Remote Administration Daemon Client User's Guide

E68270-03
August 2023

Remote Administration Daemon Client User's Guide,

E68270-03

Primary Author: Veach Sharon, Cathleen Reiher

Contents

Using This Documentation

Product Documentation Library

Documentation and resources for this product and related products are available at http://www.oracle.com/pls/topic/lookup?ctx=E37838-01.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

1

Introduction to the Remote Administration Daemon

What's New in the RAD API in Oracle Solaris 11.4

The RAD authentication API is significantly different in Oracle Solaris 11.4.

- The authentication API moved from version 1.0 to version 2.0, which required changes to the client code that uses the API. Some of the source code changes are small, such as using a different header file, library, Python module, or Java jar file. Some changes are large, such as the authentication API now interacts with the RAD HTTP/REST APIs directly. These changes are incompatible with the Oracle Solaris 11.3 authentication API.
- RAD C client bindings support only 64-bit RAD clients and modules.
- Ensure that you use the RAD client bindings for the latest version of Python, which is Python 3.7.

  The community is ending support for Python 2.7 and Python 3.5, so the RAD client bindings for these older Python versions will no longer work.

Remote Administration Daemon

RAD provides programmable interfaces that enable developers and administrators to configure and manage Oracle Solaris system components. You can configure and manage system components using C, Java, Python, and REpresentational State Transfer (REST) APIs. RAD also enables developers to create custom interfaces using these APIs to manage the system components.

RAD is designed to provide a remote administrative interface for operating system components or subsystems. The remote interfaces support easy administration of a distributed systems. However, RAD interfaces are not intended to build distributed systems. You can use RPC, RMI, CORBA, MPI, and other technologies to build distributed applications.

A RAD interface defines how a client can interact with a system through a set of methods, attributes, and events using a well-defined namespace.

Developers and administrators, who previously used *$EDITOR* can now use one the following approaches to modify system components locally:

- Using a command-line interface (CLI) or an interactive user interface (UI)
- Using a browser or a remote client
- Using a CLI, an interactive UI, and a browser or a client with an enterprise-scale provisioning tool

All of these methods require programmable access to configuration.

RAD uses a client-server design to support different types of clients such as clients written in different languages, clients running without privilege, and clients running remotely. In a client-server design, RAD acts as a server that services remote procedure calls and clients act as consumers.

By providing a procedure call interface, RAD enables non-privileged local consumers to perform actions on behalf of their users that require elevated privilege, without resorting to a CLI-based implementation. By establishing a stream protocol, RAD enables the consumers to perform actions on any system or device over a range of secure transport options.

The `rad` protocol is efficient and easy to implement, which makes it simple to support all administrative tasks provided by an interface. The protocol used by RAD is efficient and is easy to implement.

RAD differs from remote procedure call (RPC) in the following ways:

- Procedure calls in RAD are made against server objects in a browsable, structured namespace. This process permits a more logical progression of program than central allocation of program numbers.
- Procedure calls can be asynchronous. Depending on the protocol in use, a client might have multiple simultaneous outstanding requests.
- You can inspect and modify the interfaces exported by the server objects. This inspection facilitates interactive usage, debugging environments, and enables clients to use dynamically-typed languages such as Python.
- Using RAD interfaces, you can define properties and asynchronous event sources.

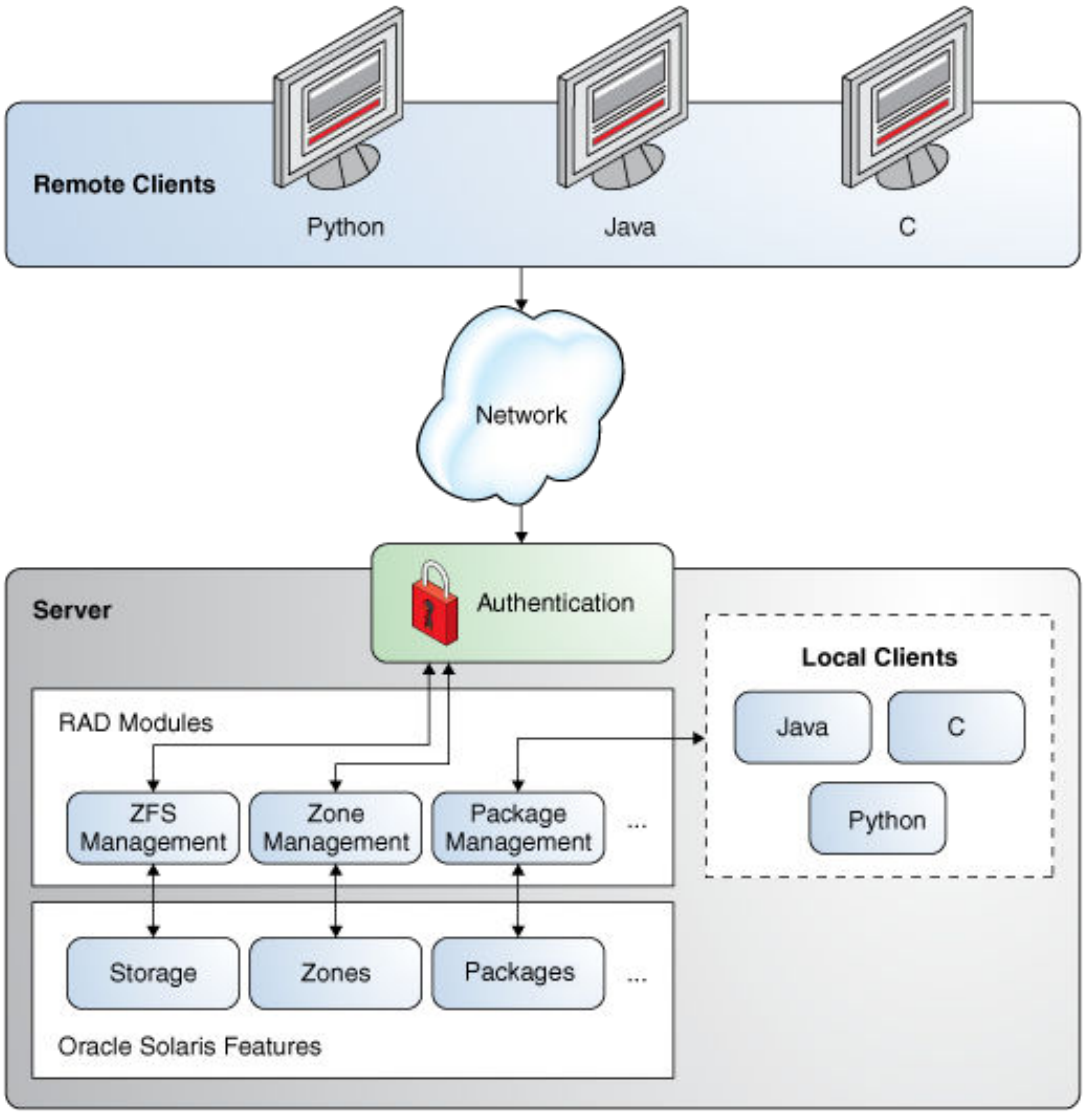
> **Note:**
>
> The native RAD protocol supports asynchronous procedure calls after the client is authenticated. The underlying implementation of other protocols, such as XML-RPC, might not support asynchronous calls.

