An Oracle Technical White Paper
February 2014

Effectively Managing the Oracle ZFS Storage Appliance with Scripting

ORACLE

Table of Contents

Introduction

The Oracle ZFS Storage Appliance is typically managed using a browser user interface (BUI) that provides a sophisticated graphical user interface with unique ease-of-use features. You can also use a command line interface (CLI), which is useful when the Oracle ZFS Storage Appliance cannot be reached through the network or you need to repetitively execute a fixed set of commands. You can access the CLI using the serial console or a secure shell (SSH).

Command-line interaction with the system offers a way to repetitively execute a number of tasks in a fixed sequence as a kind of batch job. When the batch function requires conditional executed code, you must use scripting. The CLI commands can either be entered interactively in the CLI or passed to the CLI in files.

The script language of the Oracle ZFS Storage Appliance is based on JavaScript (ECMAScript version 3) with a few extensions. JavaScript is an interpreted, loosely typed programming language offering object-oriented capabilities. The script interpreter is part of the Oracle ZFS Storage Appliance shell, so the shell interprets and executes the scripts.

Workflows are used to store scripts in the Oracle ZFS Storage Appliance and they also offer user access management for scripts and argument validation functionality. Workflows contain scripts and versioning information, and they can be started in either the BUI or CLI. They can also be started by alert or timer events. Timer events are created from workflow schedules.

This paper primarily provides details on the use of JavaScript programming features within the Oracle ZFS Storage Appliance. It also provides detail on the Oracle ZFS Storage Appliance script architecture, the JavaScript interface into the Oracle ZFS Storage Appliance, and methods for executing scripts. It is beyond the scope of this paper to provide a tutorial for the JavaScript language or CLI command language for the Oracle ZFS Storage Appliance. Familiarity with C, C++, or programming languages such as Perl or Python will help you understand the JavaScript examples provided in this document.

Appendix A contains references to books, online tutorials, blogs, and papers containing detailed information about JavaScript. The Online Help feature within the Oracle ZFS Storage Appliance, which can be accessed through the BUI, provides detailed information about CLI and BUI usage. Also view or download a copy of Oracle's *Sun ZFS Storage 7000 System Administration Guide* (referred to as the administration guide within this document) from one of the Oracle Unified Storage Systems documentation libraries.

NOTE: References to Sun ZFS Storage Appliance, Sun ZFS Storage 7000, and ZFS Storage Appliance all refer to the same family of Oracle ZFS Storage Appliance products. Some cited documentation or screen code may still carry these legacy naming conventions.

Scripting in the Oracle ZFS Storage Appliance

There are three options for interacting with the Oracle ZFS Storage Appliance using the CLI: grouping CLI commands in a file, scripting CLI commands, and using workflows.

A fixed sequence of CLI commands can be grouped in a file and sent to the Oracle ZFS Storage Appliance for execution using SSH. This is a form of batch processing of commands. The commands are executed with the privileges of the user who logs in to the Oracle ZFS Storage Appliance. An extensive desciption of commands available using the Oracle ZFS Storage Appliance shell are available in the administration guide.

When more flexibility is needed, such as executing conditional code using variables and user-defined functions, you can use scripting in the CLI. The shell offers a script interpreter that executes user-defined scripts written in the JavaScript programming language. The scripts are passed to the Oracle ZFS Storage Appliance in the same way batch jobs are passed to it, either through an SSH connection into the Oracle ZFS Storage Appliance or interactively at the CLI prompt.

Workflows, which can encapsulate scripts for later execution, reside in the Oracle ZFS Storage Appliance. You can initiate these workflows using the BUI or CLI. Timer and alert events can also be used to trigger the start of a workflow. You can access the Oracle ZFS Storage Appliance using either its console or an SSH connection.

Scripting Architecture of the Oracle ZFS Storage Appliance

The Oracle ZFS Storage Appliance shell offers the functionality to execute scripts containing JavaScript programming code. Because the JavaScript interpreter is integrated into the shell, it is known as an *application embedded environment*. This is different from the typical Web browser *client-server* use of JavaScript. JavaScript library functions, such as Document Management (DOM), are not available in application-embedded environments. When referring to JavaScript documentation, use the Core JavaScript Reference to find detailed information on available JavaScript library functions.

Since the embedded JavaScript interpreter interprets the code during execution, scripts are executed in the context of the Oracle ZFS Storage Appliance shell and all information retrieved from the Oracle ZFS Storage Appliance, or manipulation of it, is handled by functions passing the required commands to the CLI context. This means that the same CLI navigation commands can be used to walk through the CLI context structure. For instance, the following CLI command navigates to the child context `net`:

```
7000ppc1:> configuration net
7000ppc1:configuration net>
```

The following is the equivalent script command:

```
run ('configuration net');
```

The JavaScript core functions are available from the JavaScript object library. This library contains functions (methods) to manipulate complex object types such as arrays, strings, math on number objects, regexpressions, and more.

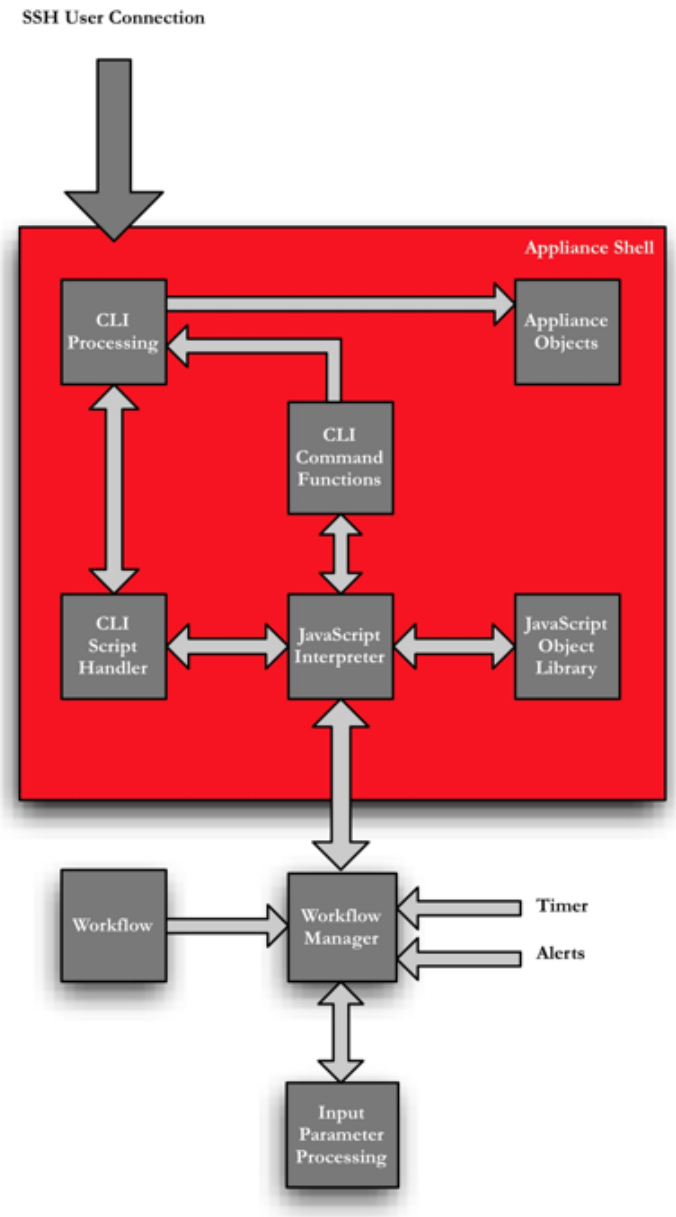

Figure 1. Scripting architecture for the Oracle ZFS Storage Appliance

Scripts stored in the Oracle ZFS Storage Appliance are controlled by the Workflow Manager. The Workflow Manager controls the starting of scripts by users employing the CLI or BUI or by timer or alert events. It also handles the user input dialog, argument verification processing, and start of the execution of scripts in workflows.

