Opsware® Automation Platform Developer's Guide: Pre-Release Draft

Corporate Headquarters

599 North Mathilda Avenue Sunnyvale, California 94085 U.S.A.
T + 1 408.744.7300 F +1 408.744.7383 www.opsware.com

Opsware SAS Version 6.0.1

Table of Contents

Preface

Pre-Release Status

The Opsware Automation Platform, API, and CLI described in this document are not yet supported. However, "preview" versions of these technologies are included in this release of Opsware SAS. You can learn about these technologies by experimenting with them in a test environment, but do not use them in a production environment.

Opsware SAS version: 6.0.1
Document Date: August 30, 2006

Welcome to the Opsware Server Automation System (SAS) – an enterprise-class software solution that enables customers to get all the benefits of Opsware Inc.'s data center automation platform and support services. Opsware SAS provides a core foundation for automating formerly manual tasks associated with the deployment, support, and growth of server and server application infrastructure.

About this Guide

This guide is a pre-release, partial draft.

Intended for advanced system administrators and software developers, this guide explains how to create client applications for the Opsware Automation Platform.

Contents of this Guide

This guide contains the following chapters:

Chapter 1: Overview - Summarizes the Opsware Automation Platform, the Opsware API, and the supported client technologies.

Chapter 2: Opsware OCLI Methods - Explains the concepts and syntax of the Opsware CLI methods. Provides scripting examples for the methods.

Chapter 3: TBD - The remaining chapters in this guide have not yet been written.

Chapter 1: Overview

IN THIS CHAPTER

This chapter discusses the following topics:

- Overview of the Opsware Automation Platform
- Opsware API Design

Overview of the Opsware Automation Platform

The Opsware Automation Platform (OAP) is a set of APIs and a runtime environment that facilitate the integration and extension of Opsware SAS. The Opsware APIs expose core services such as audit compliance, Windows patch management, and OS provisioning. The runtime environment (hub) executes scripts that can access the Opsware Global File System (OGFS).

With the Opsware Automation Platform, you can perform the following tasks:

- Build new automation applications and extend Opsware SAS to improve IT productivity and comply with your IT policies.
- Exchange information with other IT systems, such as existing monitoring, trouble ticketing, billing, and virtualization technology.
- Use the Opsware Model Repository to store and organize critical IT information about operations, environment, and assets.
- Automate the management of a wide range of applications and operating systems without having to wait for Opsware, Inc. to deliver out-of-the-box support for a particular technology.
- Incorporate existing Unix and Windows scripts with Opsware SAS, enabling the scripts to run in a secure, audited environment.

Supported Clients

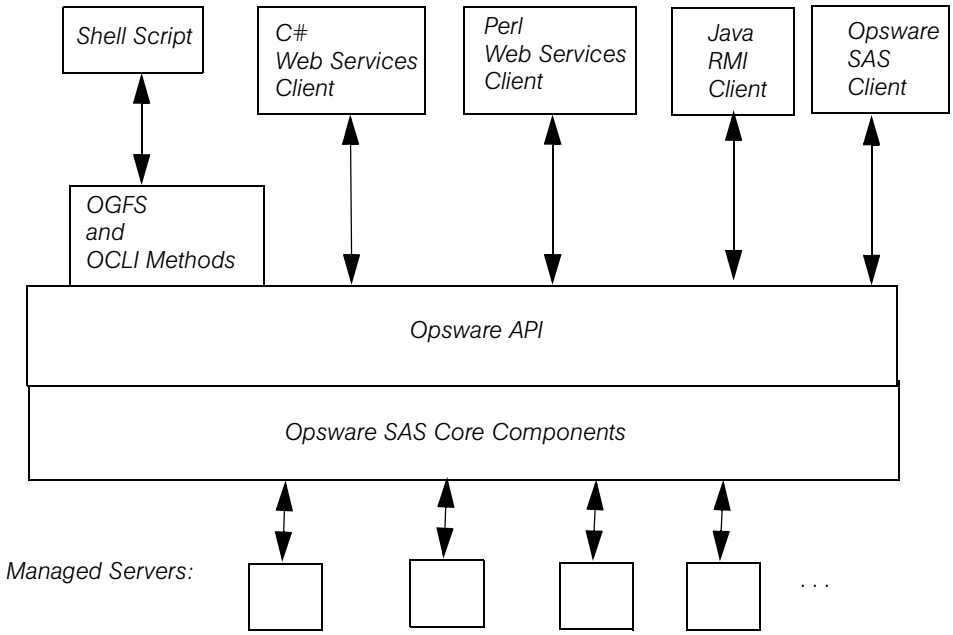
The Opsware Automation Platform supports programmers with different skills, from system administrators who write shell scripts to Java programmers familiar with the latest tools and technologies. All of the clients shown in Figure 1-1 call the same set of methods, which are organized into the services of the Opsware Automation Platform. You can create the following types of clients that call methods in the Opsware API:

- **Opsware Command-line Interface (OCLI)** - Launched from Global Shell sessions, shell scripts can access the Opsware API by invoking the OCLI methods, which are executable programs in the OGFS. Each OCLI method corresponds to a method in the Opsware API.
- **Web Services** - Using SOAP over HTTPS, these clients send requests to Opsware SAS and get responses back. The Web Services operations (defined in WSDLs) correspond to the methods in the Opsware API. You can write Web Services clients in popular languages such as Perl and C#.
- **Java RMI** - These clients invoke remote Java objects from other Java virtual machines.

The Web Services and Java RMI clients can run on servers different than the Opsware SAS core or managed servers. The OCLI methods execute in a Global Shell session on the core server where the OGFS is installed.

The Opsware SAS Client invokes the same APIs and follows the same security model as the other clients shown in Figure 1-1. Formerly called the OCC Client, the Opsware SAS Client is the desktop GUI application for end-users. The Opsware SAS client is included in the installation and cannot be modified on-site.

Figure 1-1: Clients of the Opsware Automation Platform



Security

Users of the Opsware Automation Platform must be authenticated and authorized to invoke methods on the Opsware API. To connect to Opsware SAS, a client supplies an Opsware user name and password (authentication). To invoke methods, the Opsware user must belong to a user group with the necessary permissions (authorization). These permissions restrict not only the types of Opsware SAS operations that users can perform, but also limit access to the servers and network devices used in the operations.

Before running clients, you must specify the required users and permissions with the Opsware Command Center. For instructions, see the User Group and Setup chapter of the *Opsware® SAS Administration Guide*. For information about security-related exceptions, see “Exceptions” on page 14.

Communication between clients and Opsware SAS is encrypted. For Web Services clients, the request and response SOAP messages (which implement the operation calls) are encrypted using SSL over HTTP (HTTPS).

Opsware API Design

The Opsware API is defined by Java interfaces and organized into Java packages. To support a variety of client languages and remote access protocols, the API follows a function-oriented, call-by-value model.

Services

In the Opsware API, a service encapsulates a set of related functions. Each service is specified by a Java interface with a name ending in `Service`, such as `ServerService`, `FolderService`, and `JobService`.

Services are the entry points into the API. To access the API, clients invoke the methods defined by the server interface. For example, to retrieve a list of software installed on a managed server, a client invokes the `getInstalledSoftware` method of the `ServerService` interface. Examples of other `ServerService` methods are `checkDuplex`, `setPrimaryInterface`, and `changeCustomer`.

The Opsware API contains over 70 services – too many to describe here. Table 1-1 lists a few of the services that you may want to try out first. For a full list of services, in a browser go to the URL shown in "API Documentation and the Twister" on page 16.

Table 1-1: Partial List of Services of the Opsware API

SERVICE NAME	SOME OF THE OPERATIONS PROVIDED BY THIS SERVICE
`AuditTaskService`	Create, get, and run audit tasks.
`ConfigurationService`	Create application configurations, get the software policies using an application configuration.
`DeviceGroupService`	Create device groups, assign devices to groups, get members of groups, set dynamic rules.
`EventCacheService`	Trigger actions such as updating a client-side cache of value objects. See "Event Cache" on page 15.
`FolderService`	Create folders, get children of folders, set customers of folders, move folders.