How RAD Works

In RAD architecture, the clients can be local or remote, and the clients interact with the RAD modules to perform various administrative activities. For example, a client interacts with the ZFS management RAD module to perform storage-related activities. These RAD clients can be written in C, Java, or Python.

The following figure shows the architecture of RAD.

Architecture of RAD



RAD uses three different types of authentication:

- Implicit – Used for local connections, for example, using `rc_connect_unix(NULL, ...)`. The RAD daemon reads the user's credentials from the calling process.
- Explicit with PAM – Performed by using APIs such as `rc_auth_login`() and `rc_connect_uri`() in C client, `rad.auth.RadAuth` class in Python, and `RadAuthHandler` in Java.
- Explicit with GSS – Performed by using APIs `rc_connect_gss`() or `rc_connect_uri`() in C client. This requires Kerberos setup. For more information about using Kerberos with RAD, see Generic Security Services API Transport in RAD.

To use the client bindings you must install the appropriate client program packages.

- For C – `solaris/system/management/rad/client/rad-c`
- For Java – `solaris/system/management/rad/client/rad-java`

- For Python – `solaris/system/management/rad/client/rad-python-37`, `solaris/system/management/rad/client/rad-python-35`, or `solaris/system/management/rad/client/rad-python-27`, depending on the Python version required

To use `radadrgen` to generate API-specific language client and server bindings, see radadrgen Processing Tool in *Remote Administration Daemon Module Developer's Guide*.

Tips for Using RAD Modules

- Man pages for the RAD modules are available for C client bindings and for Python.
  - For C, the man pages are in section 3RAD. For example, to view the man page of the `com.oracle.solaris.rad.kstat` module, type:

    ```
    $ man -s 3RAD kstat
    ```

  - For Python man pages, use the `pydoc` command. Prefix the module name with `rad.bindings`, and include the API version. For example, to view the man page of the `com.oracle.solaris.rad.kstat` module, type:

    ```
    $ pydoc3.7 rad.bindings.com.oracle.solaris.rad.kstat_
    ```

    > **Note:**
    >
    > The 3RAD man pages include the API version of each RAD module.

- Some operations, such as adding a new user, require additional privileges. You must ensure that the user has the appropriate rights profiles to execute the operation. For more information, see Assigning Rights to Users in *Securing Users and Processes in Oracle Solaris 11.4*.
- By default, RAD log messages are available in the `/var/svc/log/system-rad:local.log` file. To enable logging of debug messages, set the `config/debug` property of the RAD SMF instance and restart the instance:

  ```
  # svccfg -s rad setprop config/debug=true
  # svcadm refresh rad:local
  # svcadm restart rad:local
  ```

Configuring RAD Transports

This section covers how to configure RAD transports:

- Configuring RAD Transports to Accept X.509 Client Certificates
- Configuring the RAD Transport to Specify a Particular IP Address

Configuring RAD Transports to Accept X.509 Client Certificates

The `rad` daemon permits you to configure transports to accept X.509 (`x509`) client certificates that authenticate the client holder of the certificate as a user (`root` or privileged user) on the server system.

You can configure the RAD TLS transport by configuring properties in the `ssl_port` property group of the `svc:/system/rad:remote` SMF instance.

Use the following configuration options to configure the RAD TLS transport:

`allow_client_certificate`
Specifies whether to permit the clients to authenticate by using an X.509 client certificate. The default value is `true`.
The certificate must be signed by a specific CA, which defaults to the one specified by `client_ca_path`. If the certificate contains `UID=` *logname* in the Subject and user *logname* exists, the RAD daemon authenticates the connection to that user.

`client_ca_path`
Specifies the location of the PEM-formatted file that includes a CA certificate with which all client X.509 certificates must be signed. The value defaults to the `certificate/ca/uri` property value of the `svc:/system/identity:cert` SMF instance.