Accessing the CLI of the Oracle ZFS Storage Appliance

An SSH connection provides access to the CLI of the Oracle ZFS Storage Appliance. The `show` command provides information about the CLI root context, the available properties, and children contexts. The TAB key lists the available command options in the current context.

```
pBrouwers MacBook-Pro:~ pBrouwer$ ssh root@192.168.56.101
Password:
Last login: Mon Feb 14 16:09:05 2011 from 192.168.56.1
7000ppc1:> show
Properties:
                     showcode = false
                    showstack = false
                  exitcoverage = false
                   showmessage = true
                    asserterrs = false

Children:
                    configuration => Perform configuration actions
                      maintenance => Perform maintenance actions
                              raw => Make raw XML-RPC calls
                        analytics => Manage appliance analytics
                           status => View appliance status
                           shares => Manage shares

7000ppc1:>
analytics      coverage       help           script         status
assert         date           ifconfig       set            time
assertlabels   deny           maintenance    shares         traceroute
configuration  exit           nslookup       shell          tree
confirm        get            ping           show
cover          getent         raw            sleep
7000ppc1:>
```

The `script` command opens the JavaScript interpeter for you to enter script statements. To execute, type a period (**.**) and then press **Enter**.

```
pBrouwers MacBook-Pro:~ pBrouwer$ ssh root@192.168.0.140
Password:
Last login: Mon Feb 14 16:09:05 2011 from 192.168.56.1
7000ppc1:> script
("." to run)> var i=10 ;
("." to run)> printf("i = %d\n",i);
("." to run)> .
i = 10
```

```
7000ppc1:>
```

You can store script statements in a file. The file contents can then be used to send the statements to the Oracle ZFS Storage Appliance using the SSH connection, as shown in Appendix B.

To navigate through the hierarchy of the Oracle ZFS Storage Appliance context structure, specify the name of the context to navigate to, preceded by all the children above it, starting from the current context. The `done` command restores the previous context environment. Use the UNIX `cd` command to move up in the hierarchy or go back to the root context, `cd /`.

The following is the equivalent script program statement:

```
run('cd/');
```

Commands available in a certain child context can be directly executed from the current context by specifying the path to the child context followed by the command.

```
7000ppc1:> configuration net interfaces list
INTERFACE   STATE    CLASS LINKS       ADDRS                LABEL
e1000g0     up       ip    e1000g0     192.168.56.101/24    i_e1000g0
e1000g1     up       ip    e1000g1     192.168.0.147/24     i_e1000g1
7000ppc1:>
```

Accessing the Oracle ZFS Storage Appliance CLI Layer

Scripting is of no use if there are no means to retrieve status information from the Oracle ZFS Storage Appliance and manipulate the configuration information within it. Table 1 shows extensions to the available JavaScript functions that enable you to interact with the Oracle ZFS Storage Appliance.

TABLE 1. EXTENSIONS TO JAVASCRIPT FUNCTIONS

FUNCTION	DESCRIPTION
run	Runs the specified command in the shell, returning any output as a string. Note that if the output contains multiple lines, the returned string will contain embedded newlines.
props	Returns an array of the property names for the current context.
get	Gets the value of the specified property. Note that this function returns the value in native form. For example, dates are returned as Date objects.
set	Takes two string arguments, setting the specified property to the specified value.
list	Returns an array of tokens corresponding to the dynamic children of the current context.
choice	The choices function returns an array of the valid property values for any property for which the set of values is known and enumerable.

These commands are executed in the CLI context structure of the Oracle ZFS Storage Appliance. For more detailed information, see "CLI Scripting" in Chapter 1 of the administration guide.

Scripting Programming Language

The scripting functionality of the Oracle ZFS Storage Appliance is implemented by a JavaScript Language Interpreter build in the CLI layer. The supported JavaScript syntax is based upon the ECMA-3 standard with a few extensions.

Contrary to its name, JavaScript has no relationship to Java. JavaScript is an object-oriented program language that differs from C++ and Java in that it does not have strong type checking. The variable type is defined at runtime, not during compile time. So, variable types are dynamic.

The term *Script* in JavaScript's name might imply that it is a simple, procedural-type programming language, but JavaScript actually contains a rich, object-oriented feature set. Familiarity with object-oriented programming concepts, such as those used by C++, for example, can help you understand the true potential of JavaScript.

For simple tasks, using the traditional procedure-type programming features suffices. For more complex tasks, JavaScript's object-oriented features, such as using objects containing properties and methods to manipulate the properties of objects, is needed. Understanding these object-oriented aspects of JavaScript, including the scope mechanism of variables and the handling of objects and arrays, is essential to dealing with the more complex JavaScript coding.

This document provides basic information on some of the JavaScript concepts to be aware of when you are trying to use JavaScript for more complex tasks. Appendix A contains a resource list for more detailed information about JavaScript.

Note that the JavaScript environment used in the Oracle ZFS Storage Appliance is described in various reference materials as an application-embedded environment. However, material describing the use of client-side JavaScript is not applicable for use in the Oracle ZFS Storage Appliance. Concentrate on the core JavaScript sections.

Syntax

The syntax of the JavaScript language is similar to the C language syntax. It is case sensitive, so it requires consistent names for variables and functions. Semicolons are used to terminate statements and curly braces (`{}`) are used to group blocks of

statements. Although the use of semicolons in JavaScript is in some cases optional, it is best practice to always use them at the end of each statement to maintain easy-to-understand code. C and C++ comment syntaxes are recognized.

Data Types and Variables

In addition to primitive data types, such as numbers, Booleans, and strings, JavaScript supports complex data types in the form of objects. Values of primitive data types are automatically converted to the type needed for the operation.

```
var index=1;
var textarray = new Array('Line 1','Line 2','Line 3');
var array_elements = textarray.length
for ( ; index < array_elements  ; index++) {
printf('Array element' + index + ': ' + textarray[index] + '\n');
}
```

Functions are a special type of object. This means that functions can be stored in variables, arrays, and objects. Functions can also be passed as arguments to other functions.

As in other languages, JavaScript uses the value `null` to indicate that a variable does not hold a valid value. JavaScript never sets a value to `null`; this has to be done explicitly by the programmer. JavaScript uses the value `undefined` to identify a variable that has been declared but has never had a value assigned to it. In this case, the type of the variable is not known yet. The value `undefined` is also used to identify a reference to a property of an object that does not exist yet.

Variables in JavaScript can be referenced by value or by reference. Strings are a special case. In JavaScript, the contents of a string are immutable; they can never be changed. A string can be changed only by creating a new string and then copying the parts of the original string that should stay the same.

Always use the `var` statement to declare a variable, because it fixes the scope of the variable at the point of the declaration in the code. Not using the `var` statement automatically makes the variable global, which could have unexpected side effects. For instance, a global variable will not be disposed of by the JavaScript garbage collection mechanism, which might lead to memory leaks.

JavaScript Properties and Variables

The JavaScript language supports both variables and properties. At first glance, they look the same. Both are used to hold a value. Their differences lie in the way the JavaScript

interpreter creates them during runtime. Basically, properties belong to objects and variables belong to contexts. For the average user, there is no real difference.

In the JavaScript language, properties are used to add variable information about an object, for example, `traffic-light.current-color=red`. Functions are treated as objects too, so variables defined within a function are seen as properties of that function.

It is important to understand the different methods and syntax for allocating a value to a property. There are two methods, by value or by reference, and for each method a few different syntaxes can be used. For ease of reading code and maintaining it, use the "by reference" method for variables that hold references to functions, objects, and text strings.

Define text strings at the top of the program, so they can be easily found when doing maintenance on the code.

In the following example, the values of the properties of the workflow structure are defined at the top of the code. You will notice later that the workflow structure has to be declared at the far end of the code. The example shows the syntax used for both the "by reference" and "by value" methods.

