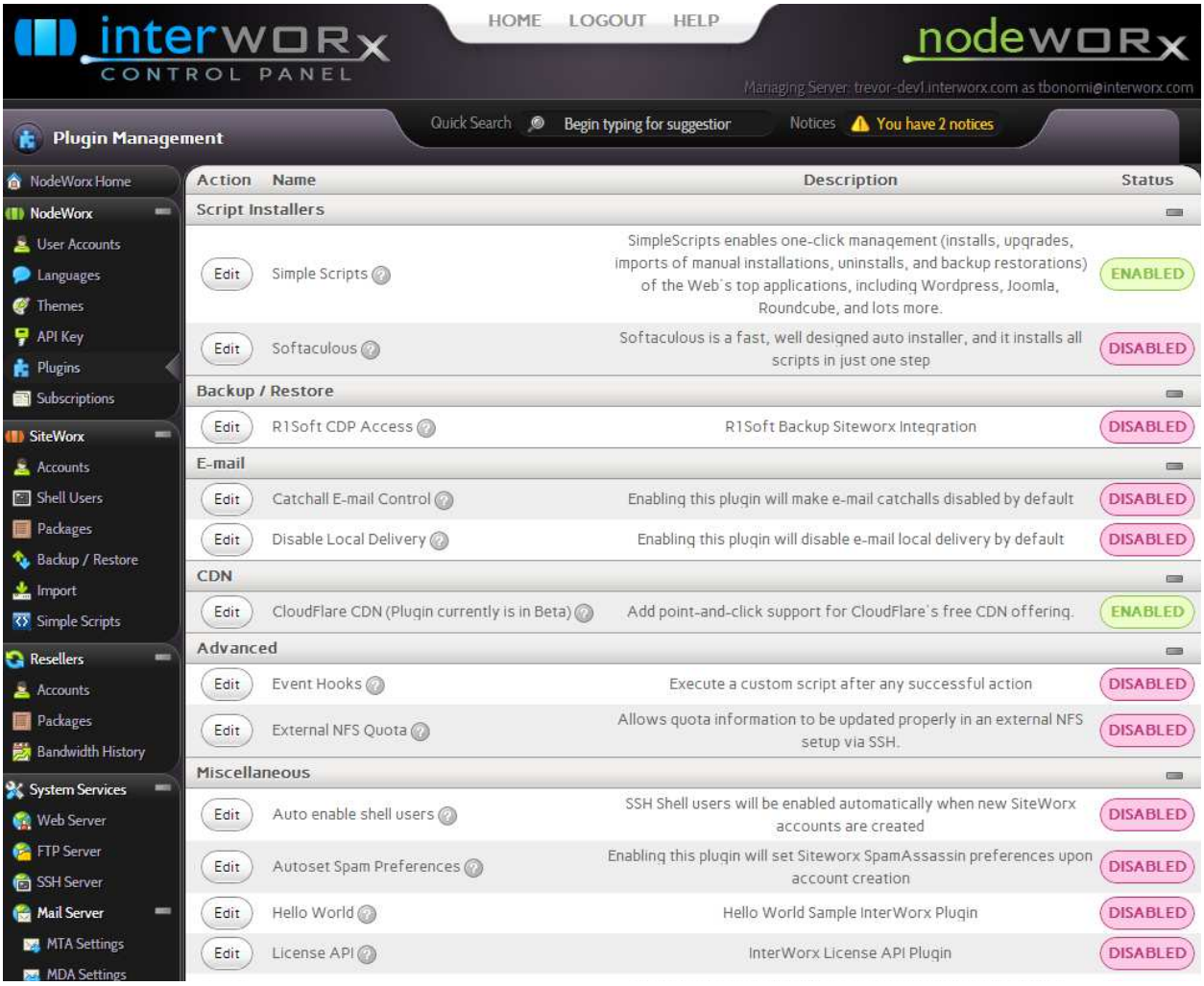

Figure 1.1: The Plugin Management page in NodeWorx

Chapter 2

Getting the Lay of the Land

2.1 Basic Requirements

Plugin developers should have some experience with the PHP programming language and at a minimum a basic understanding of object-oriented programming.