Table 1-1: Partial List of Services of the Opsware API (continued)

SERVICE NAME	SOME OF THE OPERATIONS PROVIDED BY THIS SERVICE
`InstallProfileService`	Create, get, and update OS installation profiles.
`JobService`	Get progress and results of jobs, cancel jobs, update job schedules.
`NasConnectionService`	Get host names of NAS servers, run commands on NAS servers.
`NetworkDeviceService`	Get information such as families, names, models, and types, according to specified search filters.
`SequenceService`	Create, get, and run OS sequences to install operating systems on servers.
`ServerService`	Get information about servers, reconcile (remediate) policies on servers (install software), get and set custom fields and attributes, execute OS sequences (install OS).
`SoftwarePolicyService`	Create software policies, assign policies to servers, get contents of policies, remediate (reconcile) policies with servers.
`SolPatchService`	Install and uninstall Solaris patches, add policy overrides.
`VirtualColumnService`	Manage custom fields and custom attributes.
`WindowsPatchService`	Install and uninstall Windows patches, add policy overrides.

Objects in the API

Although the Opsware API is function-oriented, its design enables clients to create object-oriented libraries. The Opsware SAS data model includes objects such as servers, folders, and customers. These are persistent objects; that is, they are stored in the Opsware Model Repository. In the API, these objects have the following items:

- A service that defines the object's behavior. For example, the methods of the `ServerService` specify the behavior of a managed server object.
- An object (identity) reference that represents an instance of a persistent object. For example, `ServerRef` is a reference that uniquely identifies a managed server. In the `ServerService`, the first parameter of most methods is `ServerRef`, which identifies the managed server operated on by the method. The `Id` attribute of a `ServerRef` is the primary key of the server object stored in the Opsware Model Repository.
- One or more value objects (VOs) that represent the data members (attributes, fields) of a persistent object. For example, `ServerVO` contains attributes such as `agentVersion` and `loopbackIP`. The attributes of `ServerHardwareVO` include `manufacurer`, `model`, and `assetTag`. Most attributes cannot be changed by client applications. If an attribute can be changed, then the API documentation for the setter method includes "Field can be set by clients."

For performance reasons, update operations on persistent objects are coarse-grained. The `update` method of `ServerService`, for example, accepts the entire `ServerVO` as an argument, not individual attributes.

Exceptions

All of the API exceptions that are specific to Opsware SAS are derived from one of the following exceptions:

- `OpswareException` - Thrown when an application-level error occurs, such as when an end-user enters an illegal value that is passed along to a method. Typically, the client application can recover from this type of exception. Examples of exceptions derived from `OpswareException` are `NotFoundException`, `NotInFolderException`, and `JobNotScheduledException`.
- `OpswareSystemException` - Thrown when an error occurs within Opsware SAS. Usually, the Opsware Administrator must resolve the problem before the client application can run.

The following exceptions are related to security:

- `AuthenticationException` - Thrown when an invalid Opsware user name or password is specified.

- `AuthorizationException` - Thrown when the user does not have permission to perform an operation or access an object. For more information on permissions, see the *Opsware® SAS Administration Guide*.

Event Cache

Some client applications need to keep local copies of Opsware SAS objects. Accessed by clients through the `EventCacheService`, the cache contains events that describe the most recent change made to Opsware SAS objects. Clients can periodically poll the cache to check whether objects have been created, updated, or deleted. The cache maintains events over a configured sliding window of time. By default, events for the most recent 2 hours are maintained. To change the sliding window size, edit the Web Services Data Access Engine configuration file, as described in the *Opsware® SAS Administration Guide*.

Searches

The search mechanism of the Opsware API retrieves object references according to the attributes (fields) of value objects. For example, the `getServerRefs` method searches by attributes of the `ServerVO` value object. The `getServerRefs` method has the following signature:

```
public ServerRef[] getServerRefs(Filter filter) . . .
```

Each `get*Refs` method accepts the `filter` parameter, an object that specifies the search criteria. A `filter` parameter with a simple expression has the following syntax:

`value-object.attribute operator value`

(This syntax is simplified. For the full definition, see “Filter Grammar” on page 55.)

The following examples are `filter` parameters for the `getServerRefs` method:

```
ServerVO.hostName = "d04.opsware.com"
ServerVO.model BEGINS_WITH "POWER"
ServerVO.use IN "UNKNOWN" "PRODUCTION"
```

Complex expressions are allowed, for example:

```
(ServerVO.model BEGINS_WITH "POWER") AND (ServerVO.use =
"UNKNOWN")
```

Not every attribute of a value object can be specified in a `filter` parameter. For example, `ServerVO.state` is allowed in a `filter` parameter, but `ServerVO.OsFlavor` is not. To find out which attributes are allowed, locate the value object in the API documentation and look for the comment, "Field can be used in a filter query."

API Documentation and the Twister

The Opsware SAS 6.0 core ships with API documentation (javadocs) that describe the Opsware API. To access the API documentation, specify the following URL in your browser:

`https://`*`occ_host`*`:1032/twister/docs/index.html`

Or:

`https://`*`occ_host`*`:443/twister/docs/index.html`

The *occ_host* is the IP address or host name of the core server running the Opsware Command Center component.

To list the services in the API documentation, specify the following URL:

`https://`*`occ_host`*`:443`

Also included in the core, the Twister is a program that lets you invoke API methods, one at a time, from within a browser. For example, to invoke the `ServerService.getServerVO` method, perform the following steps:

1. Open the API documentation in a browser.
2. In the All Classes pane, select `com.opsware.server`.
3. In the `com.opsware.server` pane, select `ServerService`.
4. In the main pane, scroll down to the `getServerVO` method.
5. Click **Try It** for the `getServerVO` method.
6. Enter your Opsware SAS user name and password.
7. In the Twister pane for `ServerService.getServerVO`, enter the ID of a managed server in the `oid` field.
8. Click **Go**. The Twister pane displays the attributes of the `ServerVO` object returned.

Java RMI Client JAR File

To compile and run Java RMI clients, you need the `opswclient.jar` file, which you can download from the following location:

`https://`*`occ_host`*`/twister/opswclient.jar`

Importing and Exporting Packages With PUT and GET

The following wiki page is available only to Opsware, Inc. employees:

`http://wiki.corp.opsware.com/owiki/OpswareReleases_2fEinstein_2fPatchManagement_2fFileTransferApi`

Chapter 2: Opsware CLI Methods

IN THIS CHAPTER

This chapter contains the following topics:

Overview of Opsware CLI Methods

In order to understand this chapter, you should already be familiar with the Opsware Global Shell and the OGFS. For a quick introduction to these features, see the "Global Shell Tutorial" in the *Opsware® SAS User's Guide: Server Automation*.

End-users access Opsware SAS through the GUI utilities, that is, the SAS Client and the SAS Web Client. At times, advanced users need to access Opsware SAS in a command-line environment to perform bulk operations or repetitive tasks on multiple servers. In Opsware SAS, the command-line environment consists of the Global Shell, OGFS, and Opsware Command-line Interface (OCLI) methods.

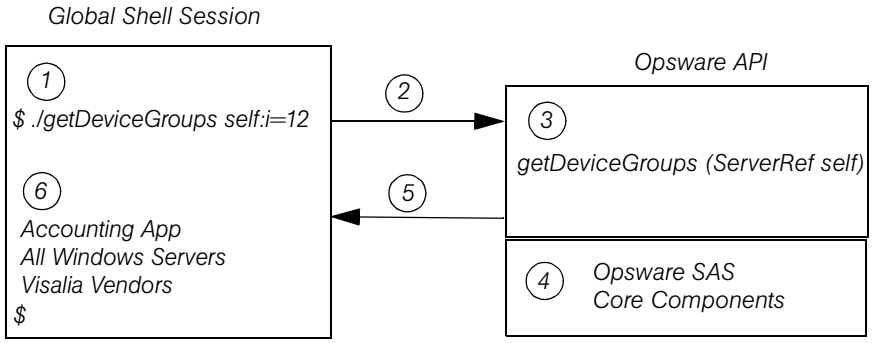
To perform Opsware SAS operations from the command-line, you invoke OCLI methods from within a Global Shell session. An OCLI method is an executable in the OGFS that corresponds to a method in the Opsware API. When you run an OCLI method, the underlying API method is invoked.

Method Invocation

As shown by Figure 2-1 when an OCLI method is invoked, the following operations occur:

1. In a Global Shell session, the user enters an OCLI method with parameters.
2. The command-line entered in the previous step is parsed to determine the API method and parameters.
3. The underlying API method is invoked.
4. An authorization check verifies that the user has permission to perform this operation. Then, Opsware SAS performs the operation.
5. The API method passes the results back to the OCLI method.
6. The OCLI method writes the return value to the `stdout` of the Global Shell session. If an exception was thrown, the OCLI method returns a non-zero status.