Example 1. Different Methods for Initializing Properties of the Object `workflow`

```
/// File: Example 1
// Object initialized with two properties, using the object literal syntax,
// in which each property name/value pair is followed by a comma.
// The property name is followed by colon.
var MyWorkflow = {
       MyVersion:          '1.0',
       MyName:             'Example 1',
       MyDescription:      'Example showing basic Object Workflow structure'
}

// New property added using variable assignment syntax.
MyWorkflow.MyDescription =       'Use of properties example';

// Workflow object initialized using literal syntax and references.
// Values do not need to be literals; they can be references to other
objects,
// as can be seen here.
var workflow = {
       name:        MyWorkflow.MyName,             // Reference syntax
       description: MyWorkflow.MyDescription,      // Reference syntax
       version:     MyWorkflow.MyVersion,                 // Reference syntax
```

```
origin:      'Oracle',                              // Literal syntax
execute:     function () { return('Hello World'); }// Literal Syntax
```

JavaScript Operators

JavaScript uses all the familiar operators from other languages. Additionally, a new type of operator is used to test "identity": `===` and `!==`. These operators differ from the `==` and `!=` operators in that they do not cause automatic typecasting the way the `==` and `!=` operators do. Note the details on these operators, because the differences are sometimes subtle.

```
var num = 10;
var num_string= '10';

if ( num==num_string )
    print('True because values are the same\n');
if ( num===num_string)
    print('Variables are of identical type and have the same value\n');
else
    print('Variables do not have same value OR are not of identical type\n');
```

The code in Example 1 results in the following output:

```
True because values are the same
Variables do not have same value OR are not of identical type
```

Note that all arithmetic operators can be used in combination with the assignment operator, for example, `*=`, `+=`, and `%=`. Read these statements as *a operator = b* and as *a = a operator b*.

JavaScript Statements

If you are familiar with UNIX shell, C, or C++ programming, it is easy to write programs in JavaScript. All the familiar statements from other languages, such as `if then`, `while`, `for`, and `switch case` statements are present in the JavaScript language.

It is good programming practice to catch runtime exceptions that might occur during the execution of your program. Exception events are triggered by events such as a failing function or selecting a share that does not exist with the command `run ('select myshare')`. The `throw` statement forces an explicit exception signal.

Catching runtime exceptions and recovering from them is handled with the JavaScript expression `try/catch/finally` statements.

The following simple cluster configuration test script checks whether the current Oracle ZFS Storage Appliance node is part of a cluster:

Example 2. Simple Cluster Configuration Test Script

```
7000ppc1:> script
("." to run)> function ClusterTest(){
("." to run)>    try {
("." to run)>            run ('cd /');
("." to run)>            run ('configuration cluster');
("." to run)>            return(true);
("." to run)>            }
("." to run)>    catch (err) {
("." to run)>            if (err=EAKSH_BADCMD) {
("." to run)>                    return(false);
("." to run)>            }
("." to run)>            else {                          // catch unknown
condition
("." to run)>                    throw("Unexpected cluster test error");
("." to run)>            }
("." to run)>    }
("." to run)> }
("." to run)> // Main start
("." to run)> printf("Let's see if this node is part of a cluster; ");
("." to run)> if ( ClusterTest() )
("." to run)>    printf("Yes it is\n");
("." to run)> else
("." to run)>    printf("No it is not\n");
("." to run)> .
Let's see if this node is part of a cluster; No it is not
```

The function `ClusterTest` uses the `try/catch` construction to execute the `run` command to navigate to the child context cluster. If the `run` command fails, it is caught by the `catch` statement, which is where you check to see whether the `run` command failed. Any other unexpected error condition terminates the program using the `throw` statement.

Executing Scripts in the Oracle ZFS Storage Appliance

The following code shows the commands used to load and execute scripts in the Oracle ZFS Storage Appliance. Remember that scripts are passed on to the script interpreter in the Oracle ZFS Storage Appliance shell.

```
pBrouwers MacBook-Pro:~ pBrouwer$ ssh root@192.168.0.140
Password:
Last login: Mon Feb 14 16:09:05 2011 from 192.168.56.1
7000ppc1:> script
("." to run)> var i=10 ;
("." to run)> printf("i = %d\n",i);
("." to run)> .
i = 10
7000ppc1:>
```

In the preceding example, script statements are entered after activating the script interpreter with the command `script`. The statements are executed after typing a period (.) and then pressing Enter. This method is fine for simple interactive use.

When dealing with more complex scripts, it is easier to group the commands in a text file and send the file over to the Oracle ZFS Storage Appliance using an SSH connection. This environment provides full JavaScript functionality, such as the use of functions and conditional statements.

The following example script uses a simplified version of a script to create and delete a share in an existing pool and project.

Example 3. Simple Script for Creating and Deleting a Share

```
// File: Example3.txt
script
// For ease of use, group all our arguments in one object.
// One could even use a shell script to create this JavaScript from a
template
// using the arguments passed on to the user in a shell script.
// Version:  1.1.3
var MyArguments = {
       pool:         'poola',
       project:      'projecta',
       share:        'volumeTest',
       what:         'delete'
}
```

```
function CreateDeleteShare (Arg) {
	run('cd /');		// Make sure we are at root child context level
	run('shares');
	try {
		run('set pool=' + Arg.pool);
	} catch (err) {
		printf("Specified pool, %s not found\n ",Arg.pool);
		return;
	}
	try  {
		run('select ' + Arg.project);
	} catch (err) {
		printf("Specified project, %s not found\n ",Arg.project);
		return;
	}
	if (Arg.what=='create' ) {
		try {
			run('filesystem ' + Arg.share);
			run('commit');
		} catch(err) {
			printf("Unable to create share, %s\n",Arg.share);
			return;
		}
		printf('Successfully created share, '+Arg.share);
		return;
	} else {
		try {
			run('select' + Arg.share);	// Check if share is
there
			run('done');			// Release for delete
			run('confirm destroy ' + Arg.share);
		} catch (err) {
			if (err.code == 10004 )
				printf("Specified share, %s, does not
exist\n",Arg.share);
			else   printf("Unable to delete share, %s\n",Arg.share);
			return;
		}
		printf("Successfully deleted share, %s\n", Arg.share);
	}
}
```

```
// Kick off the create delete function using our object MyArguments to pass
on the
// parameters needed for the job.
printf("About to %s share, %s from project, %s in pool %s\n",
             MyArguments.what,
             MyArguments.share,
             MyArguments.project,
             MyArguments.pool);
CreateDeleteShare(MyArguments);
```

The script uses an object, `MyArguments`, to hold the names of the pool, project, and share to use. This structure keeps all the variable elements at the top of the script instead of hard-coding them throughout the script. Runtime errors are handled using the `try/catch` construction. For simplicity, no distinction is made between the types of errors caught. The variable `err.code` can be used to examine what type of error caused the `run` command to fail.

Scripts used in this way are ideal for a batch-type environment, where predefined tasks need to be executed. Workflows are a better choice when tighter control of script use is required. Scripts used in workflows are stored in the Oracle ZFS Storage Appliance and cannot be modified once they are loaded into it, because their code is not visible anymore to the end user.

Access control can also be applied to workflows, enabling restrictive use, for example, to those created for administrative tasks. Workflows can prompt users for input, as you will see in the next section.

Apart from user input prompting, or start triggers by the Oracle ZFS Storage Appliance timer or alert events, scripting with SSH presents no other restrictions compared to using workflows.

Using Oracle ZFS Storage Appliance Workflows

Previous examples illustrated loading scripts into the Oracle ZFS Storage Appliance using the SSH connection or interactively at the console CLI. To permanently store a script in the Oracle ZFS Storage Appliance, you must put the script under the Workflow Manager's control. Workflows are executed asynchronously in the Oracle ZFS Storage Appliance shell with the user credentials that are used to log in to the system.

In order for the Workflow Manager to store and execute a script, the script requires some additional information. The Workflow Manager uses this to present information to the user and to gain access to the workflow start function. This information is stored in the object named `workflow` and it contains the properties shown in Table 2.

TABLE 2. OBJECT WORKFLOW PROPERTY MEMBERS

REQUIRED PROPERTIES	JAVASCRIPT TYPE	DESCRIPTION
`name`	String	Name of the workflow
`description`	String	Short description of the workflow
`execute`	Function	Script code to execute
OPTIONAL PROPERTIES		
`version`	String	Version of this workflow, in dotted decimal (major.minor.micro) form
`required`	String	Minimum version of the Oracle ZFS Storage Appliance software required for this workflow to run
`origin`	String	Workflow provider's name
`parameters`	Object	Structure defining script input parameters
`validate`	Function	JavaScript function that validates input parameters

The creation of the object `workflow` follows the JavaScript object literal syntax: a comma-separated list of colon-separated property/value pairs enclosed in curly braces.

In the following code, <`property`> is one of the names of the properties mentioned in Table 2.

```
var workflow = {
       <property>:   <object literal>|<object reference>,
       <property>:   <object literal>|<object reference>,
};
```

The value in <object literal> is either of type string, function, or object, as mentioned in Table 2.