2.2 NodeWorx vs SiteWorx

- **NodeWorx** is the server level control panel interface, accessible via http://yourserver.tld:2443/nodeworx . It provides server-level controls for managing SiteWorx accounts, reseller accounts, system services, backups and restoring, and clustering, as well as system-level IP management, distribution, and much more.
- **SiteWorx** is the website level control panel interface, accessible via http://yoursite.tld:2443/siteworx . It provides tools for maintaining domain-level users, IPs, e-mail accounts, MySQL databases, statistics analysis, backups, an interface for uploading and downloading files to and from your web space, among other services.

2.3 Controllers and Actions

- A **controller** is a part of programming architecture that houses the **actions** for a particular segment of an application. The controller **actions** allow the user to modify the interface and output as needed. Each controller generally corresponds to a "page" in the control panel.
- An **action** is a thing that can be performed. Some actions change things, some actions query data.
- The default action for a controller is called the **indexAction**. This action is executed when the controller is loaded without any other actions being called. It is the action that generally displays the default content for the controller or page, if applicable.
- **Commit actions** are those that perform a measurable, concrete change to the system. As opposed to actions that simply query information or list data, commit actions are typically used to add, edit, or delete data records at a user's request. Commit actions with forms require clear and concise function comments, with the following format:

- Plugins can hook into existing Controller Actions, or establish new Controllers and Actions.
- Plugins can also take advantage of existing InterWorx actions.

Figure 2.1: **~iworx/plugins/cloudflare/Ctrl/SW/Cloudflare.php**

```
1 /**
2  * Enable Cloudflare for a subdomain.
3  *
4  * @param Form_SW_Cloudflare_Enable $Form
5  */
6 public function enableCommitAction( Form_SW_Cloudflare_Enable $Form ) {
```

2.4 Menus

NodeWorx and SiteWorx each have menus that are used to navigate through their respective interfaces. There are two types of "menu styles", known as the "big" and "small" menu styles. The big menu style displays the menu as a full page set of icons when the user logs in. The small menu style displays the menu on the left hand side of the interface. Plugins can interact with the menu, and any change made will automatically be reflected in either menu style.

Plugins interact with the menus by defining **updateSiteworxMenu()** or **updateNodeworxMenu()** functions in the Plugin Class file (see below).

2.5 Templates

Plugin templates are **view components** responsible for the elements displayed to the user. Managed by the plugin controllers, **templates** are where visual elements are defined and laid out.

Via the controller actions, developers can modify the template output (and thereby the view) to reflect state changes in the plugin data.

2.6 Files and Directory Structure

- **Required Files**
  The required files for a proper plugin structure are few, but each is key to the proper functioning of the plugin:
  - **Plugin Directory: ~iworx/plugins/[plugin-name]**
    This is the directory where all files relevant to the plugin are stored. Here, 'plugin-name' should be short, hyphen separated (no spaces or underscores), and lowercase.
  - **Plugin INI file: ~iworx/plugins/[plugin-name]/plugin.ini**
    The plugin INI contains plugin metadata, such as the version, author, and a brief description.
  - **Plugin Class file: ~iworx/plugins/[plugin-name]/Plugin/[PluginName].php**
    This is the core of the plugin and contains the majority of its functionality, such as data storage, InterWorx menu updates, and enabling/disabling of the plugin. 'PluginName.php' should reflect the naming of 'plugin-name' above, with hyphens removed and each word capitalized.
- **Optional Files**
  The following files are optional to the core operation of a plugin, but allow users to further personalize their plugin interface:
  - **Plugin NodeWorx Controller file: ~iworx/plugins/[plugin-name]/Ctrl/Nodeworx/[PluginUrl].php**
    With this file present, a user will be able to control how the plugin is represented in the NodeWorx interface (if necessary). The naming of this file determines the URL of the controller – in this case we'd find it at **/nodeworx/plugin/url**.