`map_host_certificate_to_root`
Specifies whether to permit mapping a client X.509 certificate to the `root` user. The default value is `false`.
If the value is `true` and the client X.509 does not have a UID set in the Subject, the RAD daemon determines whether the network peer host is listed in Subject CommonName or in Subject Alternative Name. If the RAD (`rad`) daemon finds the network peer host, the RAD daemon authenticates the connection as the `root` user.

`require_client_certificate`
Specifies whether all clients authenticate by using an X.509 client certificate. The default value is `false`.

The following shell commands show how to enable the client certificates to map to the `root` user. This capability is useful when the client program is an HTTP client that can send TLS client certificates and the HTTP client uses the RAD HTTP/REST interface. Map the host certificate to the `root` user for the `rad:remote` service instance by setting the `https_port/map_host_certificate_to_root` property value to `true` as follows:

```
# svccfg -s rad:remote setprop https_port/map_host_certificate_to_root = boolean: true
# svcadm refresh rad:remote
# svcadm restart rad:remote
```

Configuring the RAD Transport to Specify a Particular IP Address

The RAD daemon uses the TCP, TLS, and GSS transports to listen and service incoming connections. By default, these transports permit you to configure a port on which the daemon listens for any address on the particular system.

In addition, the RAD daemon permits you to use the `addr` option to fine-tune a RAD transport configuration to bind to a specific IP address and to specify whether the connection from the client comes over a public or a private network interface. The IP address can be a host name or a network address.

When the RAD daemon binds to a specific address, you can separate traffic on public and private networking interfaces. For example, you might want to support different settings such as certificate and pam_service.

You can configure this behavior by modifying the existing rad:remote SMF service.

For example, you can use the addr option to distinguish between connections coming over private and public network interfaces and configure specific certificate and pam_service settings for each interface.

Example 1-1 Using the addr Option to Specify IP Addresses for the TCP Transport

The following command shows you how to configure the tcp transport to use the 192.168.18.18, 192.168.18.48, and host1.example.com IP addresses:

```
# svccfg -s rad:remote setprop https_port/addr = host: {192.168.18.18
192.168.18.48 host1.example.com}
# svcadm refresh rad:remote
# svcadm restart rad:remote
```

The following commands configure an additional RAD transport in the `rad:remote` SMF instance to listen on addresses `10.0.0.10` and that of `system1` on port `9999`:

```
# svccfg -s rad:remote
svc:/system/rad:remote> addpg tls_port xport_tls
svc:/system/rad:remote> select tls_port
svc:/system/rad:remote> setprop tls_port/addr=host: (10.0.0.10 system1)
svc:/system/rad:remote> setprop tls_port/port=9999
svc:/system/rad:remote> setprop tls_port/pam_service=rad-tls
svc:/system/rad:remote> setprop tls_port/certificate=/etc/certs/localhost/
host.crt
svc:/system/rad:remote> setprop tls_port/privatekey=/etc/certs/localhost/host.key
svc:/system/rad:remote> setprop tls_port/proto=rad
# svcadm refresh rad:remote
# svcam restart rad:remote
```

2
Connecting to RAD

RAD C Client

The public interfaces that are not specific to RAD modules, are exported in the `/usr/lib/libradclient.so` library and are defined in the following headers:

- `/usr/include/rad/radclient.h` – The client function and datatype definitions
- `/usr/include/rad/radclient_basetypes.h` – Helper routines for managing the built-in RAD types

The list of `#include` statements at the beginning of each example shows the headers that are required for that specific functionality.

Building RAD C Clients

A C client program must include the `<rad/radclient.h>` header file. To authenticate a connection to a remote instance by using the `rc_auth_login`() function, the client application must use the `#include <rad/client/c/2/auth_login.h>` header. To include signatures of functions to interact with a RAD module, the client application must use the `#include </rad/client/c/<module_version>/<module_name>.h>` header.

Connecting to RAD in C

RAD instances establish connections by using the `rc_connect_*`() set of functions. You can obtain connections for various transports such as TLS, TCP, and local UNIX socket. Each function returns a `rc_conn_t` reference. This reference acts as a handle for interactions with RAD over its connection. Every connect function has one common argument, a *locale* to use for the connection. When *locale* is `NULL`, the locale of the local client is used.

To close the connection, you must call the `rc_disconnect`() function with the connection handle.

Connecting to a Local RAD Instance in C

You can connect to a local instance using the `rc_connect_unix`() function. An implicit authentication is performed against your user ID and most RAD tasks that you request with this connection are performed with the privileges available to your user account.

The `rc_connect_unix`() function takes the following arguments:

- A string, path of the UNIX socket
- A string, locale for the connection

If the value of socket path is `NULL`, the default RAD UNIX socket path is used. If the value of *locale* is `NULL`, the locale of the local client is used.

Example 2-1 C Language – Creating a RAD Local Connection

```
#include <rad/radclient.h>
rc_conn_t conn = rc_connect_unix(NULL, NULL);
```

Connecting to a Remote Instance and Authenticating in C Using RAD

When connecting to a remote instance, no implicit authentication is performed. The connection is not established until you authenticate. You can authenticate a connection to a remote instance by using the `rc_auth_login`() function. The client application must use the `#include <rad/client/2/auth_login.h>` header and link to the authentication module C binding library, `/usr/lib/rad/client/c/libauthentication2_client.so`.

Authentication is non-interactive, and a username and a password must be provided. Optionally, a handle to the PAM authentication object is returned if a reference is provided as the second argument to the `rc_auth_login`() function.