Figure 2-1: Overview of an OCLI Method Invocation



Security

OCLI methods use the same authentication and authorization mechanisms as the SAS Client and the SAS Web Client. When you start a Global Shell session, Opsware SAS authenticates your Opsware user. When you run an OCLI method, authorization is performed. To run an OCLI method successfully, your Opsware user must belong to a group that has the required permissions. For more information on security, see the *Opsware® SAS Administration Guide*.

Mapping Between API and OCLI Methods

The OGFS represents Opsware SAS objects as directory structures, object attributes as text files, and API methods as executables. These executables are the OCLI methods. Every OCLI method matches an underlying API method. The method name, parameters, and return value are the same for both types of methods.

For example, the `setCustomer` API method has the following Java signature:

```
public void setCustomer(ServerRef self,
                        CustomerRef customer). . .
```

In the OGFS, the corresponding OCLI method has the following syntax:

```
setCustomer self:i=server-id customer:i=customer-id
```

Note that the parameter names, `self` and `customer`, are the same in both languages. (The `:i` notations are called format specifiers, which are discussed later in this chapter.) In this example, the return type is void, so the OCLI method does not write the result to the `stdout`. For information on how OCLI methods return strings that represent objects, see "Return Values" on page 40.

Differences Between OCLI Methods and Unix Commands

Although you can run both Unix commands and OCLI methods in the Global Shell, OCLI methods differ in several ways:

- Unlike many Unix commands, OCLI methods do not read data from `stdin`. Therefore, you cannot insert an OCLI method within a group of commands connected by pipes (|). (However, OCLI methods do write to `stdout`.)
- Most Unix commands accept parameters as flags and values (for example, `ls -l /usr`). With OCLI methods, command-line parameters are name-value pairs, joined by equal signs.
- Unix commands are text based: They accept and return data as strings. In contrast, OCLI methods can accept and return complex objects.
- With OCLI methods, you can specify the format of the parameter and return values. Unix commands do not have an equivalent feature.

OCLI Method Tutorial

This tutorial introduces you to OCLI methods with a few examples for you to try out in your environment. After completing this tutorial, you should be able to run OCLI methods, examine the `self` file of an Opsware object, and create a script that invokes OCLI methods on multiple servers.

Before starting the tutorial, you need the following capabilities:

- You can log on to the SAS Client.
- Your Opsware user has Read & Write permissions on at least one managed server. Typically assigned by a security administrator, permissions are discussed in the *Opsware® SAS Administration Guide*.
- Your Opsware user has all Global Shell permissions on the same managed server. For information on these permissions, see the "aaa Utility" section in the *Opsware® SAS User's Guide: Server Automation*.
- You are familiar with the Global Shell and the OGFS. If these features are new to you, before proceeding with this tutorial you should step through the "Global Shell Tutorial" in the *Opsware® SAS User's Guide: Server Automation*.

The example commands in this tutorial operate on a Windows server named `abc.opsware.com`. This server belongs to a server group named All Windows Servers. When trying out these commands, substitute `abc.opsware.com` with the host name of the managed server you have permission to access.

1 Open a Global Shell session.

You can open a Global Shell session from within the SAS Client. From the **Actions** menu, select **Global Shell**. You can also open a Global Shell session from a terminal client running on your desktop. For instructions, see "Opening a Global Shell Session" in the *Opsware® SAS User's Guide: Server Automation*.

2 List the OCLI methods for a server.

The `method` subdirectory of a specific server contains executable files-- the methods you can run for that server. The following example lists the OCLI methods for the `abc.opsware.com` server:

```
$ cd /opsw/Server/@/abc.opsware.com/method
$ ls -1
addDeviceGroups
attachPolicies
attachVirtualColumn
```



```
checkDuplex
clearCustAttrs
. . .
```

These methods have instance context – they act on a specific server instance (in this case, `abc.opsware.com`). The server instance can be inferred from the path of the method. Methods with static context are discussed in step 5.

3 Run an OCLI method without parameters.

To display the public server groups that `abc.opsware.com` belongs to, invoke the `getDeviceGroups` method:

```
$ cd /opsw/Server/@/abc.opsware.com/method
$ ./getDeviceGroups
Accounting App
All Windows Servers
Visalia Vendors
```

4 Run a method with a parameter.

Command-line parameters for methods are indicated by name-value pairs, separated by white space characters. In the following invocation of `setCustomer`, the parameter name is `customer` and the value is 20039. The `:i` at the end of the parameter name is an ID format specifier, which is discussed in a later step.

The following method invocation changes the customer of the `abc.opsware.com` server from Opsware to C39. The ID of customer C39 is 20039.

```
$ cd /opsw/Server/@/abc.opsware.com
$ cat attr/customer ; echo
Opsware
$ method/setCustomer customer:i=20039
$ cat attr/customer ; echo
C39
```

5 List the static context methods for managed servers.

Static context methods reside under the `/opsw/api` directory. These methods are not limited to a specific instance of an object.

To list the static methods for servers, enter the following commands:

```
$ cd /opsw/api/com/opsware/server/ServerService/method
$ ls
```

The methods listed are the same as those displayed in step 2.

6 Run a method with the `self` parameter.

This step invokes `getDeviceGroups` as a static context method. Unlike the instance context method shown in step 3, the static context method requires the `self` parameter to identify the server instance.

For example, suppose that the `abc.opsware.com` server has an ID of 530039. To list the groups of this server, enter the following commands:

```
$ cd /opsw/api/com/opsware/server/ServerService/method
$ ./getDeviceGroups self:i=530039
Accounting App
All Windows Servers
Visalia Vendors
```

Compare this invocation of `getDeviceGroups` with the invocation in step 3 that demonstrates instance context. Both invocations run the same underlying method in the API and return the same results.

7 Examine the `self` file of a server.

Within Opsware SAS, each managed server is an object. However, OGFS is a file system, not an object model. The `self` file provides access to various representations of an Opsware SAS object. These representations are the ID, name, and structure.

The default representation for a server is its name. For example, to display the name of a server, enter the following commands:

```
$ cd /opsw/Server/@/abc.opsware.com
$ cat self ; echo
abc.opsware.com
```

If you know the ID of a server, you can get the name from the `self` file, as in the following example:

```
$ cat /opsw/.Server.ID/530039/self ; echo
abc.opsware.com
```

8 Indicate an ID format specifier on a `self` file.

To select a particular representation of the `self` file, enter a period, then the file name, followed by the format specifier. For example, the following `cat` command includes the format specifier (`:i`) to display the server ID:

```
$ cd /opsw/Server/@/abc.opsware.com
$ cat .self:i ; echo
com.opsware.server.ServerRef:530039
```


This output shows that the ID of `abc.opsware.com` is 530039. The `com.opsware.server.ServerRef` is the class name of a server reference, the corresponding object in the Opsware API.

The leading period is required with format specifiers on files and method return values, but is not indicated with method parameters.

9 Indicate the structure format specifier.

The structure format specifier (`:s`) indicates the attributes of a complex object. The attributes are displayed as name-value pairs, all enclosed in curly braces. Structure formats are used to specify method parameters on the command-line that are complex objects. (For an example method call, see "Complex Objects and Arrays As Parameters" on page 39.)

The following example displays `abc.opsware.com` with the structure format:

```
$ cd /opsw/Server/@/abc.opsware.com
$ cat .self:s ; echo
{
managementIP="192.168.8.217"
modifiedBy="spujare"
manufacturer="DELL COMPUTER CORPORATION"
use="UNKNOWN"
discoveredDate=1149012848000
origin="ASSIMILATED"
osSPVersion="SP4"
locale="English_United States.1252"
reporting=false
netBIOSName=
previousSWReg=1150673874000
osFlavor="Windows 2000 Advanced Server"
. . .
```

The attributes of a server are also represented by the files in the `attr` directory, for example:

```
$ pwd
/opsw/Server/@/abc.opsware.com
$ cat attr/osFlavor ; echo
Windows 2000 Advanced Server
```

10 Create a script that invokes an OCLI method.

The example script shown in this step iterates through the servers of the public server group named All Windows Servers. On each server, the script runs the `getCommCheckTime` OCLI method.

First, return to your home directory in the OGFS:

```
$ cd
$ cd public/bin
```

Next, run the `vi` editor:

```
$ vi
```

In `vi`, insert the following lines to create a `bash` script:

```
#!/bin/bash
# iterate_time.sh

METHOD_DIR="/opsw/api/com/opsware/server/ServerService/
method"
GROUP_NAME="All Windows Servers"
cd "/opsw/Group/Public/$GROUP_NAME/@/Server"

for SERVER_NAME in *
do
   SERVER_ID=`cat $SERVER_NAME/.self:i`
   echo $SERVER_NAME
   $METHOD_DIR/getCommCheckTime self:i=$SERVER_ID
   echo
   echo
done
```

Save the file in `vi`, naming it `iterate_time.sh`. Quit `vi`.