A minimal workflow script would look as follows:

```
// Example of basic workflow definition using object literals
// in the workflow object constructor.
var workflow = {
       name:         'Minimum workflow code',
       description:  'Example of basic workflow structure',
       execute:      function() { return('Hello World'); }
};
```

Instead of directly specifying the data value for a property, a reference to an earlier defined object or variable can be used inside the object workflow. The following example shows the minimal workflow definition.

Example 4. Code for Defining a Minimal Workflow

```
// File: Example4.txt
// Example of basic workflow definition using variables
// in the workflow object's constructor
//
//
var WorkflowName = 'Minimum workflow code';
var WorkflowDescription = 'Example of basic workflow structure';

function Main () {
       return('Hello World');
}
var workflow = {
       name:         WorkflowName,
       description:  WorkflowDescription,
       execute:      Main
};
```

To load the preceding workflow example in the Oracle ZFS Storage Appliance, you can use either the BUI or the CLI upload command in the "maintenance workflow" child context of the Oracle ZFS Storage Appliance.

The following screenshots show the BUI workflow load steps.

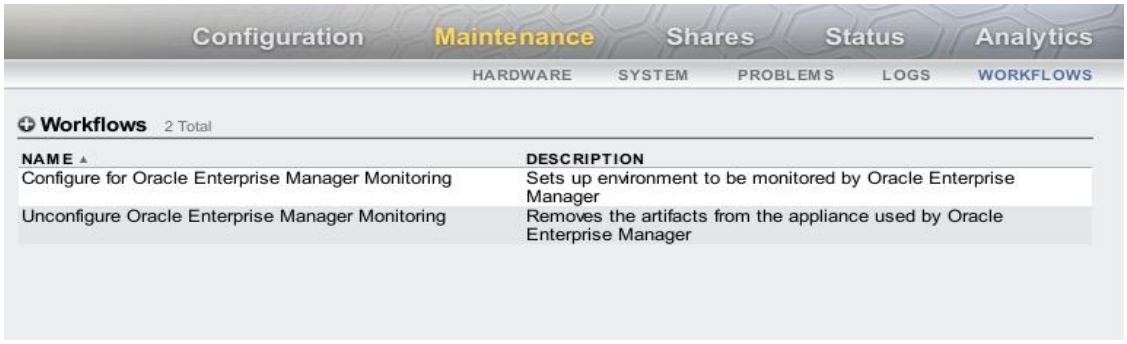

Figure 2. Workflows in the Oracle ZFS Storage Appliance BUI

Use the + button to bring up the load workflow file dialog box.

Figure 3. BUI displaying the Add Workflow window

Syntax errors in the workflow file are checked during the load process.

Once loaded, the workflow is shown with the name and description info as set in the `workflow` object's name and description properties.

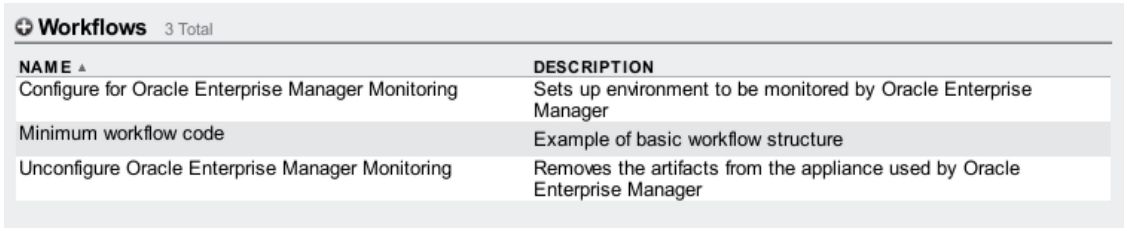

Figure 4. BUI showing newly added workflow

To execute the workflow, double-click it.

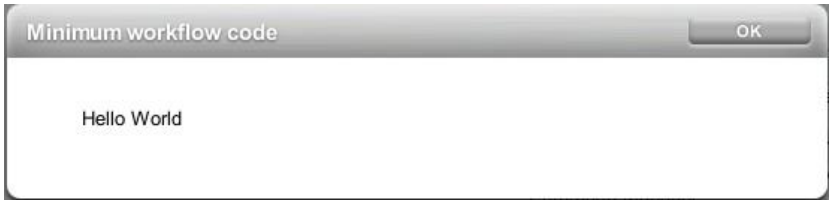

Figure 5. New workflow output in the BUI

Adding some input parameters to the workflow using the workflow property `parameters` makes the workflow more useful.

```
parameters = {
       <parameter1name>: {
              label:        <String>
              type:         <String>
       }
<parameter2name>: {
              label:        <String>
              type:         <String>
              options:      <Array>
              optionlabels: <Array>
       }
<parameterNname>: {
              label:        <String>
              type:         <String>
              optional:     <Boolean>
       }
};
```

This in itself is an object with the following structure:

- The property `parameterNname` is the name of the input parameter by which it can be referred to in the script code.
- The property `parameterNname` is itself an object and always needs to contain the `label` and `type` properties.
- The value of the property `label` is used in the code in Example 5.
- The value of the property `type` specifies the type of the value stored in the `paramenterNname` object, such as Boolean, string, or file.

Note: A complete list of the `type` definitions can be found in the administration guide of the Oracle ZFS Storage Appliance.

The following properties are not required, but you can use them to specify requirements on a parameter:

- The property `optional` – When set to `true`, specifies that no input is required in the UI for this parameter.

- The property pair `options-optionlabels` – Presents a fixed list of input values. To use this function, you must set the property `type` to the value `ChooseOne`.

Next, you will see how the Workflow Manager uses the properties in an object when a workflow is started. The example shows the BUI interface. For the CLI interaction, see the administration guide. The processing mechanism used by the Workflow Manager is the same for both the BUI and the CLI.

The object `parameters` is used to build a dialog box that contains fields into which you can enter the input required by the object `parameters`. When you fill in the fields and click PROCEED, the Workflow Manager executes the validation function, as specified in the property `validate`, and uses the reference to the object `parameters` as an argument. When an error is signaled by the `validate` function, the Workflow Manager brings back the dialog box indicating in which field an error was detected.

Once the function `validate` is passed correctly, the Workflow Manager calls the function specified in the workflow property `execute`, again with a reference to the object `parameters` as an argument.

Example 5 takes the create/delete share script example and adapts it for use in a workflow. The Workflow Manager prompts for a pool and project name for which the share create/delete operation takes place. The choice between delete/create is presented using a pull-down list construction.

Example 5. Basic Workflow Script for Creating and Deleting a Share

```
// Simple example of how to create/delete a share.
// File example5.txt
// Information to be used in workflow object:
var MyWorkflow = {
       MyVersion:          '1.0.0',
       MyName:             'Create/Delete a share',
       MyDescription:      'Example of how to create/delete a share',
       Origin:             'Oracle Corporation',
       err: {              // Definition of error codes.
                           // Define a range for your project that
                           // can be recognized by the sysadmins in
                           // your org.
              WP_SCRIPT_WORKFLOWS_CREATE_SHARE:  8001,
              WP_SCRIPT_WORKFLOWS_DELETE_SHARE:  8002,
       }
}

// This example workflow uses four input parameters.
```

```
// The last parameter is an example of the use of a fixed list of values.
var MyParams = {
	pool:  {		// Pool to create/delete the share.
		label:		'Pool Name',
		type:		'String',
	},
	project: {		// Project to create/delete the share.
		label:		'Project name',
		type:		'String',
	},
	share: {		// Share to create/delete.
		label:		'Share name',
		type:		'String',
	},

	what:  {		// Create or delete the share.
		label:		'Operation',
		type:		'ChooseOne',
		options:	['create','delete'],
		optionlabels:	['Create','Delete'],
	}
}

// Verify function from workflow.
// Check whether pool and project exist.
function VerifyPoolandProject(p) {
	var err_msg = ' does not exist';
	run ('cd /');		// Make sure we are at root child context level.
	run ('shares');
	try {			// Check whether pool name exists.
		run('set pool='+p.pool);
	} catch(err) {
		return( {pool: 'Specified pool, ' + p.pool + err_msg } );
	}
	try {
		run('select ' + p.project);
	} catch(err) {
		return( {project: 'Specified project, ' + p.project + err_msg }
);
	}
```

```
	if (p.what=='delete' ) {   // Check whether the share to be deleted
exists.
		try {
			run('select'+ p.share);
		}
		catch(err) {
			return({share: 'Specified share, ' + p.share + err_msg
});
		}
	}
	return;
}
function CreateDeleteShare (p) {
	run('cd /');		// Make sure we are at root child context level.
	run('shares');
	run('set pool=' + p.pool);
	run('select ' + p.project);
	if (p.what=='create' ) {
		try {
			run('filesystem ' + p.share);
			run('commit');
		} catch(err) {
			throw {
				code:
MyWorkflow.err.WP_SCRIPT_WORKFLOWS_CREATE_SHARE,
				message: 'Unable to create ' +
					p.share + ',' + err.message
			}
		}
		return('Successfully created share, '+p.share);
	} else {
		try {
			run('confirm destroy ' + p.share);
		} catch(err) {
			throw {
				code:
MyWorkflow.err.WP_SCRIPT_WORKFLOWS_DELETE_SHARE,
				message: 'Unable to delete ' +
					p.share + ',' + err.message
			}
		}
```

```
                return('Successfully deleted share, '+p.share);
        }
}

var workflow = {
        name:           MyWorkflow.MyName,
        description:    MyWorkflow.MyDescription,
        version:        MyWorkflow.MyVersion,
        origin:         MyWorkflow.Origin,
        parameters:     MyParams,
        validate:       VerifyPoolandProject,
        execute:        CreateDeleteShare
};
```