Example 2-2 C Language – Creating a RAD Remote Connection Over TCP IPv4 on Port 7777

```
#include <rad/radclient.h>
#include <rad/client/2/auth_login.h>

rc_instance_t *pam_inst;
rc_conn_t conn = rc_connect_tcp("host1",7777, NULL);

if (conn !=NULL) {
            rc_err_t status = rc_auth_login(conn, &pam_inst, "user", "password");
            if (status == RCE_OK){
                printf("Connected and authenticated!\n");
                }
}
```

RAD is deployed as two cooperating processes. A `proxy` process is responsible for authentication and establishing communications. A `slave` process is created by the proxy and handles module processing. A slave is created for each client connection.

Connecting to a RAD Instance by Using a URI in C

You can use a uniform resource identifier (URI) to connect to a local or remote RAD instance. For more information, see Connecting in Python to a RAD Instance by Using a URI.

The following functions are supported in C:

- `rc_uri_t *rc_alloc_uri(const char *src, rc_scheme_t schemes)`
- `rc_credentials_t *rc_alloc_pam_credentials(const char *pass)`
- `void rc_free_credentials(rc_credentials_t *cred)`
- `rc_credentials_class_t rc_uri_get_cred_class(rc_uri_t *uri)`
- `rc_uri_t *rc_alloc_uri(const char *src, rc_scheme_t schemes)`
- `rc_conn_t * rc_connect_uri(const char *uri, rc_credentials_t *cred)`
- `void rc_uri_set_cred_class(rc_uri_t *uri, rc_credentials_class_t class)`
- `rc_scheme_t rc_uri_get_schemes(rc_uri_t *uri)`

- `int rc_uri_get_port(rc_uri_t *uri)`
- `const char *rc_uri_get_host(rc_uri_t *uri)`
- `rc_scheme_t rc_uri_get_scheme(rc_uri_t *uri)`
- `const char *rc_uri_scheme_tostr(rc_scheme_t scheme)`
- `const char *rc_uri_get_src(rc_uri_t *uri)`
- `const char *rc_uri_get_user(rc_uri_t *uri)`
- `const char *rc_uri_get_path(rc_uri_t *uri)`
- `void rc_free_uri(rc_uri_t *uri)`

You can use the `rc_uri_t` structure to connect to a RAD instance. `rc_uri_t` is the main structure with which you interact.

You can allocate a `rc_uri_t` structure with the `rc_alloc_uri`() function. This function returns `NULL` on failure or a pointer to a valid `rc_uri_t` structure. For example, if you require PAM authentication for a remote connection, you must allocate a `rc_credentials_t` structure using one of the `alloc`() credential functions. This allocation depends on the authentication type. RAD supports two types of authentication, PAM and generic security service (GSS).

You can connect to RAD by using the `rc_connect_uri`() function. This returns a `rc_conn_t`() function that can be used to establish the connection by using `rc_connect_unix`(), `rc_connect_tcp`(), or other functions. You can use the other informative functions to interact with the allocated structure to obtain useful information. The various `rc_free_uri`() functions can clean the memory after you finish using the structures.

RAD Namespace in C

Most RAD objects are represented in the abstract data representation (ADR) document as interfaces. You can search RAD objects by searching the RAD namespace.

To access a RAD object, you need the following:

- A proxy, which can be used to search the RAD namespace. An interface proxy class allows you to use a proxy to search the RAD namespace. Interface proxy is defined in the binding module of each interface.
- The list and lookup functions (*module_interface*__`rad_list`() and *module_interface*__`rad_lookup`()) provided by client binding library of a module. These functions also provide the option to perform either strict or relaxed versioning.

> **Note:**
>
> These functions use a double underscore between *interface* and the following function.

As RAD interfaces become more extensive in Oracle Solaris, the probability of two clients interoperating increases when using the strict and relaxed API versioning. Each versioning mode is applicable on a per-interface and per-connection basis.

Relaxed versioning is the default mode for RAD communication. In this mode, RAD accepts connections for any minor version of an interface if the major version of the server module

matches the major version of the client binding. Relaxed versioning mode maximizes the cross-system compatibility of the interfaces. In strict versioning mode, RAD rejects the client unless the minor version of the module at the server is greater than or equal to the minor version of the client binding.

The proxy automatically provides the base name and the version details using functions for the interface instances. It is structured as follows:

```
<domain name>:type=<interface name>[,optional additional key value pairs]
```

The `<domain name>` and the `<interface name>` are automatically derived from the ADR interface description language (IDL) definition and are stored in the module binding.

Certain interfaces return or accept object references directly to or from clients. These objects might not be named. Objects that are not named are **anonymous**. Anonymous objects cannot be looked up in the RAD namespace, but the interface provides access methods that make it simple to interact with them.

Creating a Name for a RAD Object in C

The client is not required to create an object name, as *module_interface*__`rad_list`() creates it internally.

Searching for RAD Objects in C

Client binding of a module provides a search function for each interface defined in the form: *module_interface*__`rad_list`(). You can provide a pattern (`glob` or `regex`) to narrow the search within the objects of an interface type.

In addition, the `libradclient` library provides the function `rc_list`(), where the caller provides the entire name or pattern and version to search the objects.

Obtaining a Reference to a RAD Singleton in C

A module developer creates a singleton to represent an interface. This interface can be accessed easily. For example, the `zonemgr` module defines a singleton interface, **ZoneInfo**. It contains information about the zone that contains the RAD instance with which you are communicating.