Change the permissions of `iterate_time.sh` with `chmod`, and then run it:

```
$ chmod 755 iterate_time.sh
$ ./iterate_time.sh
abc.opsware.com
2006/06/20 16:46:56.000
. . .
```


Format Specifiers

Format specifiers indicate how values are displayed or interpreted in the OCLI environment. You can apply a format specifier to a method parameter, a method return type, the `self` file, and an object attribute. To indicate a format specifier, append a colon followed by one of the letters shown in Table 2-1.

If a format specifier is indicated for a file or a method return value, a period must precede the file or method name. For method return values that have format specifiers, the leading period is not included.

Table 2-1: Summary of Format Specifiers

FORMAT SPECIFIER	DESCRIPTION	VALID OBJECT TYPES	ALLOWED AS METHOD PARAMETER?
`:n`	**Name**: A string identifying the object. Unique names are preferred, but not required. For objects that do not have a name, this representation is the same as the ID representation.	Opsware objects	Yes. If the name is ambiguous, an error occurs.
`:i`	**ID**: A format that uniquely identifies the object type and its Opsware ID. Also known as an object reference.	Opsware objects; Dates (`java.util.Calendar`) objects	Yes. If the type is clear from the context, the type may be omitted.
`:s`	**Structure**: A compact representation intended for specifying complex values on the command-line. Attributes are enclosed in curly braces.	Any complex object	Yes

Table 2-1: Summary of Format Specifiers (continued)

FORMAT SPECIFIER	DESCRIPTION	VALID OBJECT TYPES	ALLOWED AS METHOD PARAMETER?
`:d`	**Directory**: Represents an attribute as a directory in the OGFS.	Any complex object that is an attribute. This representation cannot be used for method parameters or return values.	No

Position of Format Specifiers

A format specifier immediately follows the item it affects. For files, a format specifier follows the file name. In the following example, note the leading period:

```
cat .self:s
```

When applied to a method return type, a format specifier follows the method name. The following invocation displays the IDs of the groups returned:

```
./.getDeviceGroups:i
```

With method parameters, a format specifier follows the parameter name and precedes the equal sign, as in the following example:

```
./setCustomer self:i=9977 customer:i=239
```

A method parameter with a format specifier does not have a leading period.

Default Format Specifiers

Every value or object has a default format specifier. For example, the name format specifier is the default for the `osVersion` attribute. The following two `cat` commands generate the same output:

```
cd /opsw/Server/@/d04.opsware.com/attr
cat osVersion
cat .osVersion:n
```

The name format specifier is the default for Opsware objects stored in the Model Repository, such as servers and customers. The structure format specifier is the default for other complex objects.

ID Format Specifier Examples

The next example displays the ID of the facility that the `d04.opsware.com` server belongs to:

```
cd /opsw/Server/@/d04.opsware.com/attr
cat .facility:i ; echo
```

(The preceding `echo` command is optional. It generates a new-line character, which makes the output easier to read. The semicolon separates `bash` statements entered on the same line.)

The output of a value with the ID format specifier is prefixed by the Java class name. For example, if the facility value has an ID of 39, then the previous `cat` command displays the following output:

```
com.opsware.locality.FacilityRef:39
```

The following invocation of the `getDeviceGroups` method lists the IDs of the public server groups that `d04.opsware.com` belongs to:

```
cd /opsw/Server/@/d04.opsware.com/method
./.getDeviceGroups:i
```

For more ID format examples, see “The self File” on page 34.

Structure Format Specifier Syntax

The structure format represents complex objects, which can contain various attributes. You might use this format to specify a method parameter that is a complex object. For examples, see “Complex Objects and Arrays As Parameters” on page 39.

The structure format is a series of name-value pairs, separated by white space characters, enclosed in curly braces. Each name-value pair represents an attribute. The structure format has the following syntax:

```
{ name-1=value-1 name-2=value-2 . . . }
```

Here’s a simple example:

```
{ version=10.1.3 isCurrent=true }
```

Any white space character can be used as a delimiter:

```
{
  version=10.1.3
  isCurrent=true
}
```

Attributes can be specified as structures, enabling the representation of nested objects. In the following example, the `versionDesc` attribute is represented as a structure:

```
{
program=agent
versionDesc={
        version=10.1.3
        isCurrent=true
        comment="Latest version"
        }
}
```

To specify an array within a structure, repeat the attribute name. The following structure contains an array named `steps` that has three elements with the values 33, 14, and 28.

```
{ moduleName="Some Initiator" steps=33 steps=14 steps=28 }
```

Structure Format Specifier Examples

The following example specifies the structure format for the `facility` attribute:

```
cd /opsw/Server/@/d04.opsware.com/attr
cat .facility:s
```

This `cat` command generates the following output. Note that `customers` is an array, which contains an element for every customer associated with this facility.

```
{
modifiedBy="192.168.9.246"
customers="Customer Independent"
customers="Not Assigned"
customers="Opsware Inc."
customers="Acme Inc."
. . .
ontogeny="PROD"
createdBy=
status="ACTIVE"
createdDt=-1
realms="Transitional"
realms="C39"
realms="C39-agents"
modifiedDt=1146528752000
name="C39"
displayName="C39"
}
```

The following invocation of `getDeviceGroups` indicates the structure format specifier for the return value:


```
cd /opsw/Server/@/d04.opsware.com/method
./.getDeviceGroups:s
```

This call to `getDeviceGroups` displays the following output. Because `d04.opsware.com` belongs to two server groups, the output includes two structures. In each structure, the `devices` array has elements for the servers belonging to that group.

```
{
dynamic=true
devices="m302-w2k-vm1.dev.opsware.com"
devices="d04.opsware.com"
. . .
status="ACTIVE"
public=true
fullName="Device Groups Public All Windows Servers"
description="test"
createdDt=-1
modifiedDt=1142019861000
parent="Public"
}

{
dynamic=true
devices="opsware-nibwp.build.opsware.com"
devices="glengarriff.snv1.dev.opsware.com"
devices="millstreet"
. . .
fullName="Device Groups Public z_testsrvgroup"
. . .
}
```

The structure format specifier is the default for methods that retrieve value objects (VOs). For example, the following two calls to `getServerVO` are equivalent:

```
cd /opsw/Server/@/d04.opsware.com/method
./.getServerVO:s
./getServerVO
```

In this example, `getServerVO` displays the following output:

```
{
managementIP="192.168.198.93"
modifiedBy=
manufacturer="DELL COMPUTER CORPORATION"
use="UNKNOWN"
discoveredDate=1145308867000
origin="ASSIMILATED"
osSPVersion="RTM"
```

```
locale="English_United States.1252"
reporting=false
netBIOSName=
previousSWReg=1147678609000
osFlavor="Windows Server 2003, Standard Edition"
peerIP="192.168.198.93"
modifiedDt=1145308868000
. . .
serialNumber="HVKZS51"
}
```

This structure represents the `ServerVO` class of the Opsware API. Every attribute in this structure corresponds to a file in the `attr` directory. In the next example, the `getServerVO` and `cat` commands both display the value of the `serialNumber` attribute of a server:

```
cd /opsw/Server/@/d04.opsware.com
./method/getServerVO | grep serialNumber
cat attr/serialNumber ; echo
```

Directory Format Specifier Examples

The following command changes the current working directory to the customer associated with the server `d04.opsware.com`:

```
cd /opsw/Server/@/d04.opsware.com/attr/.customer:d
```

The next command lists the name of this customer:

```
cat /opsw/Server/@/d04.opsware.com/attr/\
.customer:d/attr/name
```

The directory specifier can be used only in command arguments that require directory names. The following `cat` command fails because it attempts to display a directory:

```
cat /opsw/Server/@/d04.opsware.com/attr/.customer:d # WRONG!
```

However, the next command is legal:

```
ls /opsw/Server/@/d04.opsware.com/attr/.customer:d
```

Value Representation

Because they run in a shell environment (Global Shell), OCLI methods accept and return data as strings. However, the underlying API methods can accept and return other data types, such as numbers, booleans, and objects. The sections that follow describe how the OGFS and OCLI methods represent non-string data types.

Opsware Objects in the OGFS

The Opsware data model includes objects such as servers, server groups, customers, and facilities. In the OGFS, these objects are represented as directory structures:

```
/opsw/Customer
/opsw/Facility
/opsw/Group
/opsw/Library
/opsw/Realm
/opsw/Server
. . .
```

The preceding list is not complete. To see the full list, enter `ls /opsw`.