Notice that the object `workflow` is specified at the end of the code, since all objects and functions referenced in that object must be defined first. For ease of readability and manageability, all the script information is held in an object and is specified at the top of the code. References to elements of this object are later used in the object `workflow`. The same is done for the properties `validate` and `execute` in the workflow. This makes the workflow script easier to read.

The object `MyParams` specifies the input information requested from the user for this workflow. Four parameters are requested: three are of type text and one is a list with two items, as shown in Figure 6. The reference to the object `MyParams` is stored in the property `parameters` in the object `workflow`.

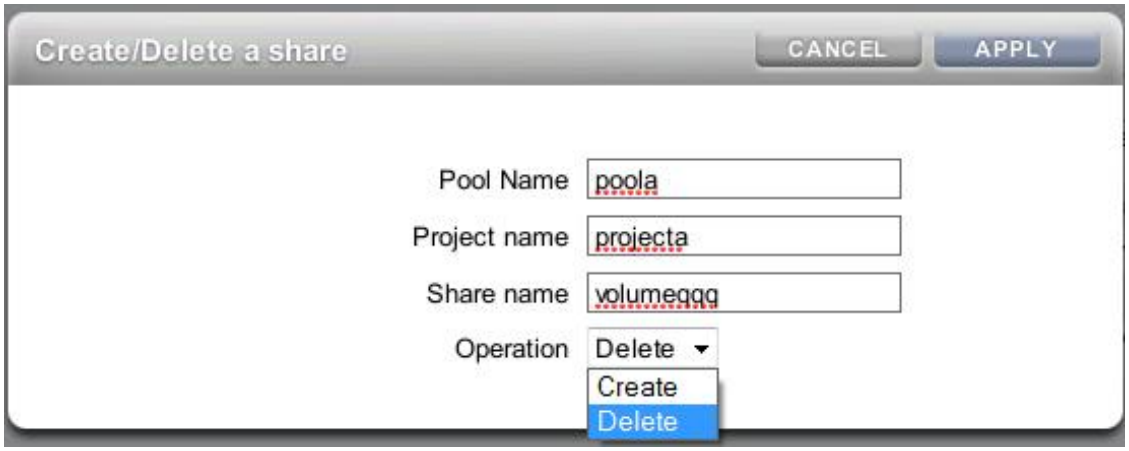

Figure 6. Example of user dialog box

Executing the workflow in the BUI brings up the dialog box shown in Figure 7.

In this example, volumeqqq does not exist. Clicking APPLY triggers the share select error in the validate function `ValidatePoolandProject` in the object `workflow`, as shown in Figure 7.

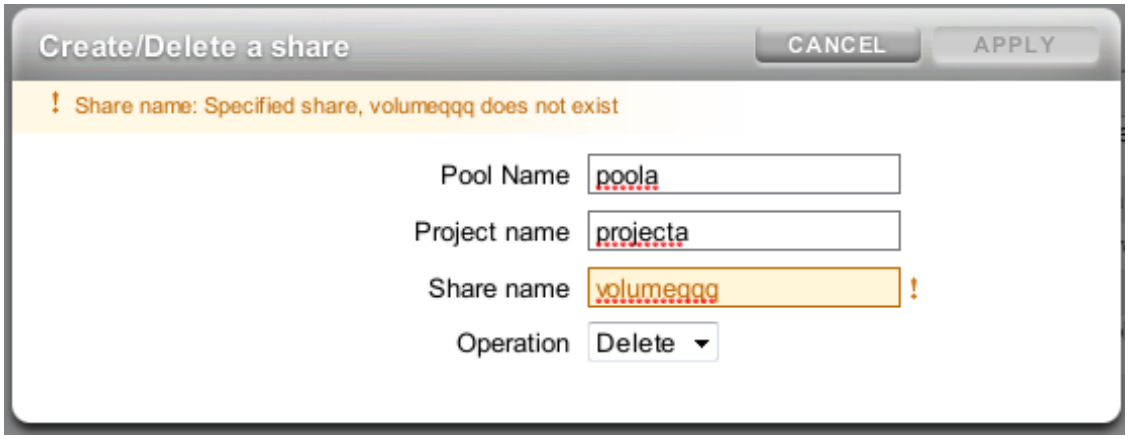

Figure 7. Example of user dialog box input error

The error is the result of the following statement:

```
return({share: 'Specified share, ' + p.share + err_msg });
```

The return statement `share:` contains a reference to the property name of the input parameter for which an error is flagged. As a result, the Workflow Manager highlights the input field value that triggered the error and displays the error message preceded by the value of the `label` property of the field definition in the object `MyParams`.

The error is caught using the JavaScript `try/catch` functionality.

Alert Workflows

You can also use workflows to implement a custom function or action in response to an Oracle ZFS Storage Appliance-generated alert. The Oracle ZFS Storage Appliance can generate alerts for a variety of situations. Alert messages are stored under Maintenance → Logs → Alerts. By defining an alert action, you can bind a workflow to an alert. Workflows triggered by alert actions run in the background and do not prompt for user input. The “Configuration” section of the administration guide provides detailed information about alerts, how they are customized, and where alert logs can be found.

In order to tie workflows to events, you must add new properties to the object `workflow`. Both the `alert` property and `setid` property must be set to `true` to enable the workflow to run with the privileges of the owner of the workflow file (as set up in the `owners` role). Since no user input is required, the object describing the input parameters is not needed.

The code in Example 6 adapts the previous example to provide a very simple alert action workflow.

Example 6. Minimum Code for Alert Workflow

```
// File: Example 6
// Example 5 adapted for alert usage
var MyWorkflow = {
       MyVersion:     '1.0',
       MyName:        'Example 6',
       MyDescription: 'Example of use of Alert',
       Origin:        'Oracle'
};

var workflow = {
       name:          MyWorkflow.MyName,
       description:   MyWorkflow.MyDescription,
       version:       MyWorkflow.MyVersion,
       alert:         true,                    // Workflow triggered by alert
       setid:         true,
       origin:        MyWorkflow.Origin,
       execute:       function (MyAlert) {
                           audit('workflow started for alert'+MyAlert.uuid);
                            }
};
```

For the workflow to send information to the outside world, the function `audit` must be used. This function takes a single string as argument, and the text is placed in the audit log of the Oracle ZFS Storage Appliance. The Workflow Manager passes an object to the function defined for the property `execute`. This object contains the elements shown in Table 3.

TABLE 3. ELEMENTS IN THE EXECUTE PROPERTY

PROPERTY	TYPE	DESCRIPTION
`class`	String	The class of the alert
`code`	String	The code for the alert
`uuid`	String	The alert's unique identifier
`timestamp`	Date	Event time
`items`	Object	Object with more detailed information on the event

The property `items` is an object that contains the following detailed information on the event that triggered the workflow.