Example 2-3 C Language – Obtaining a Reference to a RAD Singleton

```
#include <rad/radclient.h>
#include<rad/client/1/zonemgr.h>

rc_instance_t *inst;
rc_err_t status;
char *name;

rc_conn_t *conn = rc_connect_unix(NULL, NULL);
if (conn !=NULL) {
   status = zonemgr_ZoneInfo__rad_lookup(conn, B_TRUE, &inst, 0);
  if(status == RCE_OK) {
    status =zonemgr_ZoneInfo_get_name(inst, &name);
  if (status ==RCE_OK)
     printf("Zone name: %s\n", name);
   }
}
```

In the preceding example, you have connected to a local RAD instance, and have obtained a remote object reference directly using the lookup function provided by the `zonemgr` binding. After you have the remote reference, you can access the properties with the *module_interface*__`get_<property>`() function.

Listing RAD Instances of an Interface in C

An interface can contain multiple RAD instances. For example, the `zonemgr` module defines a `Zone` interface and an instance of this interface exists for each zone on the system. A module provides a list function for each of its interfaces in the form, *module_interface*__`rad_list`().

Example 2-4 C Language – Listing RAD Interface Instances

```
#include<rad/radclient.h>
#include<rad/radclient_basetypes.h>
#include<rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, NULL);
if (conn !=NULL) {
  status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_GLOB, &name_list,
    &name_count, 0);
   if(status == RCE_OK) {
      for (int i =0; i < name_count; i++) {
      char*name =adr_name_tostr(name_list[i]);
        printf("%s\n", name);
      }
     name_array_free(name_list, name_count);
     }
  }
```

Obtaining a Remote Object Reference From a Name in C

The list function returns a *name*, in the form of a `adr_name_t` reference. Once you retrieve a *name*, you can obtain a remote object reference as shown in the following example.

Example 2-5 C Language – Obtaining a Remote Object Reference From a Name

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include<rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
rc_instance_t *zone_inst;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, NULL);
if (conn != NULL) {
        status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_GLOB, &name_list,
          &name_count, 0);
      if (status == RCE_OK) {
               status = rc_lookup(conn, name_list[0],
                  NULL, B_TRUE, &zone_inst);
              if (status == RCE_OK) {
                      char *name;
```

```
                        status = zonemgr_Zone_get_name(zone_inst, &name);
                       if (status == RCE_OK)
                               printf("Zone name: %s\n",
                                   name);
                       free(name);
                  }
                  name_array_free(name_list, name_count);
          }
}
```

Sophisticated RAD Searches in C

You can search for a zone by its *name* or *ID*, or search for a set of zones by pattern matching. Use the list function to restrict the results. For example, if zones are identified by *name*, you can search for a zone named `test-0` by using glob patterns as follows.

Example 2-6 C Language – Using Glob Patterns

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
   status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_GLOB, B_TRUE, &name_list,
    &name_count, 1, "name", "test-0");
      if (status == RCE_OK) {
           for (int i = 0; i < name_count; i++) {
                const char *name = adr_name_tostr(name_list[i]);
                printf("%s\n", name);
             }
           name_array_free(name_list, name_count);
        }
}
```

Glob Pattern Searching in RAD in C

You can use a glob pattern to find zones with wildcard pattern matching. Keys or values in the pattern may contain an asterisk, `*`, for wildcard pattern matching. For example, you can search all the zones with a *name* that begins with *test* as follows.

Example 2-7 C Language – Using Glob Patterns With Wildcards

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
    status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_GLOB, &name_list,
      &name_count, 1, "name", "test*");
```

```
    if (status == RCE_OK) {
        for (int i = 0; i < name_count; i++) {
            const char *name = adr_name_tostr(name_list[i]);
            printf("%s\n", name);
        }
        name_array_free(name_list, name_count);
    }
}
```

Regex Pattern Searching in RAD in C

You can also use the extended regular expression (ERE) search capabilities of RAD to search for a zone. For example, you can find only zones with the *name test-0* or *test-1* as follows.

Example 2-8 C Language – Using Regex Patterns

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
    status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_REGEX,
    &name_list, &name_count, 1, "name", "test-0|test-1");
    if (status == RCE_OK) {
        for (int i = 0; i < name_count; i++) {
            const char *name = adr_name_tostr(name_list[i]);
            printf("%s\n", name);
        }
        name_array_free(name_list, name_count);
    }
}
```

The key and the value must be valid EREs as determined by the connected RAD instance. The expression is compiled and executed on the server.

RAD Interface Components in C

The module developer defines an API in an ADR IDL document. It contains one or more of the following components, each of which performs a task:

- Enumerations
  - Values
- Structures
  - Fields
- Dictionary
- Interfaces
  - Properties
  - Methods

- Events

The `radadrgen` utility is used to process the document to generate language-specific components, which facilitates client-server interaction within RAD. For more information about the role of ADR and RAD, see Chapter 2, Abstract Data Representation for RAD in *Remote Administration Daemon Module Developer's Guide*. The following sections describe each component.

RAD Enumerations in C

Enumerations provide a restricted range of choices for a property, an interface method parameter, a result, or an error.

Using RAD Enumeration Types in C

Enumerated types are defined in the binding header with the type prepended with the module name. The values of the enumerated types are prepended to follow the C coding standard naming conventions.