Object Attributes

The attributes of an Opsware SAS object are represented by text files in the `attr` subdirectory. The name of each file matches the name of the attribute. The contents of a file reveals the value of the attribute.

For example, the `/opsw/Server/@/buzz.opsware.com/attr` directory contains the following files:

```
agentVersion
codeset
createdBy
createdDt
customer
defaultGw
description
discoveredDate
facility
hostName
locale
lockInfo
loopbackIP
managementIP
manufacturer
. . .
```

To display the management IP address of the `buzz.opsware.com` server, enter the following commands:

```
cd /opsw/Server/@/buzz.opsware.com/attr
cat managementIP ; echo
```

Custom Attributes

Custom attributes are name-value pairs that you can assign to Opsware objects such as servers. In the OGFS, custom attributes are represented as text files in the `CustAttr` subdirectory. You can create custom attributes in a Global Shell session by creating new text files under `CustAttr`. The following example creates a custom attribute named `MyGreeting`, with a value of `hello there`, on the `buzz.opsware.com` server:

```
cd /opsw/Server/@/buzz.opsware.com/CustAttr
echo -n "hello there" > MyGreeting
```

For more examples, see "Managing Custom Attributes" in *Opsware® SAS User's Guide: Server Automation*.

The self File

The `self` file resides in the directory of an Opsware SAS object such as a server or customer. This file provides access to various representations of the current object, depending on the format specifier. (For details, see "Format Specifiers" on page 27.)

To list the ID of the `buzz.opsware.com` server, enter the following commands:

```
cd /opsw/Server/@/buzz.opsware.com
cat .self:i ; echo
```

For a server, the default format specifier is the name. The following commands display the same output:

```
cat self ; echo
cat .self:n ; echo
```

The next command lists the attributes of a server in the structure format:

```
cat .self:s
```


Primitive Values

Table 2-2 indicates how primitive values are converted between the API and their string representations in OCLI methods. Except for Dates, primitive values do not support format specifiers. Dates support ID format specifiers.

Table 2-2: Conversion Between Primitive Types and OCLI Methods

PRIMITIVE TYPE	JAVA EQUIVALENT	OUTPUT FROM OCLI METHOD	INPUT TO CLI METHODS
String	`java.lang.String`	Character string, presented in the encoding of the current session.	Character string, converted to Unicode from the current session encoding.
Number	`byte, short, int, long, float, double`; and their object equivalents	Decimal format, not localized. Scientific notation for very large or small values.	Examples - Decimal: `101, 512.34, -104` Hex: `0x1F32, 0x2e40` Octal: `0543` Scientific: `4.3E4, 6.532e-9, 1.945e+02`
Boolean	`boolean, Boolean`	`true` or `false`	The string “true” and all mixed-case variants evaluate to `true`. All other values evaluate to `false`.
Binary data	`byte[], Byte[]`	Binary string. No conversion from session encoding.	Binary string. No conversion to session encoding.

Table 2-2: Conversion Between Primitive Types and OCLI Methods

PRIMITIVE TYPE	JAVA EQUIVALENT	OUTPUT FROM OCLI METHOD	INPUT TO CLI METHODS
Date	`java.util.Calendar`	Date value. By default, presented in this format: `YYYY/MM/DD HH:MM:SSS` The time is presented in UTC. If an ID format specifier is indicated, the value is presented as the number of milliseconds since the epoch, in UTC.	Same as output.

Arrays

The representation of array objects depends on whether they are standalone (an array attribute file or a method return value) or contained in the structure of a complex object.

First, standalone array objects are presented according the the underlying type, separated by new-line characters. Within an array element, a new-line character is escaped by \n and a backslash by \\.

Array values can be output or input using any representation supported by the underlying type. For example, by default, the `getDeviceGroups` method lists the groups as names:

```
All Windows Servers
Servers in Austin
Testing Pool
```

If you indicate the ID format specifier, (`.getDeviceGroups:i`) the method displays the IDs of the groups:

```
com.opsware.device.DeviceGroupRef:15960039
com.opsware.device.DeviceGroupRef:10390039
com.opsware.device.DeviceGroupRef:17380039
```


Second, an array contained in the structure of a complex object is represented as a set of name-value pairs, using the attribute as the name. The attribute appears multiple times, once for each element in the array. The order in which the attributes appear determine the order of the elements in the array. The following example shows a structure that contains two attributes, a string called `subject` and a three-element array of numbers called `ranks`:

```
{ subject="my favorites" ranks=17 ranks=44 ranks=24 }
```

Arrays can also be represented by directories. Within an array directory, each array element has a corresponding file (for primitive types) or subdirectory (for complex types). The name of each entry is the index number of the array element, starting with zero.

For an array that is the attribute of a complex object, you should modify the array by editing its attribute file. This action completely replaces the array with the contents of the edited file.

For an array containing elements that are complex objects, you should modify the array by changing its directory representation. To change an element value, edit the element file. For example, suppose you have an array with five string elements. The `ls` command lists the elements as follows:

```
0 1 2 3 4
```

The following command changes the value of the third element:

```
echo -n "My new value" > 2
```

OCLI Method Parameters and Return Values

This section discusses the details of method context (instance or static), parameter usage, return values, and exit status.

Method Context and the self Parameter

In the OGFS, a method resides in multiple locations. The location of a method is related to its context, which is either instance or static.

The method with instance context resides in `method` directory of a specific Opsware SAS object. The method invocation does not require the `self` parameter. The instance of the object affected by the method is implied by the method location. The following example changes the customer of the `d04.opsware.com` server:

```
cd /opsw/Server/@/d04.opsware.com/method
```

```
./setCustomer customer:i=9
```

A method with static context resides in a single location under `/opsw/api`. The method invocation requires the self parameter to identify the instance affected by the method. In the following static context example, `self:i` specifies the ID of the managed server:

```
cd /opsw/api/com/opsware/server/ServerService/method
./setCustomer self:i=230054 customer:i=9
```

Passing Arguments on the Command-Line

The command-line arguments are specified as name-value pairs, joined by the equal sign (=). The name-value pairs are separated by one or more white space characters, typically spaces. The names on the command-line match the parameter names of the corresponding Java method in the Opsware API.

For example, in the Opsware API, the `setCustomField` method has the following definition:

```
public void setCustomField(CustomFieldReference self,
   java.lang.String fieldName, java.lang.String strValue)...
```

The following OCLI method example assigns a value to a custom field of the server with ID 3670039:

```
cd /opsw/api/com/opsware/server/ServerService/method
./setCustomField self:i=3670039 \
fieldName="Service Agreement" strValue="Gold"
```

As described in the previous section, a method with an instance context does not require the `self` parameter. The following `setCustomField` example is equivalent to the preceding example:

```
cd /opsw/.Server.ID/3670039
./setCustomField \
fieldName="Service Agreement" strValue="Gold"
```

You can specify the command-line arguments in any order. The following two OCLI method invocations are equivalent:

```
./setCustomField fieldName="My Stuff" strValue="abc"
./setCustomField strValue="abc" fieldName="My Stuff"
```

To specify a null value for a parameter, either omit the parameter or insert a white space after the equal sign. In the following examples, the value of `myParam` is null:

```
./someMethod myField="more info" myParam= anotherParam=9834
./someMethod myField="more info"          anotherParam=9834
```


Specifying the Type of a Parameter

If a method has an abstract type for a parameter, you must specify the concrete type as well as the value. In the following example, the `com.opsware.folder.FolderRef` type is required:

```
cd /opsw/api/com/opsware/folder/FolderService/method
./remove self:i="com.opsware.folder.FolderRef:730555"
```

Complex Objects and Arrays As Parameters

To pass an argument that is a complex object, enclose the object's attributes in curly braces, as shown in the "Structure Format Specifier Syntax" on page 29.

The following example creates a public server group named `AllMine`. The `create` method has a single parameter, `pattern`, which encloses the `parent` and `shortName` attributes in curly braces. In this example, `getPublicRoot` returns 2340555, the ID of the top public group.

```
cd /opsw/api/com/opsware/device/DeviceGroupService/method
./.getPublicRoot:i ; echo
./create "pattern={ parent:i=2340555 shortName='AllMine' }"
```

Specify array parameters by repeating the parameter name, once for each array element. For example, the following invocation of the `assign` method specifies the first two elements in the array parameter named `policies`:

```
cd /opsw/api/com/opsware/swmgmt
cd SoftwarePolicyService/method
./attachPolicies self:i=4220039 \
policies:i=4400335 policies:i=4400942
```

Overloaded Methods

A Java method name is overloaded if multiple methods in the same class have the same name but different parameter lists. With overloaded OCLI methods, the argument names on the command-line indicate which method to invoke. The `setCustomField` method, for example, is overloaded to support the setting of different data types. The following two commands invoke different versions of the method:

```
./setCustomField \
fieldName="Service Agreement" strValue="Gold"
./setCustomField \
fieldName=hmp longValue=2245
```

Return Values

If the API method underlying an OCLI method returns a value, then the OCLI method outputs the value to `stdout`. As with Unix commands, you can redirect a method's `stdout` to a file or assign it to an environment variable.