Figure 2.2: Small Menu Style Example

Figure 2.3: Big Menu Style Example

  - **Plugin SiteWorx Controller file: ~iworx/plugins/[plugin-name]/Ctrl/Siteworx/[PluginUrl].php**
    Much like above, this file allows for control of the plugin with regards to the SiteWorx interface. The naming of this file determines the URL of the controller – in this case we'd find it at **/siteworx/plugin/url**.
  - **Plugin default template file: ~iworx/plugins/[plugin-name]/templates/default.tpl**
    The core template file, this file controls the layout and presentation of the plugin's output in-browser.

- Other files can be included as necessary, such as those for input forms, images, payloads, formatters, custom libraries, and data sources. These are more advanced features and described in detail later on.

Chapter 3

Your First InterWorx Plugin

3.1 A Plugin of Your Very Own

The quickest way to build your first plugin is to create a working framework using the key components outlined in chapter 2. With the structure in place, additional functionality can be added in a much more organized fashion.

A basic example is the standard "Hello World" plugin, comprised of the following files:

- **Plugin Class file: ~iworx/plugins/hello/Plugin/Hello.php**
  This file is the plugin "core". The majority of functionality and configuration of the plugin is here, and can be modified to suit a variety of needs. [see Figure 3.1]

- **~iworx/plugins/hello/Ctrl/Nodeworx/HelloWorld.php**
  Since we updated the NodeWorx menu to include our new plugin (line 62), we should define a controller for it. Assigning the title below (line 12) will allow us to specify the title header in-browser. [see Figure 3.2]

- **~iworx/plugins/hello/Ctrl/Siteworx/HelloWorld.php**
  Same as above, we've included the plugin on the SiteWorx menu, so we should define a controller and set the title (line 12). [see Figure 3.3]

- **~iworx/plugins/hello/plugin.ini**
  The plugin INI is where the plugin metadata is stored, much of which is displayed in **NodeWorx > Plugin Management**. [see Figure 3.4]

- **~iworx/plugins/hello/templates/default.tpl**
  The template for this plugin isn't too complicated – just some text to display on the page. Templates ordinarily are where forms and payloads would be included. [see Figure 3.5]

Figure 3.1: ~iworx/plugins/hello/Plugin/Hello.php

```
/**
 * Hello World plugin.
 *
 * @package    InterWorx
 * @subpackage Plugin
 */
class Plugin_Hello extends Plugin {
  /**
   * Init the plugin, disabling if necessary.
   */
  protected function _init() {
    /*
     * You can conditionally disable the plugin
     */
    //$this->_disable( 'reason for disabling goes here, will go in iworx.log' );

    /*
     * You can set plugin variables as needed like this
     */
    //$this->_setVar( 'myvariable', 'somevalue' );

    /*
     * And retrieve them later like this
     */
    //$this->getVar( 'myvariable' );

    /*
     * You can set mixed datastore variables as needed like this
     */
    //$this->setDatastoreVar( 'storedvar', array( 1, 2, 3 ) );

    /*
     * And retrieve it later like this
     */
    //$this->getDatastoreVar( 'storedvar' );
  }

  /**
   * Customizations to siteworx menu can be done here.
   *
   * @param IWorxMenuManager $MenuMan
   */
  public function updateSiteworxMenu( IWorxMenuManager $MenuMan ) {
    $new_data = array( 'text'   => 'Hello World',
                       'url'    => '/siteworx/hello/world',
                       'class'  => 'iw-i-star' );

    $MenuMan->addMenuItemAfter( 'iw-menu-home', 'hello', $new_data );
    // also updateMenuItem( $id, $data )
    //      removeMenuItem( $id )
  }

  /**
   * Customizations to siteworx menu can be done here.
   *
   * @param IWorxMenuManager $MenuMan
   */
  public function updateNodeworxMenu( IWorxMenuManager $MenuMan ) {
    $new_data = array( 'text'   => 'Hello World',
                       'url'    => '/nodeworx/hello/world',
                       'class'  => 'iw-i-star' );
    $MenuMan->addMenuItemAfter( 'iw-menu-home', 'hello', $new_data );
    // also updateMenuItem( $id, $data )
    //      removeMenuItem( $id )
  }
}
```