Example 2-9 C Language – `zonemgr` ErrorCode Enumeration for RAD in C

```
typedef enum zonemgr_ErrorCode {
     ZEC_NONE =0,
     ZEC_FRAMEWORK_ERROR = 1,
     ZEC_SNAPSHOT_ERROR = 2,
     ZEC_COMMAND_ERROR = 3,
     ZEC_RESOURCE_ALREADY_EXISTS = 4,
     ZEC_RESOURCE_NOT_FOUND = 5,
     ZEC_RESOURCE_TOO_MANY = 6,
     ZEC_RESOURCE_UNKNOWN = 7,
     ZEC_ALREADY_EDITING = 8,
     ZEC_PROPERTY_UNKNOWN = 9,
     ZEC_NOT_EDITING = 10,
     ZEC_SYSTEM_ERROR = 11,
     ZEC_INVALID_ARGUMENT = 12,
     ZEC_INVALID_ZONE_STATE = 13,
}zonemgr_ErrorCode_t;
```

RAD Structures in C

Structures (Structs) are used to define new types and are composed from existing built-in types and other user defined types. Structs are simple forms of interfaces with no methods or events. They are not included in the RAD namespace.

Using RAD Struct Types in C

The `zonemgr` module defines a property struct, which represents an individual zone configuration property. The structure has the following members, `name`, `type`, `value`, `listValue`, and `complexValue`. Like enumerations, structures are defined in the binding header and follow similar naming conventions.

To free a structure, free functions *module_structure*`_free`() are provided by the binding to ensure proper cleanup of any memory held in the nested data.

Example 2-10 C Language – `zonemgr` Property Struct Definition and Its Free Function

```
typedef enum zonemgr_PropertyValueType {
     ZPVT_PROP_SIMPLE = 0,
     ZPVT_PROP_LIST = 1,
     ZPVT_PROP_COMPLEX = 2,
} zonemgr_PropertyValueType_t;

typedef struct zonemgr_Property {
 char * zp_name;
 char * zp_value;
 zonemgr_PropertyValueType_t zp_type;
 char * * zp_listvalue;
 int zp_listvalue_count;
 char * * zp_complexvalue;
 int zp_complexvalue_count;
} zonemgr_Property_t;

void zonemgr_Property_free(zonemgr_Property_t *);
```

Dictionary Support in C for RAD

C does not support dictionary data types natively. To support dictionary in types and functions, you must enable the dictionary functionality for each dictionary type as part of a module's C binding. You can create, free, and query a dictionary for its size. The supported operations on a dictionary include getting, putting, and removing an element. The functions `_keys`() and `_values`() return an array of all keys and values, respectively. The `_map`() function is called with a pointer to a function that is invoked with each key-value pair. For more information about dictionary, see Dictionary Definitions in RAD Modules in *Remote Administration Daemon Module Developer's Guide*.

The C binding dictionary is a wrapper around the `libadr` library. The `libadr` library functions that are supported for dictionary are similar to the functions supported by C. The functions are in the native C type instead of the `libadr` (`adr_data_t`()) type. For more information, see Dictionary Support in libadr in *Remote Administration Daemon Module Developer's Guide*.

The following is an example of a generated type and API of a dictionary where the key type is `integer` and the value type is `string`. In this example, `<module>` is the name of the module.

```
typedef struct <module>__rad_dict_integer_string
   <module>__rad_dict_integer_string_t;

<module>__rad_dict_integer_string_t *
   <module>__rad_dict_integer_string_create(
   const rc_instance_t *inst);

void <module>__rad_dict_integer_string_free(
   <module>__rad_dict_integer_string_t *dict);
rc_err_t <module>__rad_dict_integer_string_contains(
   <module>__rad_dict_integer_string_t *dict, int key);
unsigned int <module>__rad_dict_integer_string_size(
   <module>__rad_dict_integer_string_t *dict);
rc_err_t <module>__rad_dict_integer_string_remove(
   <module>__rad_dict_integer_string_t *dict, int key,
   char **value);
rc_err_t <module>__rad_dict_integer_string_get(
   <module>__rad_dict_integer_string_t *dict, int key,
    char **value);
rc_err_t <module>__rad_dict_integer_string_put(
   <module>__rad_dict_integer_string_t *dict, int key,
```

```
   const char *value, char **old_value);
int *<module>__rad_dict_integer_string_keys(
   <module>__rad_dict_integer_string_t *dict);
char **<module>__rad_dict_integer_string_values(
   <module>__rad_dict_integer_string_t *dict);
int <module>__rad_dict_integer_string_map(
   <module>__rad_dict_integer_string_t *dict,
int (*func)(int, const char *, void *), void *arg);
```

The generated type can be used like any other type in RAD. A sample C client binding definition is as follows:

```
rc_err_t <module>_<interface>_set_DictProp(rc_instance_t *,
  <module>__rad_dict_integer_string_t *);
```

> **Note:**
>
> The dictionary type and associated functions are thread-safe.

RAD Interfaces in C

Interfaces, also known as objects, are the entities which populate the RAD namespace. They must have a *name*. An interface is composed of events, properties, and methods.

Obtaining a RAD Object Reference in C

See the RAD Namespace in C section.

Working With RAD Object References in C

Once you have an object reference, you can use this object reference to interact with RAD directly. All attributes and methods defined in IDL are accessible by invoking calling functions in the generated client binding.

The following example shows how to work with the object references. In this example, you get a reference to a zone and then boot the zone.