To change the representation of the return value, insert a leading period and append a format specifier to the method name. The following example returns server references as IDs, instead of the default names:

```
cd /opsw/api/com/opsware/server/ServerService/method
./.findServerRefs:i
```

If you indicate a format specifier that is incompatible with the method's return type, the file system responds with an error.

Exit Status

Like Unix shell commands, OCLI methods use the exit status (`$?`) to indicate the result of the call. An exit status of zero indicates success; a non-zero indicates an error. OCLI methods output error messages to `stderr`.

Table 2-3: Exit Status Codes for OCLI Methods

EXIT STATUS	CATEGORY	DESCRIPTION
0	Success	The method completed successfully.
1	Command-Line Parse Error	The command-line for the method call is malformed and could not be parsed into a set of options (--option[=value]) and parameter values (param=value).
2	Parameter Parse Error	The parameter values could not be parsed into the object types required by the API.
3	API Usage Error	The call failed because of a usage error, such as an invalid parameter value.
4	Access Error	The user does not have permission to perform the operation.
5	Other Error	An error occurred other than those indicated by exit statuses 1- 4.

For example, the following `bash` script checks the exit status of the `getDeviceGroups` method:

```
#!/bin/bash

cd /opsw/Server/@/toro.snv1.corp.opsware.com/method
./getDeviceGroups
cmnd_exit_status=$?

if [ $cmnd_exit_status -eq 0 ]
then
  echo "The command was successful."
else
  echo "The command failed."
  echo "Exit status = "  $cmnd_exit_status
fi
```

An OCLI method invokes an underlying API method. If the API method throws an exception, the OCLI method returns a non-zero exit status. When debugging a method call, you might find it helpful to view information about a thrown exception. The `/sys/last-exception` file in the OGFS contains the stack trace of an exception thrown by the most recent API call. After this file has been read, the system discards the file contents.

Search Filters and OCLI Methods

Many methods in the Opsware API accept object references as parameters. To retrieve object references based on search critera, you invoke methods such as `findServerRefs` and `findJobRefs`. For example, you can invoke `findServerRefs` to search for all servers that have `opsware.com` in the `hostname` attribute.

Search Syntax

Methods such as `findServerRefs` have the following syntax:

`find`*`object`*`Refs filter='[`*`object-type`*`:]`*`expression`*`'`

The `filter` parameter includes an expression, which specifies the search criteria. You enclose an expression in either parentheses or curly brackets. A simple expression has the following syntax:

`value-object.attribute operator value`

(This syntax is simplified. For the full definition, see “Filter Grammar” on page 55)

Search Examples

Most of the SAS object types have associated finder methods. This section shows how to use just a few of them. To see how searches are used with other OCLI methods, see "Example Scripts" on page 44.

Finding Servers

Find servers with host names containing `opsware.com`:

```
cd /opsw/api/com/opsware/server/ServerService/method
./.findServerRefs:i \
filter='device:{ ServerVO.hostname CONTAINS opsware.com }'
```

Find servers with a use attribute value of either `UNKNOWN` or `PRODUCTION`:

```
cd /opsw/api/com/opsware/server/ServerService/method
./.findServerRefs:i \
filter='{ ServerVO.use IN "UNKNOWN" "PRODUCTION" }'
```

The following `bash` script shows how to search for servers, save their IDs in a temporary file, and then specify each ID as the parameter of another method invocation. This script displays the public groups that each Linux server belongs to.

```
#!/bin/bash

TMPFILE=/tmp/server-list.txt
rm -f $TMPFILE

cd /opsw/api/com/opsware/server/ServerService/method

./.findServerRefs:i \
filter='{ ServerVO.osVersion CONTAINS Linux }' > $TMPFILE

for ID in `cat "$TMPFILE"`
do
  echo Server ID: $ID
  ./getDeviceGroups self:i=$ID
  echo
done
```

Finding Jobs

The examples in this section return the IDs of jobs such as server audits or policy remediations. The `job_status` field, a searchable attribute, can have the following values:

```
CANCELLED
ABORTED
```



```
SUCCESS
FAILURE
WARNING
ACTIVE
ZOMBIE
PENDING
UNKNOWN
```

Find the jobs that have completed successfully:

```
cd /opsw/api/com/opsware/job/JobService/method
./.findJobRefs:i filter='job:{ job_status = "SUCCESS" }'
```

Find the jobs that have completed successfully or with warning:

```
cd /opsw/api/com/opsware/job/JobService/method
./.findJobRefs:i \
filter='job:{ job_status IN "SUCCESS" "WARNING" }'
```

Find the jobs that have been started today:

```
cd /opsw/api/com/opsware/job/JobService/method
./.findJobRefs:i \
filter='job:{ JobInfoVO.startDate IS_TODAY "" }'
```

Find all server audit jobs:

```
cd /opsw/api/com/opsware/job/JobService/method
./findJobRefs \
filter='job:{ JobInfoVO.description = "Server Audit" }'
```

Find the jobs that have run on the server with the ID `280039`:

```
cd /opsw/api/com/opsware/job/JobService/method
./.findJobRefs:i filter='job:{ job_device_id = "280039" }'
```

Find today's jobs that have failed:

```
cd /opsw/api/com/opsware/job/JobService/method
./.findJobRefs:i \
filter='job:{ (( JobInfoVO.startDate IS_TODAY "" ) \
& ( job_status = "FAILURE" )) }'
```

Finding Other Objects

This section has examples that search for software policies and packages.

Find the software policies created by the Opsware user `jdoe`:

```
cd /opsw/api/com/opsware/swmgmt/SoftwarePolicyService/method
./.findSoftwarePolicyRefs:i \
filter='{ SoftwarePolicyVO.createdBy CONTAINS jdoe }'
```

Find the MSIs with `ismtool` for the Windows 2003 platforms:

```
cd /opsw/api/com/opsware/pkg/UnitService/method
./.findUnitRefs:i \
filter='software_unit:{ ((UnitVO.unitType = "MSI") \
& ( UnitVO.name contains "ismtool" ) \
& ( software_platform_name = "Windows 2003" )) }'
```

Searchable Attributes and Valid Operators

Not every attribute of a value object can be specified in a search filter. For example, you can search on `ServerVO.use` but not on `ServerVO.OsFlavor`.

To find out which attributes are searchable for a given object type, invoke the `getSearchableAttributes` method. The following example lists the attributes of `ServerVO` that can be specified in a search expression:

```
cd /opsw/api/com/opsware/search/SearchService/method
./getSearchableAttributes searchableType=device
```

The `searchableType` parameter indicates the object type. To determine the allowed values for `searchableType`, enter the following commands:

```
cd /opsw/api/com/opsware/search/SearchService/method
./getSearchableTypes
```

To find out which operators are valid for an attribute, invoke the `getSearchableAttributeOperators` method. The following example lists valid operators (such as `CONTAINS` and `IN`) for the attribute `ServerVO.hostname`:

```
cd /opsw/api/com/opsware/search/SearchService/method
./getSearchableAttributeOperators searchableType=device \
searchableAttribute=ServerVO.hostname
```

Example Scripts

This section has code listings for simple `bash` scripts that invoke a variety of OCLI methods. These scripts demonstrate how to pass method parameters on the command-line, including complex objects and the `self` parameter. If you decide to copy and paste these example scripts, you will need to change some of the hardcoded object names, such as the `d04.opsware.com` server. For tutorial instructions on creating and running scripts within the OGFS, see step 10 on page 25.

Of the following scripts, the most interesting is `remediate_policy.sh` on page 48. It creates a software policy, adds a package to the policy, and in the last line, installs the package on a managed server by invoking the `startFullRemediateNow` method.

create_custom_field.sh

This script creates a custom field (virtual column), named `TestFieldA` attaches the field to all servers, and then sets the value of the field on a single server. Until it is attached, the custom field does not appear in the SAS Web Client. You can create custom fields for servers, device groups, or software policies. To create a custom field, your Opsware user must belong to a user group with the Manage Virtual Columns permission (new in 6.0.1).