Figure 3.2: **~iworx/plugins/hello/Ctrl/Nodeworx/HelloWorld.php**

```
/**
 * Nodeworx hello world sample plugin controller.
 *
 * @package    InterWorx
 * @subpackage Plugin
 */
class Ctrl_Nodeworx_HelloWorld extends Ctrl_Nodeworx_Plugin {
  /**
   * Default action.
   */
  public function indexAction() {
    $this->getView()->assign( 'title', 'Hello World Sample Plugin, nodeworx' );
  }
}
```

Figure 3.3: **~iworx/plugins/hello/Ctrl/Siteworx/HelloWorld.php**

```
/**
 * Siteworx hello world sample plugin controller.
 *
 * @package    InterWorx
 * @subpackage Plugin
 */
class Ctrl_Siteworx_HelloWorld extends Ctrl_Siteworx_Plugin {
  /**
   * Default action.
   */
  public function indexAction() {
    $this->getView()->assign( 'title', 'Hello World Sample Plugin, siteworx' );
  }
}
```

Figure 3.4: **~iworx/plugins/hello/plugin.ini**

```
[plugin]
name="Hello World"
description="Hello World Sample InterWorx Plugin"
details="This plugin adds a 'Hello World' menu item to the top of BOTH NodeWorx and SiteWorx."
version="1.0"
author="interworx"
```

Figure 3.5: **~iworx/plugins/hello/templates/default.tpl**

```
Hello, world!
```

Chapter 4

Plugin Advanced Feature Overview

These advanced features are discussed briefly here, and the case studies below demonstrate specific usage examples.

4.1 Form System

InterWorx's form system is the ideal means to collect data from user input, and is composed of several parts:

- **Forms**
  Much like controllers, forms are container elements that hold the various input objects that comprise a submittable form.
- **Inputs**
  Inputs come in many varieties, and can represent nearly any sort of user-provided data. Basic types include strings, integers, and selects. They can also be extended to handle specific formats to suit any need.
- **Validators**
  Validators are objects that can be assigned to inputs and are useful for validating input coming in via forms.
- **Datasources**
  Datasources are objects that define what values are invalid for a given input (for example, with a select, checkbox, or radio input).

It is not required that the InterWorx Form System be used by custom plugins, but doing so includes some nice benefits, such as built-in round-trip ajax validation, automatic form html and javascript management, etc.

4.2 Payload System

Payloads are an integral part of how information is displayed within InterWorx. Payloads allow developers to easily present and update data dynamically in a wide variety of contexts. To accomplish this, each payload typically includes several key components:

- **Payload Factory Classes**
  The primary payload object, payload factory classes allow the various types of payloads (Payload_Factory_NW_Dns or Payload_Factory_SW_Users, for example) to be instantiated as needed.
- **Payload Data Formats**
  The data to be displayed is called the Payload Data, and is a numerically-indexed array of similarly defined stdClass objects.
- **Payload Formatting**
  Payload Formatters allow various aspects of a payload – columns, cells, headers – to be formatted by a configurable set of rules. These are only relevant in the web interface.

Figure 4.1: Form System Interface Elements

- **Payload Actions**
  By providing an array of controller actions and various format parameters, payloads can optionally execute actions on the displayed data.