Example 2-11 C Language – Working With RAD Object References

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
rc_instance_t *zone_inst;
zonemgr_Result_t *result;
zonemgr_Result_t *error;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
     status = zonemgr_Zone__rad_lookup(conn, B_TRUE, &zone_inst, 1, "name",
"test-0");
     if (status == RCE_OK) {
          status = zonemgr_Zone_boot(zone_inst, NULL, 0, &result, &error);
          rc_instance_rele(zone_inst);
```

```
        }
}
```

Accessing a Remote Property in RAD in C

This example shows how to access a remote property.

Example 2-12 C Language – Accessing a RAD Remote Property

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

int
main(int argc, char **argv)
{
        rc_err_t status;
        rc_instance_t *zone_inst;
        char *name;
        zonemgr_Property_t **result;
        zonemgr_Result_t *error;
        int result_count;

        rc_conn_t *conn = rc_connect_unix(NULL, NULL);

        if (conn != NULL) {
                status = zonemgr_Zone__rad_lookup(conn, B_TRUE, &zone_inst,
1,
                    "name", "test-0");
                if (status == RCE_OK) {
                        zonemgr_Resource_t global = { .zr_type = "global"};
                        status =
zonemgr_Zone_getResourceProperties(zone_inst,
                            &global, NULL, 0, &result, &result_count, &error);
                        if (status == RCE_OK) {
                                for (int i = 0; i < result_count; i++){
                                        if (result[i]->zp_value != NULL &&
                                            result[i]->zp_value[0] != '\0') {
                                                printf("%s=%s\n",
                                                    result[i]->zp_name,
                                                    result[i]->zp_value);
                                        }
                                }
                                zonemgr_Property_array_free(result,
result_count);
                        }
                        rc_instance_rele(zone_inst);
                }
        }
}
```

In this example, you have accessed the list of global resource properties of the `Zone` and printed the name and value of every property that has a value.

RAD Event Handling in C

An event is an asynchronous notification generated by RAD and consumed by clients. For more information, see RAD Events in *Remote Administration Daemon Module Developer's Guide*.

The following example shows how to subscribe to and handle events. The `ZoneManager` instance defines a `StateChange` event that clients can subscribe to information about the changes in the runtime state of a zone.

Example 2-13 C Language – Subscribing to and Handling RAD Events

```
#include <unistd.h>
#include <time.h>
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

void stateChange_handler(rc_instance_t *inst, zonemgr_StateChange_t *payload,
struct timespec timestamp, void *arg)
{
    printf("event: zone state change\n");
    printf("payload:\n zone: %s\n old state: %s\n new state: %s\n",
    payload->zsc_zone, payload->zsc_oldstate, payload->zsc_newstate);

    zonemgr_StateChange_free(payload);
}

rc_err_t status;
rc_instance_t *zm_inst;
int result_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
        status = zonemgr_ZoneManager__rad_lookup(conn, B_TRUE, &zm_inst, 0);
        if (status == RCE_OK) {
            status = zonemgr_ZoneManager_subscribe_stateChange(zm_inst,
stateChange_handler, NULL);
            if (status == RCE_OK)
                printf("Successfully subscribed to statechange event!\n");
            rc_instance_rele(zm_inst);
        }
 }
 for (;;)
 sleep(1);
```

In this example, you have subscribed to a single event by passing a handler and a handle for the `ZoneManager` object. The handler is invoked asynchronously by the framework with various event details and user data. In this example, the user data is `NULL`.

RAD Error Handling in C

The list of possible errors are defined by the `rc_err_t` enumeration. The following example shows how it can be used.

Example 2-14 C Language – Handling RAD Errors

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
rc_instance_t *zone_inst;
zonemgr_Result_t *result;
zonemgr_Result_t *error;
```

```
rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
      status = zonemgr_Zone__rad_lookup(conn, B_TRUE, &zone_inst, 1, "name", "test-0");
     if (status == RCE_OK) {
             status = zonemgr_Zone_boot(zone_inst, NULL, 0, &result, &error);
            if (status == RCE_SERVER_OBJECT) {
                 printf("Error Code %d\n", error->zr_code);
                    if (error->zr_stdout != NULL)
                         printf("stdout: %s\n", error->zr_stdout);
                     if (error->zr_stderr != NULL)
                         printf("stderr: %s\n", error->zr_stderr);
                      zonemgr_Result_free(error);
                }
                rc_instance_rele(zone_inst);
        }
}
```

> **Note:**
>
> You might get a payload with `rc_err_t`, value `RCE_SERVER_OBJECT`. This means that the server is sending additional information about the error. This payload is only present if your interface method or property has defined an `error` element, where the payload is the content of that error. If the interface method or property defines no `error` element for the interface method or property, no payload exists and no error reference argument exists for the `get` or `set` functions.

RAD TLS Client in C

The `libradclient` C implementation includes the `rc_connect_tls_ex`() function that permits the client to connect to the RAD service over TLS. The connection requires that you specify the X.509 client certificate and its key file. For information about the server setup, see Configuring RAD Transports.

The `rc_connect_tls`() and `rc_connect_tls_ex`() functions take the following arguments:

```
rc_conn_t *
rc_connect_tls(const char *host, int port, const char *cert_files,
    char *locale);

rc_conn_t *
rc_connect_tls_ex(const char *host, int port, const char *cert_files,
    char *locale, const char *cert, const char *key);
```

host
Name of the host.

port
Port number.

cert_files
A colon-separated list of locations for server certificate validation. This value can be null.

locale
Name of the locale. If `locale` is NULL, the value is set to the locale that the client system uses.

cert
Location of the X.509 client certificate. This value can be null.

key
Location of the key file associated with the client certificate. This value can be null.

RAD Java Client

The public Java interfaces are exported in the following packages:

- `com.oracle.solaris.rad.client` – The client implementation of the RAD protocol and associated functionality
- `com.oracle.solaris.rad.connect` – The classes for connecting to a RAD instance

> **Note:**
>
> Most of the examples are based on the **zonemgr** interface. To better understand these examples, see Appendix A, zonemgr ADR Interface Description Language Example in *Remote Administration Daemon Module Developer's Guide*.

Connecting to RAD in Java

RAD instances can communicate through the `Connection` class. Various factory interfaces are available to get different types of connections to a RAD instance. Each mechanism returns a connection instance that provides a standard interface to interact with RAD. The connection can be closed with the `close`() method.

Connecting to a RAD Local Instance in Java

You can connect to a local instance by using the `Connection.connectUnix()` class. An implicit authentication is performed against your user ID and most RAD tasks you request with this connection are performed with the privileges available to your user account.

Example 2-15 Java Language – Creating a Local RAD Connection