Unlike a custom attribute, a custom field applies to all instances of a type. For an example that creates a custom attribute in the OGFS, see "Managing Custom Attributes" in the *Opsware® SAS User's Guide: Server Automation*.

The `create_custom_field.sh` script has the following code:

```
#!/bin/bash
# create_custom_field.sh

cd /opsw/api/com/opsware/custattr/VirtualColumnService/method

# Create a virtual column.
# Remember the name because you cannot search for the
# displayName.
./create vo='{ name=TestFieldA type=SHORT_STRING \
displayName="Test Field A" }'

column_id=`./.findVirtualColumn:i name=TestFieldA`

echo --- column_id = $column_id

cd /opsw/api/com/opsware/server/ServerService/method

# Attach the column to all servers.
# All servers will have this custom field.
./attachVirtualColumn virtualColumn:i=$column_id

# Get the ID of the server named d04.opsware.com
devices_id=`./.findServerRefs:i \
filter=\
'device:{ ServerVO.hostname CONTAINS "d04.opsware.com" }'`

echo --- devices_id = $devices_id
```

```
# Set the value of the custom field (virtual column) for
# a specific server.
./setCustomField self:i=$devices_id fieldName=TestFieldA \
strValue="This is something."
```

create_device_group.sh

This script creates a static device group and adds a server to the group. Next, the script creates a dynamic group, sets a rule on the group, and refreshes the membership of the group. The last statement of the script lists the devices that belong to the dynamic group.

Here is the script's code:

```
#!/bin/bash
# create_device_group.sh

cd /opsw/api/com/opsware/device/DeviceGroupService/method

# Get the ID of the public root group (top of hierarchy).
public_root=`./.getPublicRoot:i`

# Create a public static group.
./create "vo={ parent:i=$public_root shortName='Test Group A' }"

# Get the ID of the group just created.
group_id=`./.findDeviceGroupRefs:i \
filter='{ DeviceGroupVO.shortName = "Test Group A" }' `

echo --- group_id = $group_id

cd /opsw/api/com/opsware/server/ServerService/method

# Get the ID of the server named d04.opsware.com
devices_id=`./.findServerRefs:i \
filter=\
'device:{ ServerVO.hostname CONTAINS "d04.opsware.com" }'`

echo --- devices_id = $devices_id

cd /opsw/api/com/opsware/device/DeviceGroupService/method

# Add a server to the device group.
./addDevices \
self:i=$group_id devices:i=$devices_id

# Create a dynamic device group.
./create \
```



```
"vo={ parent:i=$public_root \
shortName='Test Dyn B' dynamic=true }"

# Get the ID of the device group.
dynamic_group_id=`./.findDeviceGroupRefs:i \
filter='{ DeviceGroupVO.shortName = "Test Dyn B" }' `

echo --- dynamic_group_id = $dynamic_group_id

# Set the rule so that this group contains servers with
# hostnames containing the string opsware.com.
# The rule parameter has the same syntax as the filter
# parameter of the find methods.
./setDynamicRule self:i=$dynamic_group_id \
rule='device:{ ServerVO.hostname CONTAINS opsware.com }'

# By default, membership in dynamic device groups is refreshed
# once
# an hour, so force the refresh now.
./refreshMembership selves:i=$dynamic_group_id now=true

# Display the names of the devices that belong to the group.
echo --- Devices in group:
./getDevices selves:i=$dynamic_group_id
```

create_folder.sh

This script creates a folder named `/Test 1`, lists the folders under the root (/) folder, and then creates the subfolder `/Test 1/Test 2`. After creating these folders, you can view them under the Library in the navigation pane of the SAS Client.

Here is the code for this script:

```
#!/bin/bash
# create_folder.sh

cd /opsw/api/com/opsware/folder/FolderService/method

# Get the ID of the root (top) folder.
root_id=`./.getRoot:i`

# Create a new folder under the root folder.
./create members="{ name='Test 1' folder:i=$root_id }"

# Display the names of the folders under the root folder.
./getChildren self:i=$root_id
```

```
# Get the ID of the folder named "Test 1"
folder_id=`./.findFolders:i \
filter='{ FolderVO.name = "Test 1" }'`

# Create a subfolder.
./create members="{ name='Test 2' folder:i=$folder_id }"

folder_id=`./.findFolders:i \
filter='{ FolderVO.name = "Test 2" }'`
```

remediate_policy.sh

This script creates a software policy named `TestPolicyA` in an existing folder named `Test 2`, adds a package containing `ismtool` to the policy, attaches the policy to a single server (not a group), and then remediates the server. The remediation action launches a job that installs the package onto the server. You can check the progress and results of the job in the SAS Client. For examples that search for jobs with OCLI methods, see "Finding Jobs" on page 42.

In this script, in the `create` method of the `SoftwarePolicyService`, the value of the `platforms` parameter is hardcoded. In most of these example scripts, hardcoding is avoided by searching for an object by name. In the case of platforms, searching by the `name` attribute is difficult because if differs from the `displayName` attribute, which is exposed in the SAS Client but is not searchable. The easiest way to find a platform ID is by going to the twister and running the `PlatformService.findPlatformRefs` method with no parameters.

The `update` method in this script hardcodes the ID of `softwarePolicyItems`, an object that can be difficult to search for by name if the Software Repository contains many packages with similar names. One way to get the ID is to run the SAS Client, search for Software by fields such as File Name and Operating System, open the package located by the search, and note the Opsware ID in the properties view of the package.

In the following listing, the `update` method has a bad line break. If you copy this code, edit the script so that the `vo` parameter is on a single line.

Here is the source code for the `remediate_policy.sh` script:

```
#!/bin/bash
# remediate_policy.sh
```



```
# Get the ID of the folder where the policy will reside.
cd /opsw/api/com/opsware/folder/FolderService/method
folder_id=\
`./.findFolders:i filter='{ FolderVO.name = "Test 2" }'`

cd /opsw/api/com/opsware/swmgmt/SoftwarePolicyService/method

# Create a software policy named TestPolicyA.
# This policy resides in the folder located in the preceding
# findFolders call.
# The platform for this policy is Windows 2003 (ID 10007)
./create softwarePolicyVO="{ platforms:i=10007 \
name="TestPolicyA" \
folder:i=$folder_id }"

policy_id=`./.findSoftwarePolicyRefs:i \
filter='{ SoftwarePolicyVO.name = "TestPolicyA" }'`

echo --- policy_id = $policy_id

# Call the update method to add a package to the software
# policy. The package ID is 4230039.
#
# NOTE: The following command has a bad line break.
# The vo parameter should be on a single line.
#
./update self:i=$policy_id force=true\
# The next 2 lines should be on a single line.
vo='{
softwarePolicyItems:i=com.opsware.pkg.windows.MSIRef:4230039 }'

cd /opsw/api/com/opsware/server/ServerService/method

# Get the ID of the server named d04.opsware.com
devices_id=`./.findServerRefs:i \
filter='device:{ ServerVO.hostname CONTAINS "d04.opsware.com"
}'`

echo --- devices_id = $devices_id

# Attach the policy to a single server (not a group).
./attachPolicies self:i=$devices_id \
policies:i=$policy_id

# Remediate the server to install the package in the policy.
job_id=`./.startFullRemediateNow:i self:i=$devices_id`
```

```
echo --- job_id = $job_id
```

remove_custom_field.sh

Although not common in an operational environment, removing custom fields is sometimes necessary in a testing environment. Note that a custom field must be unattached before it can be removed.

Here is the code for `remove_custom_field.sh`:

```
#!/bin/bash
# remove_custom_field.sh

if [ ! -n "$1" ]
   then
   echo "Usage: `basename $0` <name>"
   echo "Example: `basename $0` hmp"
   exit
fi

cd /opsw/api/com/opsware/custattr/VirtualColumnService/method

column_id=`./.findVirtualColumn:i name=$1`

echo --- column_id = $column_id

cd /opsw/api/com/opsware/server/ServerService/method

# Column must be detached before it can be removed.
./detachVirtualColumn virtualColumn:i=$column_id

cd /opsw/api/com/opsware/custattr/VirtualColumnService/method

# Remove the virtual column.
./remove self:i=$column_id
```

schedule_audit_task.sh

This script starts an audit task, scheduling it for a future date. With OCLI methods, date parameters are specified with the following syntax:

```
YYYY/MM/DD HH:MM:SS.sss
```

The method that launches the task, `startAudit`, returns the ID of the job that performs the audit. For examples that search for jobs with OCLI methods, see "Finding Jobs" on page 42.