TABLE 4. EVENT INFORMATION

PROPERTY	TYPE	DESCRIPTION
`url`	String	URL to Web page containing a description of the event
`action`	String	The action that should be taken by the user in response to the event
`impact`	String	The impact of the event that precipitated the alert
`description`	String	A human-readable string describing the alert
`severity`	String	The severity of the event that precipitated the alert
`response`	String	Automated response action information
`type`	String	Type of alert, for example, minor or major

The information is also shown in the BUI in response to a request for detailed information about an alert, as shown in Figure 8.

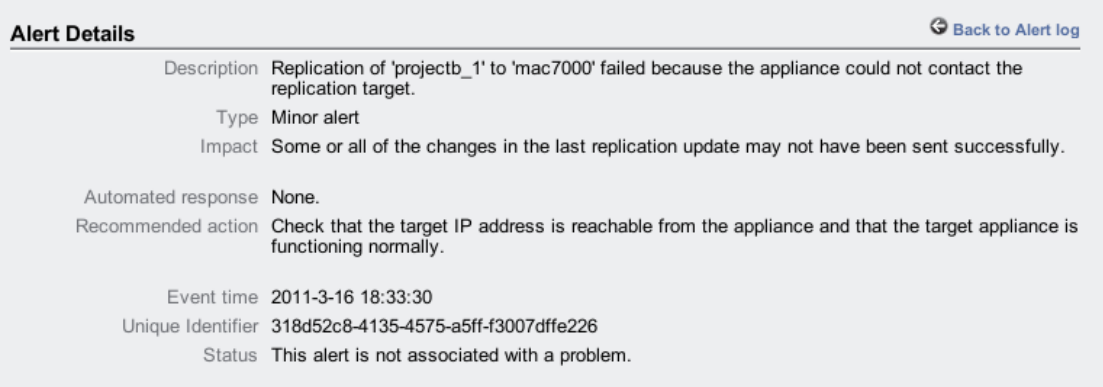

Figure 8. Detailed alert information in the Oracle ZFS Storage Appliance BUI

To bind the example code to, for instance, a replication event in the Oracle ZFS Storage Appliance, you must first load the workflow into the Oracle ZFS Storage Appliance. The screenshot in Figure 9 shows the dialog box in which you must define an alert action in the configuration context. The alert action must be bound to an event category. For each category, you can select a subset event type.

You can configure more than one alert action. If you select Execute workflow, the previously loaded example script would be executed.

In this example, two actions are set to execute when a replication action fails: sending an e-mail and starting a workflow. Note the TEST option for triggering the configured action manually.

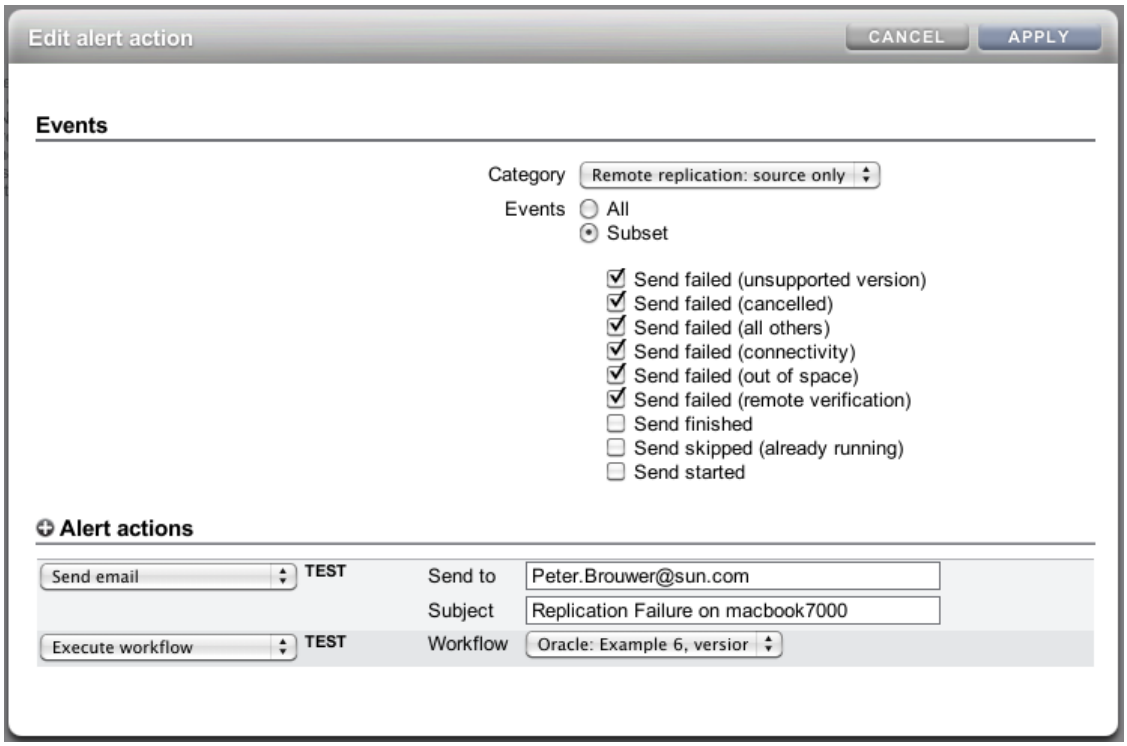

Figure 9. Adding an alert action in the Oracle ZFS Storage Appliance BUI

Scheduled Workflows

The start of workflows can be managed by setting up schedules for them. The workflow manager creates timer events from the schedules to start a workflow at a specific time.

Schedules can be created using the CLI for an existing workflow or incorporated in the workflow by defining an array type object named schedule containing one or more entries using the structure as shown in table 5.

TABLE 5. WORKFLOW SCHEDULE OBJECT PROPERTIES

PROPERTY	TYPE	DESCRIPTION
`offset`	Number	Determines the starting point in the defined period. Zero start point is Thursday.
`period`	Number	Defines the frequency of the Schedule
`unit`	String	Specifies if either seconds or month are used as unit in the offset and period

Example 7. Workflow Schedule definition example

```
    // File: Example 7
    var MyWorkflow = {
            MyVersion:        '1.0',
            MyName:   'Example 7',
            MyDescription: 'Example of use of Schedules',
            Origin:   'Oracle'
    };
    var MySchedules = [
            // use the inline arithmetic to create readable code
            // half hr interval
            // Starting day seems to be Thursday for the offset.
            { offset: 0, period: 1800, units: "seconds" },
            // every Monday on 10:15
            { offset: 4*24*60*60+10*60*60+15*60, period: 7*24*60*60, units:
    "seconds"}
    ];
    var workflow = {
            name:         MyWorkflow.MyName,
            description:  MyWorkflow.MyDescription,
            version:      MyWorkflow.MyVersion,
            origin:       MyWorkflow.Origin,
            alert:        false,
            setid:        true,
            schedules:    MySchedules,
            scheduled:    true,
            execute:      function () {
                            audit ('Example 7: started via scheduled event');
                          }
};
```

After uploading the workflow code in the Oracle ZFS Storage Appliance, we can use its CLI to verify the workflow schedule ;

```
Node1:> maintenance workflows
Node1:maintenance workflows> select workflow-007
Node1:maintenance workflow-007> schedules
Node1:maintenance workflow-007 schedules> show
Schedules:

NAME                 FREQUENCY            DAY                  HH:MM
```

```
schedule-000        halfhour            -                   --:00
schedule-001        week                Monday              10:15
```

Applying Best Practices to Scripting

The following section provides recommendations for developing and implementing scripts.

Best Practices for Coding Style

Since the programming structure of JavaScript resembles the structure of the programming languages C and C++, code-writing standard practices for C or C++ apply to JavaScript code, for example, the use of curly braces and indents, as seen in this document.

Use variable names that make sense and are easily understood. Do not use variables such as `i`, `n`, or `x`. Keep variable names short and concise, but descriptive. A variable name such as `FC_Lun` or `iSCSI_Lun` makes more sense than `LUN`.

Use the English language when using a program language. It makes it easier to share workflow scripts and request input or support from non-local language speaking colleagues.

Use simple and well-structured code statements, and add comments. Comments should add information, not repeat the existing information. Well-written comments should help you return to a piece of code after a year to make some updates. When workflows are going to be shared, their use and purpose should be clear from the embedded comments.