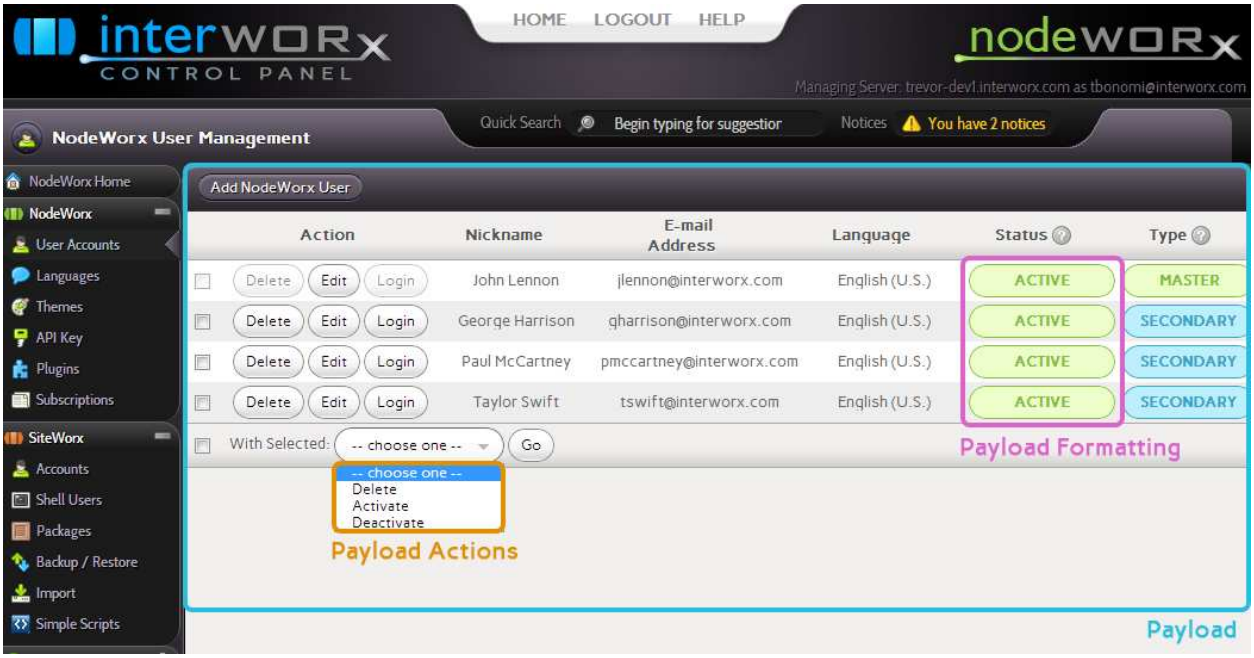

Figure 4.2: Payload System Interface Elements

4.3 Switching Users

For certain user operations, it is advantageous to execute an action as that user (instead of the default iworx user). In such a case the **~iworx/bin/runasuser.pex** can be called with parameters for the user's Unix name and the script or program to be called.

Chapter 5

Case Studies

As mentioned previously, only a small subset of files are actually necessary for the most basic of plugins. As more in-depth functionality is needed, however, more files may be required. The following programs contain examples of how additional utility can be achieved with the InterWorx plugin system.

5.1 Plugin Case Study: auto-enable-shell-account

Auto Enable Shell Account is a plugin designed to activate shell account access for all new SiteWorx users added to an account. To accomplish this, it needs to perform only a small handful of tasks, but far more than we've seen thus far.

- The first new function encountered here is **initializeMyEditForm()** (line 18). This optional function allows us to define settings that will be configurable via the plugin menu in **NodeWorx > Plugin Management**. It takes a form object as input, **Form_NW_Plugins_Edit**. At the start of the function a new form group is added (line 19), and datastore variables within the plugin (lines 20 and 22) are accessed (they are stored in the expected fashion, seen below). [see Figure 5.1]

Figure 5.1: **~iworx/plugins/auto-enable-shell-account/Plugin/AutoEnableShellAccount.php**

```
1  /**
2   * Plugin to auto-enable ssh shell users on account creation.
3   *
4   * @package    InterWorx
5   * @subpackage Plugin
6   */
7  class Plugin_AutoEnableShellAccount extends Plugin {
8    /**
9     * Init my edit form.
10    *
11    * Add plugin settings that specify which resellers this feature is enabled
12    * for.
13    *
14    * @param Form_NW_Plugins_Edit $Form
15    */
16   public function initializeMyEditForm( Form_NW_Plugins_Edit $Form ) {
17     $Group = $Form->addGroup( 'extra_options', 'Extra Plugin Settings' );
18     $all_resellers = $this->getDatastoreVar( 'all_resellers' );
19     $Group->addInput( new Input_Flag( 'all_resellers', $all_resellers ) );
20     $resellers = $this->getDatastoreVar( 'resellers' );
21     if( $resellers === '' ) {
22       $resellers = array();
23     }
24     $Group->addInput( new Input_Select( 'resellers', $resellers ) )
25       ->setDataSource( new DataSource_NW_ResellerIds() )
26       ->setRequired( true )
27       ->setMultipleAllowed( true );
28   }
```