Here is the code for `schedule_audit_task.sh`:


```
#!/bin/bash
# schedule_audit_task.sh

cd /opsw/api/com/opsware/compliance/sco/AuditTaskService/method

# Get the ID of the audit task to schedule.
audit_task_id=`./.findAuditTask:i \
filter='audit_task:{ \
(( AuditTaskVO.name BEGINS_WITH "HW check" ) \
& ( AuditTaskVO.createdBy = "gsmith" )) }'`

echo --- audit_task_id = $audit_task_id

# Schedule the audit task for Oct. 17, 2008.
# In the startDate parameter, note that the last delimiter for
# the time is a period, not a colon.
job_id=`./startAudit self:i=140039 \
schedule:s='{ startDate="2008/10/17 00:00:00.000" }' \
notification:s='{ onFailureOwner="sjones@opsware.com" \
onFailureRecipients="jdoe@opsware.com" \
onSuccessOwner="sjones@opsware.com" \
onSuccessRecipients="jdoe@opsware.com" }'`

echo --- job_id = $job_id
```

Getting Usage Information on OCLI Methods

In a future release, the OCLI methods will display usage information. Until then, you can get the necessary information from the API documentation or the OGFS with the techniques described in the following sections.

Listing the Services

The Opsware API methods are organized into services. To find out what services are available for OCLI methods, enter the following commands in a Global Shell session:

```
cd /opsw/api/com/opsware
find . -name "*Service"
```

To list the services in the API documentation, specify the following URL in your browser:

```
https://occ_host:1032
```

The *occ_host* is the IP address or host name of the core server running the Opsware Command Center component.

Finding a Service in the API Documentation

The path of the service in the OGFS maps to the Java package name in the API documentation. For example, in the OGFS, the `ServerService` methods appear in the following directory:

```
/opsw/api/com/opsware/server
```

In the API documentation, the following interface defines these methods:

```
com.opsware.server.ServerService
```

Listing the Methods of a Service

In the OGFS, you can list the contents of the `method` directory of a service, For example, to display the method names of the `ServerService`, enter the following command:

```
ls /opsw/api/com/opsware/server/ServerService/method
```

In the API documentation, perform the following steps to view the methods of `ServerService`:

1. In the upper left pane, select `com.opsware.server`.
2. In the lower left pane, select `ServerService`.
3. In the main pane, scroll down to view the methods.

Listing the Parameters of a Method

In the API documentation, perform the steps described in the preceding section. In the Method Detail section of the service interface page, view the parameters and return types. (For more information about method parameters, see "Passing Arguments on the Command-Line" on page 38.)

Getting Information About a Value Object

The API documentation shows that some service methods pass or return value objects (VOs), which contain data members (attributes). For example, the `ServerService.getServerVO` method returns a `ServerVO` object. To find out what attributes `ServerVO` contains, perform the following steps:

1. In the API documentation, select the `ServerVO` link. You can find the this link in several places:
   - The method signature for `getServerVO`
   - The list of classes (lower left pane) for `com.opsware.server`

- On the Index page. A link to the Index page is at the top of the main pane of the API documentation.

2. On the `ServerVO` page, note the getter and setter methods. Each getter-setter pair corresponds to an attribute contained in the value object. For example, `getCustomer` and `setCustomer` indicate that `ServerVO` contains an attribute named `customer`.

Determining If an Attribute Can Be Modified

Only a few object attributes can be modified by client applications. To find out if an attribute can be modified, perform the following steps:

1. In the API documentation, go to the value object page, as described in the preceding section.
2. In the Method Detail section of the setter method, look for "Field can be set by clients."

For Opsware SAS objects represented in the OGFS, such as servers and customers, you can determine which attributes are modifiable by checking the access types of the files in the `attr` directory. The files that have read-write (`rw`) access types correspond to modifiable attributes. For example, to list the modifiable attributes of a server, enter the following commands:

```
cd /opsw/Server/@/server-name/attr
ls -l | grep rw
```

Determining If an Attribute Can Be Used in a Filter Query

To find out if an attribute of a value object can be used in a filter query (a search), perform the following steps:

1. In the API documentation, go to the value object page.
2. In the Method Detail section of the getter method that corresponds to the attribute, look for the string, "Field can be used in a filter query."

From within a Global Shell session, to find out if an attribute can be searched on, follow the techniques described in "Searchable Attributes and Valid Operators" on page 44

Appendix A: Search Filter Syntax

IN THIS APPENDIX

This appendix discusses the following topics:

- Filter Grammar
- Usage Notes

Filter Grammar

A search filter is a parameter for methods such as `getServerRefs`. The formal syntax for a filter follows:

```
<filter>                ::= (<expression-junction>)+

<expression-junction>   ::= <expression-list-open> <junction>
   (<expression>)+ <expression-list-close>

<expression>            ::= <expression-open> <attribute>
   <general-delimiter> <operator> <general-delimiter>
   <value-list> <expression-close>

<attribute>             ::= <resource_field>
<vo_member>             ::= <text>
<resource_field>        ::= <text>
<value-list>            ::= (<double-quote> <text> <double-
quote>)* | (<number>)*

<text>                  ::= [a-z] [A-Z] [0-9]
<number>                ::= [0-9] [.]

<junction>              ::= <union-junction> |
   <intersect-junction>
<union-junction>        ::= '|'
<intersect-junction     ::= '&'
<expression-list-open>  ::= '('
<expression-list-close> ::= ')'
<expression-open>       ::= '(' | '{'
```

```
<expression-close>        ::= '(' | '}'
<general-delimiter>       ::= <whitespace>
<whitespace>              ::= ' '
<double-quote>            ::= '"'
<escape-character>        ::= '\'

<operator>               ::= <equal_to> |...| <contains_or_above>
```

Valid operators for the preceding line:

```
<equal_to>                ::= '='     | 'EQUAL_TO'
<not_equal_to>            ::= '!='    | '<>' | 'NOT_EQUAL_TO'
<in>                      ::= '='     | 'IN'
<not_in>                  ::= '!='    | '<>' | 'NOT_IN'
<greater_than>            ::= '>'     | 'GREATER_THAN'
<less_than>               ::= '<'     | 'LESS_THAN'
<greater_than_or_equal>   ::= '>='    | 'GREATER_THAN_OR_EQUAL'
<less_than_or_equal>      ::= '<='    | 'LESS_THAN_OR_EQUAL'
<begins_with>             ::= '=*'    | 'BEGINS_WITH'
<ends_with>               ::= '*='    | 'ENDS_WITH'
<contains>                ::= '*=*'   | 'CONTAINS'
<not_contains>            ::= '*<>*'  | 'NOT_CONTAINS'
<in_or_below>             ::= 'IN_OR_BELOW'
<in_or_above>             ::= 'IN_OR_ABOVE'
<between>                 ::= 'BETWEEN'
<not_between>             ::= 'NOT_BETWEEN'
<not_begins_with>         ::= 'NOT_BEGINS_WITH'
<not_ends_with>           ::= 'NOT_ENDS_WITH'
<is_today>                ::= 'IS_TODAY'
<is_not_today>            ::= 'IS_NOT_TODAY'
<within_last_days>        ::= 'WITHIN_LAST_DAYS'
<within_last_months>      ::= 'WITHIN_LAST_MONTHS'
<within_next_days>        ::= 'WITHIN_NEXT_DAYS'
<within_next_months>      ::= 'WITHIN_NEXT_MONTHS'
<not_within_last_days>    ::= 'NOT_WITHIN_LAST_DAYS'
<not_within_last_months>  ::= 'NOT_WITHIN_LAST_MONTHS'
<not_within_next_days>    ::= 'NOT_WITHIN_NEXT_DAYS'
<not_within_next_months>  ::= 'NOT_WITHIN_NEXT_MONTHS'
<contains_or_below>       ::= 'CONTAINS_OR_BELOW'
<contains_or_above>       ::= 'CONTAINS_OR_ABOVE'
```

Usage Notes

The same junction type must be used within each expression junction:

- valid: `((x = y) & (a = y) & (x = a))`

- invalid: `((x = y) & (a = y) | (x = a))`

A text value needs to have double-quotes surrounding it but a number does not. Any double-quote that is part of the value must be escaped with a backslash:

- valid number: `123.456`
- valid text: `"abc"`
- invalid text: `abc`
- valid text: `"ab\"c"`
- invalid text: `"ab"c"`
- invalid text: `ab"c`

Parentheses must surround groups of expressions which will junction with another group of expressions:

- valid grouping: `((x = y) & (a = b)) | (n = r)`
- invalid grouping: `(x = y) & (a = b) | (n = r)`