JavaScript Best Practices

Use the following JavaScript best practices:

- Use semicolons. In general, each statement in JavaScript is terminated by a semicolon. However, JavaScript allows some freedom and tries to make assumptions if semicolons are not present in certain situations. Also the Workflow Manager does some checking during the load phase of a workflow into the Oracle ZFS Storage Appliance. To avoid any ambiguity, always use a semicolon at the end of each statement.
- Avoid cluttering the global namespace. Using global variables always presents the risk of name conflicts or resolving references at the wrong level in the scope chain. Use objects to encapsulate variables; see the object `MyWorkflow` in Example 5. Using objects also helps create portable code that is easy to maintain.
- Use the `var` statement to define a variable before using it. When a variable has not been defined, JavaScript will use the variable as defined with a global scope. As a result, a script can continue to increase memory usage because the undefined (now global) variable is not included in the JavaScript garbage collection mechanism. This phenomenon is known as memory leak.

- Avoid the `with` statement. The `with` statement is often used to save typing when dealing with deeply nested object hierarchies. However, there is no control over how variables are resolved when JavaScript traverses up the hierarchy chain. Use a variable that holds a reference to the object that would have been used within the `with` statement.

Training Best Practices

The following are recommended training best practices:

- Read and study. Use the wealth of information about JavaScript available on the internet in the form of blogs and articles. But do not underestimate the value of information on real paper. Nothing substitutes for sitting in a comfortable chair with a cup of coffee reading a book.
- Last but not least, simulate. The Oracle ZFS Storage Appliance Simulator is an excellent tool for getting familiar with the scripting and workflow environment. The simulator supports all script and workflow functionality.

Tips and Examples for Client-Side Appliance Control

This section provides some simple examples that might trigger you to come up with an elegant way of solving your problem. The examples are by no means foolproof solutions. For simplicity, error checking and exceptions are not dealt with in the following code examples.

Automatically Executing CLI Scripts Using SSH

Avoid the password prompt each time a CLI script is executed using SSH by installing on the Oracle ZFS Storage Appliance a public key that has been generated with the `ssh-keygen -t rsa -b 1024` command on the host from which the scripts are executed. Setup scripts can be executed using the following syntax for `ssh`, where <`key_rsa`> is the name of the file containing the key generated with the `ssh-keygen` command.

```
ssh -i .ssh/<key_rsa> root@MyAppliance
```

Using CLI Scripting with UNIX Shell

Sending a sequence of commands to the Oracle ZFS Storage Appliance when an SSH connection is used for access can sometimes get confusing. To understand the options available, refer back to the basics. Like any UNIX-type command interface, SSH has two I/O mechanisms: `stdin` and `stdout`. Basically, `stdin` is a communication interface into SSH and `stdout` is a communication interface out of SSH. The input and output can be redirected.

```
ssh root@myAppliance < inputfile > outputfile
```

In the preceding command, the input is redirected for `ssh` to read the characters from file `inputfile` and the output from `ssh` is redirected to file `outputfile`. So if `inputfile` looks like the following, characters in `inputfile` are sent to the CLI of the Oracle ZFS Storage Appliance.

```
configuration net interfaces list
```

The output of the command sequence is picked up by `ssh` and redirected to `outputfile`.

At the CLI prompt of the Oracle ZFS Storage Appliance, you can interactively enter JavaScript commands and execute them. This means that you can write JavaScript commands in file `inputfile`, send them to the Oracle ZFS Storage Appliance using `ssh`, and capture the output of the JavaScript commands in `outputfile`. This is quite powerful, because it means you can write "intelligent" batch jobs. This capability merges the `batch` command environment with the intelligent dynamic coding environment.

Now that you understand the `ssh` mechanism, you can write scripts that are sent to the Oracle ZFS Storage Appliance using `ssh`, and you can check on the exit status of the scripts on the client side.

So, for example, if you have a script that creates a share, and you want to check on a failure of that script, you could return the error code back to SSH (the client shell) and react to the error situation in the client shell code.

Example 8 uses the old create/delete share script from Example 3. First, you must substitute all the code that emits text messages with code that returns numeric codes. Text strings are too difficult to process in the shell. Then you must add additional shell coding to handle the shell argument processing and initiate the shell variables to be used to pass the variables to the Oracle ZFS Storage Appliance scripting part.

Note that the text between the two "EOF" strings in the file is the bit of JavaScript code that is sent to the Oracle ZFS Storage Appliance using `ssh stdin`. Error codes from the JavaScript part are passed back using the `print` command at the end of the JavaScript code and captured in the shell script's `ScriptError` variable.

Example 8. Shell Script to Pass Arguments to Oracle ZFS Storage Appliance Script

```
#!/bin/sh
# File example8.txt
# Shell script using input arguments to be passed to appliance script job.
# Script error codes are passed back to the shell.
Usage() {
   echo "$1 -u <Appliance user> -h <appliance> -s <share> -c|-d -j <project>
-p <pool>"
   exit 1
}
# Error code definitions
PoolNotFound=100
ProjectNotFound=101
CreateShareFailed=102
DeleteShareFailed=103
UnKnownError=999
#
# Shell script main
PROG=$0
# Check used command line options
while getopts u:h:s:j:p:cd flag
do
```

```
        case "$flag" in
        c)      create="true"; action="create";;
        d)      delete="true"; action="delete";;
        p)      pool="$OPTARG";;
        j)      project="$OPTARG";;
        s)      share="$OPTARG";;
        u)      user="$OPTARG";;
        h)      appliance="$OPTARG";;
        \?)     Usage $PROG ;;
        esac
done
# Create and Delete action are multi-exclusive
[ "$create" = "true" -a "$delete" = "true" ] && Usage $PROG
# None of the arguments can be empty
[ -z "$pool" -o -z "$project" -o -z "$share" -o -z "$appliance" -o -z "$user"
] && Usage $PROG
#
# Now get to the job at hand
# Start ssh and feed the script code in using stdin
ScriptError=`ssh $user@$appliance << EOF
script
// Above command activates script mode in the appliance.
// For ease of use, group all arguments in one object.
// Note we use the shell variables obtained from the shell command line.
// Version:  1.0.0
var MyArguments = {
        pool:               '$pool',
        project:            '$project',
        share:              '$share',
        what:               '$action'
}
// We could use the script variables throughout the scripting code but it
might create
// confusion, so keep all the shell-script variable interaction concentrated
in one
// place in the script.
var MyErrors = {
        PoolNotFound:       '$PoolNotFound',
        ProjectNotFound:    '$ProjectNotFound',
        CreateShareFailed:  '$CreateShareFailed',
        DeleteShareFailed:  '$DeleteShareFailed',
```

```
	UnKnownError:	'$UnknownError',
}
function CreateDeleteShare (Arg) {
	run('cd /');		// Make sure we are at root child context level
	run('shares');
	try {
		run('set pool=' + Arg.pool);
	} catch (err) {
		return(MyErrors.PoolNotFound);
	}
	try  {
		run('select ' + Arg.project);
	} catch (err) {
		return(MyErrors.ProjectNotFound);
	}
	if (Arg.what=='create' ) {
		try {
			run('filesystem ' + Arg.share);
			run('commit');
		} catch(err) {
			return(MyErrors.CreateShareFailed);
		}
		return(0);
	} else {
		try {
			run('select' + Arg.share);	// Check if share is
there
			run('done');			// Release for delete
			run('confirm destroy ' + Arg.share);
		} catch (err) {
			if (err.code == 10004 )
				return(MyErrors.DeleteShareFailed);
			else return(MyErrors.UnKnownError);
		}
		return(0);
	}
}
// Kick off the create delete function using our MyArguments object to pass
on the
```

```
// parameters needed for the job.
err=CreateDeleteShare(MyArguments);
// The devil is in the tail; return the error code to stdout of ssh so the
shell can
// pick it up in the ScriptError variable.
print(err);
.
EOF`
echo $ScriptError

[ "$ScriptError" != "0" ] && {
   case $ScriptError in
       $PoolNotFound)      Message="Specified pool : $pool, not found";;
       $ProjectNotFound)   Message="Specified project : $project, not found";;
       $CreateShareFailed) Message="Share $share could not be created";;
       $DeleteShareFailed) Message="Share $share could not be deleted";;
       $UnknownError)      Message="Unexpected script error";;
   esac
       echo $Message
       exit 1

}
echo "$action of share: share in project: $project, pool: $pool, was
successful"
```

Note that the shell variable could have been used throughout the JavaScript part, but this would make the JavaScript more difficult to maintain. For ease of use, all the shell variables used are concentrated at the start of the JavaScript part. Also note that the shell substitutes the shell variables in the CLI script before the script is sent to the Oracle ZFS Storage Appliance.