- The next function processes the edit form we just initialized – the aptly-named **processMyEditForm()** (line 8). Taking the same form as input, we use this space to handle the data submitted in the previous step, optionally storing it with **setDatastoreVar()** (lines 14 and 15). [see Figure 5.2]

Figure 5.2: **~iworx/plugins/auto-enable-shell-account/Plugin/AutoEnableShellAccount.php**

```
1    /**
2     * Process my edit form.
3     *
4     * @param Form_NW_Plugins_Edit $Form
5     */
6    public function processMyEditForm( Form_NW_Plugins_Edit $Form ) {
7      if( $Form->getValue( 'all_resellers' ) ) {
8        $resellers = array();
9      } else {
10       $resellers = $Form->getValue( 'resellers' );
11     }
12     $this->setDatastoreVar( 'all_resellers', $Form->getValue( 'all_resellers' ) );
13     $this->setDatastoreVar( 'resellers', $resellers );
14   }
```

- Next, we simply set the priority of the **plugin**. Plugin priority determines the order in which plugins are initialized in the system. [see Figure 5.3]

Figure 5.3: **~iworx/plugins/auto-enable-shell-account/Plugin/AutoEnableShellAccount.php**

```
1  /**
2   * Get priority.
3   *
4   * @return integer
5   */
6  public function getPriority() {
7    return 40;
8  }
```

- Lastly, we have the **postAction()**, where the majority of the plugin functionality resides. It accepts inputs that include the Controller, Action, and Form, and proceeds to use **routeFromPHP()** (lines 37 and 40) to call the NodeWorx Shell controller, which then enables the recently-added SiteWorx user (line 46). [see Figure 5.4]

Figure 5.4: **~iworx/plugins/auto-enable-shell-account/Plugin/AutoEnableShellAccount.php**

```
1  /**
2   * Handle the controller:action we care about.
3   *
4   * @param string        $ctrl_act
5   * @param Ctrl_Abstract $Ctrl
6   * @param string        $action
7   * @param mixed         $params
8   */
9  public function postAction( $ctrl_act, Ctrl_Abstract $Ctrl, $action, $params ) {
10   if( $ctrl_act !== 'Ctrl_Nodeworx_Siteworx:addCommit' ) {
11     return;
12   }
13   if( !$Ctrl->isViewSuccess() ) {
14     return;
15   }
16   if( $this->_userHasNoAccess() ) {
17     return;
18   }
19   assert( $params instanceof Form );
20   /**
21    * @var Form.
22    */
23   $Form     = $params;
24   $uniqname = $Form->getValue( 'uniqname' );
25   $password = $Form->getValue( 'password' );
26   $ctrl = 'Ctrl_Nodeworx_Shell';
27   if( IW::NW()->isReseller() ) {
28     $NW = new NodeWorx( NodeWorx::MASTER_ID );
29   } else {
30     $NW = IW::NW();
31   }
32   $action = 'changeshell';
33   $input = array( 'shell' => Ini::get( Ini::SHELL, 'default' ),
34                   'users' => array( $uniqname ) );
35   $Reply = IW::FC()->routeFromPHP( $ctrl, $action, $input, $NW );
36   $action = 'enable';
37   $input = array( 'users' => array( $uniqname ) );
38   $Reply = IW::FC()->routeFromPHP( $ctrl, $action, $input, $NW );
39   $sshConfig = SSHD::readConfig();
40   if( count( $sshConfig['allowusers'] ) > 0 ) {
41     $SSH       = new SSHD();
42     $allowed   = $sshConfig['allowusers'];
43     $allowed[] = $uniqname;
44     $SSH->setAllowUsers( $allowed );
45     $SSH->writeConfig();
46   }
47   if( $Reply->wasSuccessful() === true ) {
48     $msg = 'Shell account user enabled';
49   } else {
50     $msg = "Tried to enable shell user {$uniqname}, but failed.";
51   }
52   $Ctrl->getView()->addMessage( $msg );
53 }
```

5.2 Plugin Case Study: CloudFlare

The InterWorx **CloudFlare** plugin allows for full support of CloudFlare's CDN technology from within SiteWorx. As one of the larger plugins, it is a prime example of how easily the core plugin functionality can be extended.

- The CloudFlare SiteWorx controller contains two prime examples of previously mentioned commit actions (lines 8 and 25). Both actions take very similar forms as input and call similar private functions (lines 12 and 29), defined elsewhere. [see Figure 5.5]

Figure 5.5: **~iworx/plugins/cloudflare/Ctrl/SW/Cloudflare.php**

```
1  /**
2   * Enable Cloudflare for a subdomain.
3   *
4   * @param Form_SW_Cloudflare_Enable $Form
5   */
6  public function enableCommitAction( Form_SW_Cloudflare_Enable $Form ) {
7    $parent    = $Form->getValue( 'parent' );
8    $subdomain = $Form->getValue( 'subdomain' );
9    $errors = array();
10   $this->_enableSubdomain( $subdomain, $parent, $errors );
11   if( empty( $errors ) ) {
12     $this->_getPlugin()->setDomainStatusData( $subdomain, '1' );
13     $this->_displaySuccess( 'index', 'CloudFlare Enabled on ' . $subdomain );
14   } else {
15     $this->_displayFailure( 'index', $errors );
16   }
17 }
18
19 /**
20  * Disable Cloudflare for a subdomain.
21  *    * @param Form_SW_Cloudflare_Disable $Form
22  */
23 public function disableCommitAction( Form_SW_Cloudflare_Disable $Form ) {
24   $parent    = $Form->getValue( 'parent' );
25   $subdomain = $Form->getValue( 'subdomain' );
26   $errors = array();
27   $this->_disableSubdomain( $subdomain, $parent, $errors );
28   if( empty( $errors ) ) {
29     $this->_getPlugin()->setDomainStatusData( $subdomain, '0' );
30     $this->_displaySuccess( 'index', 'CloudFlare Disabled on ' . $subdomain );
31   } else {
32     $this->_displayFailure( 'index', $errors );
33   }
34 }
```

- Looking at that first form, we can see that building a basic input form isn't complicated. [see Figure 5.6]

- After the form is instantiated, the subdomain input is added using **DataSource_Subdomains()** (line 18), described below. Shown here, it can be seen that the datasource simply consists of a subset of CNAME DNS records, passed to the constructor as an array of subdomain names (line 28). [see Figure 5.7]

- Building a payload, while one of the more complicated objects, is still straightforward. Columns are added to the payload by creating new **Payload_Columns** (lines 25, 26, 31 and 32), and have a considerable number of functions for configuration (setName, setOrderBy, etc). [see Figure 5.8]

Figure 5.6: **~iworx/plugins/cloudflare/Form/SW/Cloudflare/Enable.php**

```
1  /**
2   * Form_SW_Cloudflare_Enable class.
3   *
4   * @package    InterWorx
5   * @subpackage Form
6   *
7   */
8  class Form_SW_Cloudflare_Enable extends Form {
9    /**
10    * GetInputForm function.
11    */
12   public function initInputForm() {
13     $Form = new Form_Input();
14     $Form->addInput( new Input_String( 'parent' ) );
15     $parent = $Form->getValue( 'parent' );
16     $DS = new DataSource_Subdomains( $parent );
17     $Form->addInput( new Input_Select( 'subdomain' ) )
18       ->setDataSource( $DS );
19     $this->setInputForm( $Form );
20   }
21
22   /**
23    * Initialize.
24    */
25   protected function _initialize() {
26     parent::_initialize( 'enable_cloudflare_domain' );
27     $this->addInput( $this->getInput( 'subdomain' )->getDisabledCopy() );
28   }
29 }
```