Appendix A: References

NOTE: References to Sun ZFS Storage Appliance, Sun ZFS Storage 7000, and ZFS Storage Appliance all refer to the same family of Oracle ZFS Storage Appliance products. Some cited documentation or screen code may still carry these legacy naming conventions.

- *Oracle ZFS Storage Appliance System Administration Guide*: http://www.oracle.com/technetwork/documentation/oracle-unified-ss-193371.html
- Oracle ZFS Storage Appliance Administration Guide can be accessed using the online help, which can be accessed through the Appliance BUI
- *JavaScript: The Definitive Guide, 6th Edition* by David Flanagan (O'Reilly Media, 2006)
- ECMA Script Wikipedia page: http://en.wikipedia.org/wiki/ECMAScript
- JavaScript Wikipedia page: http://en.wikipedia.org/wiki/JavaScript
- ECMA Scripting Language Specification: http://www.ecmascript.org/docs.php
- JavaScript online tutorial: http://www.howtocreate.co.uk/tutorials/javascript/introduction

Appendix B: Using the Oracle ZFS Storage Appliance Simulator

The Oracle ZFS Storage Appliance Simulator is an excellent platform for getting familiar with the CLI interface and the scripting language. Access the CLI either using the Oracle ZFS Storage Appliance console or an SSH connection.

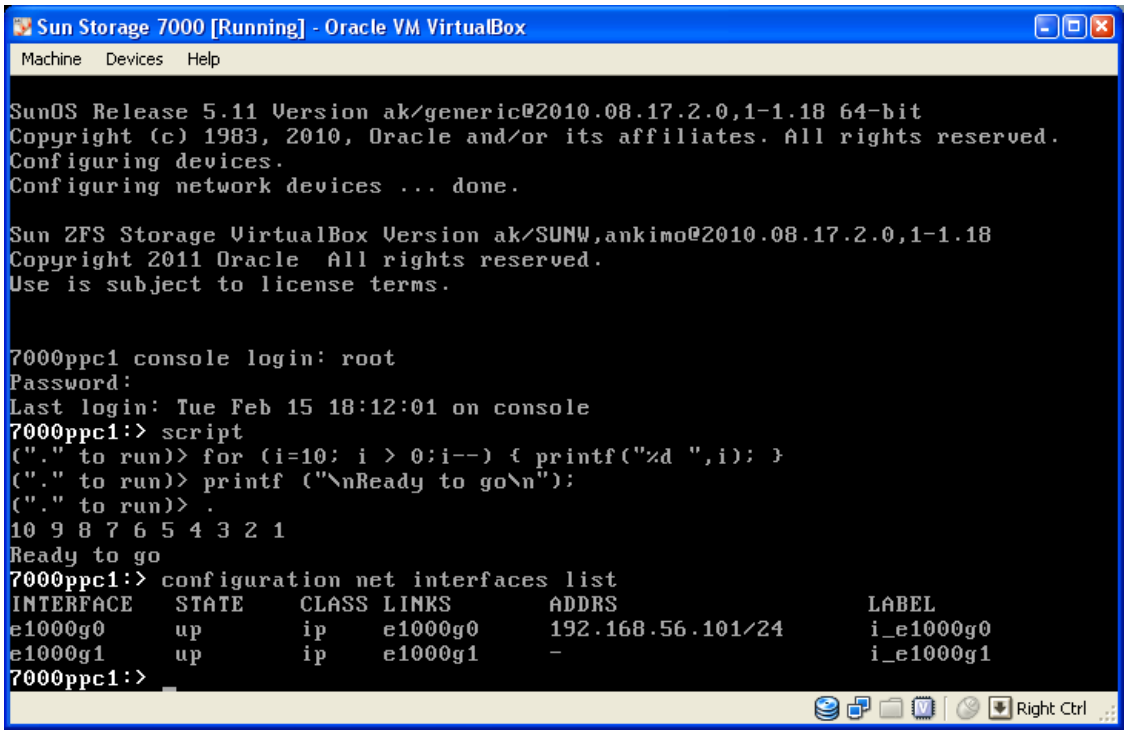

Figure 10. Oracle ZFS Storage Appliance console access using the simulator

Script commands can be entered interactively to explore the syntax and use of the JavaScript language in the Oracle ZFS Storage Appliance.

You can use the network interface configuration information obtained through the console to start an SSH connection to the Oracle ZFS Storage Appliance. Using SSH makes it easy to execute a batch of CLI commands or some simple scripts in the Oracle ZFS Storage Appliance.

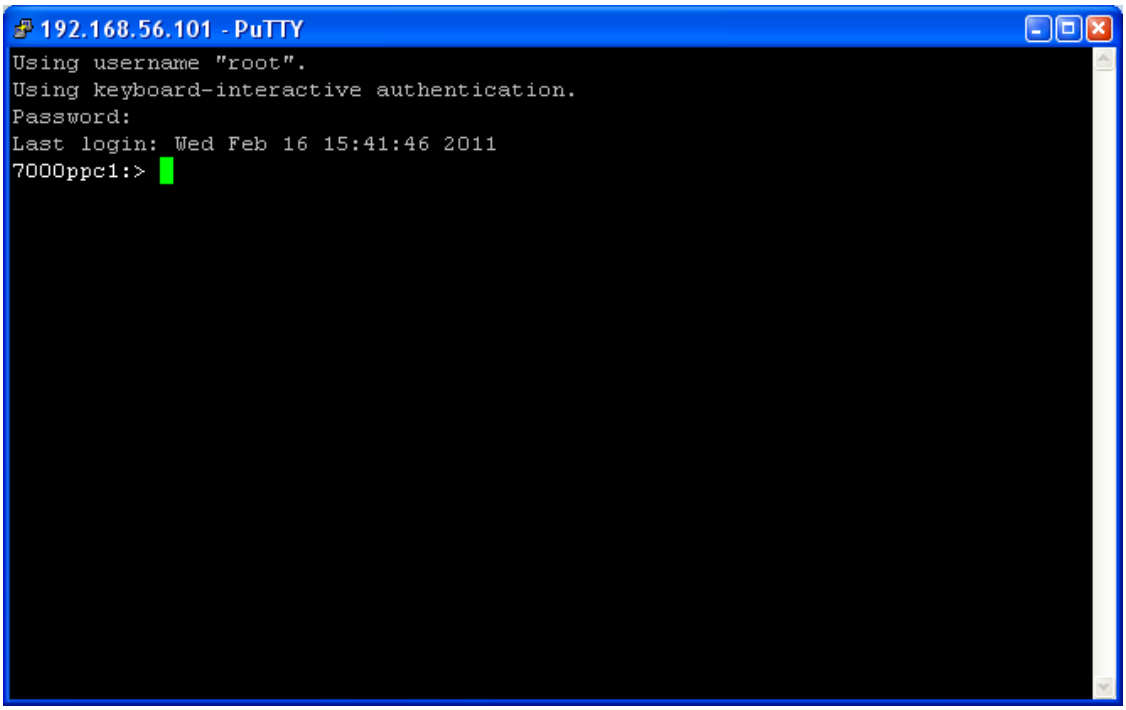

Figure 11. SSH connection to the Oracle ZFS Storage Appliance

Commands can be grouped in a file. By feeding the file into the `stdin` option of the `ssh` command, you can feed a batch of commands into the Oracle ZFS Storage Appliance.

For example, the following commands were put into the file `mybatchjob`:

```
Configuration net interfaces list
script printf("hello\n") ; printf("End of My Batch Job\n");
```

Then the file was fed into the `ssh` command:

```
pBrouwers MacBook-Pro:~ pBrouwer$ ssh root@192.168.56.101 < mybatchjob
Password:
Last login: Mon Feb 14 16:09:05 2011 from 192.168.56.1
INTERFACE     STATE  CLASS  LINKS         ADDRESS             LABEL
E1000g0       up     ip     e1000g0       192.168.56.101      i_e1000g0

Hello
End of My Batch Job
pBrouwers MacBook-Pro:~ pBrouwer$
```

Effectively Managing the Oracle ZFS Storage Appliance with Scripting
February 2014, Version 2.0
Author: Peter Brouwer

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together