Figure 5.7: **~iworx/plugins/cloudflare/DataSource/Subdomains.php**

```
1  /**
2   * DataSource for CloudFlare subdomains.
3   *  * @package    InterWorx
4   * @subpackage Input
5   */
6  class DataSource_Subdomains extends DataSource_Array {
7    /**
8     * Constructor.
9     *
10    * @param string $parent
11    */
12   public function __construct( $parent ) {
13     $subdomains = array();
14     $dns = IW::SW()->getSimpleDnsRecordsFor( $parent );
15     foreach( $dns as $record ) {
16       if( preg_match( '/^ftp\.*/', $record['host'] ) ) {
17         continue;
18       }
19       if( preg_match( '/^cloudflare-resolve-to\.*/', $record['host'] ) ) {
20         continue;
21       }
22       if( $record['type'] === Dns_Record::CNAME ) {
23         $subdomains[] = $record['host'];
24       }
25     }
26     parent::__construct( $subdomains );
27   }
28 }
```

Figure 5.8: **~iworx/plugins/cloudflare/Payload/Factory/SW/Cloudflare.php**

```
1  /**
2   * Factory for CloudFlare configuration.
3   *
4   * @package    InterWorx
5   * @subpackage Payload
6   */
7  class Payload_Factory_SW_Cloudflare {
8    /**
9     * Domains payload.
10    *
11    * @param SiteWorx $SW
12    * @param string   $domain
13    * @return Payload
14    */
15   static public function subdomains( SiteWorx $SW, $domain = null ) {
16     $plugin = IW::PluginManager()->getPlugin( 'cloudflare' );
17     $domains = self::_getDomains( $SW, $domain );
18
19     $Payload = new Payload( $domains );
20     $Payload->setName( 'cloudflare_domains' );
21     $Payload->setTitle( 'Cloudflare DNS Settings' );
22
23     $Payload->setColumn( new Payload_Column( 'type' ) );
24     $Payload->setColumn( new Payload_Column( 'subdomain' ) )
25       ->setIsSortable( true )
26       ->setOrderBy( 'subdomain' )
27       ->setLabel( 'Subdomain' );
28
29     $Payload->setColumn( new Payload_Column( 'record' ) );
30     $Payload->setColumn( new Payload_Column( 'status' ) )
31       ->addFormatter( new IWorx_Formatter_CloudflareEnabledDisabled( 'subdomain' ) )
32       ->setLabel( '##LG_STATUS##' );
33
34     return $Payload;
35   }
36 }
```

5.3 Plugin Case Study: Session History

The Session History plugin allows the system administrator to log user activity across the system for any number of purposes. It provides very standard examples of both formatters and templates.

- The page formatter takes a row from the payload output (line 19) and parses the controller and action name into a convenient link (line 21). [see Figure 5.9]

Figure 5.9: **~iworx/plugins/history/IWorx/Formatter/History/Page.php**

```
1  /**
2   * Displays the page uri based on the ctrl and action name.
3   *
4   * @package    InterWorx
5   * @subpackage Payload
6   */
7  class IWorx_Formatter_History_Page extends IWorx_Formatter_Complex {
8    /**
9     * Format.
10    *
11    * @param mixed  $value The value being rendered
12    * @param object $row   The data for the entire row
13    * @param array  $data  The entire dataset of the Payload
14    *
15    * @return string
16    */
17   public function format( $value, $row, array $data ) {
18     $ctrl = Ctrl_Util::convertClassNameToPath( $row->ctrl_name );
19     $link = "{$ctrl}?action={$row->action_name}";
20     return $link;
21   }
22 }
```

- Finally, the history template is a textbook sample of the possibilities of the template system, using the Smarty syntax. [see Figure 5.10]

Figure 5.10: **~iworx/plugins/history/templates/history.tpl**

```
{iw_add_js file="/plugins/images/history/search_session_history.js"}

{literal}
<style scoped>
  .iw-fake-table-hd .iw-form-img {   display: none;   }
  .td-timestamp {   white-space: nowrap;   min-width: 120px;   }
  .td-query_string {   white-space: nowrap;   }
  .td-change, .th-change {   max-width: 200px;   }
</style>
{/literal}

<div id="history-search" class="iw-fake-table-hd">
  <div class="right">
    {iw_form_input form=$search_form input_name=search}
    {iw_quickhelp id="LG_QH_SESSION_HISTORY_SEARCH"}
    {iw_form_buttons form=$search_form}
  </div>

  <div class="clear">
  </div>
</div>
{iw_payload payload=$iw_payload}
```