```
import com.oracle.solaris.rad.connect.Connection;

Connection con = Connection.connectUnix();
```

Connecting to a Remote RAD Instance and Authenticating in Java

When connecting to a remote instance, no implicit authentication is performed. The connection is not established until you authenticate. The `com.oracle.solaris.rad.client` package provides a utility class (`RadAuthHandler`)

which can be used to perform a PAM login. If you provide a locale, username and password, authentication is non-interactive. If `locale` is null, then `C` is used.

The following example shows how to connect to a TCP instance on port 7777.

Example 2-16 Java Language – Creating Remote RAD Connection Over TCP IPv4 on Port 7777

```
import com.oracle.solaris.rad.client.RadAuthHandler;
import com.oracle.solaris.rad.connect.Connection;

Connection con = Connection.connectTCP("host1", 7777);
System.out.println("Connected: " + con.toString());
RadAuthHandler hdl = new RadAuthHandler(con);
hdl.login("C", "user", "password"); // First argument is locale
con.close();
```

RAD is deployed as two cooperating processes. A `proxy` process is responsible for authentication and establishing communications. A `slave` process is created by the proxy and handles module processing. A slave is created for each client connection.

Connecting to a RAD Instance by Using a URI in Java

You can use a URI to connect to a local or remote RAD instance. You can use the class `URIconnection` in Java for connecting using a URI. For more information, see Connecting in Python to a RAD Instance by Using a URI.

The following constructors are supported.

```
public URIConnection(String src) throws IOException {
        this(src, DEFAULT_SCHEMES);
}

public URIConnection(String src, Set<String> schemes)
        throws IOException {
}

public URIConnection(String src, Set<String> schemes,
        Set<String> certfiles) throws IOException {
}
```

Use the different constructors depending on how much control you need over the connection.

For methods, the following functions are supported for adding or removing certificates for TLS connections, and connecting and processing PAM information.

```
public void addCertFile(String certfile) {
}

public void rmCertFile(String certfile) {
}

public Connection connect(Credentials cred) throws IOException {
}

public void processPAMAuth(PAMCredentials cred, Connection con) throws IOException {
}
```

The following utility functions are supported for providing information about a RAD instance:

- `public String getAuth()`
- `public String getCredClass()`
- `public void setCredClass(String klass) throws IOException`
- `public String getHost()`
- `public String getPath()`
- `public int getPort()`
- `public String getSrc()`
- `public String getScheme()`
- `public Set<String> getSchemes()`
- `public String getUser()`

You can use the class `PAMCredentials` to create a set of PAM credentials for authentication. The supported constructor is `public PAMCredentials(String pass)`.

RAD Namespace in Java

Most RAD objects that are represented in the ADR document as *<interfaces>*. You can search RAD objects by searching the RAD namespace. To access a RAD object, you need a proxy, which is used to search the RAD namespace. An interface proxy class enables you to use a proxy to search the RAD namespace. The interface proxy is defined in the binding module of each interface.

The proxy provides the base name and the version details for the interface instances and is structured as follows:

```
<domain name>:type=<interface name>[,optional additional key value pairs]
```

The `<domain name>` and the `<interface name>` are automatically derived from the ADR IDL definition and are stored in the module binding.

Certain interfaces return or accept object references directly to or from clients. These objects might not be named. Objects that are not named are **anonymous**. Anonymous objects cannot be looked up in the RAD namespace, but the interface provides access methods that make it simple to interact with them.

Creating a Name for a RAD Object in Java

The names are changed to be represented by a domain string and a Map <String, String> for the key or value pairs. The `ADRName` constructors are expanded to include:

```
ADRName(String domain, Map<String, String> kvpairs)
ADRName(String domain, Map<String, String> kvpairs,
ProxyInterface proxy, Version version)
```

Searching for RAD Objects in Java

Using the `Connection` class, you can list the objects by name and obtain a remote object reference.

RAD Singletons in Java

A module developer creates a singleton to represent an interface. This interface can be accessed easily. For example, the `zonemgr` module defines a singleton interface, **ZoneInfo**. It contains information about the zone that contains the RAD instance with which you are communicating.

In Java, you need to compile the code with the language binding in the `CLASSPATH`. RAD Java Language bindings are in the `system/management/rad/client/rad-java` package.

The JAR files for the various bindings are installed in `/usr/lib/rad/java`. Each major interface version is accessible in a JAR file which is named after the source ADR document and it's major version number. For example, to access major version 1 of the `zonemgr` API, use `/usr/lib/rad/java/zonemgr_1.jar`. Symbolic links are provided as an indication of the default version a client should use.

Example 2-17 Java Language – Obtaining a Reference to a RAD Singleton

```
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.ZoneInfo;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());
ZoneInfo zi = con.getObject(new ZoneInfo());
System.out.println("ZoneInfo: " + zi.getname());
```

In this example, you have performed the following:

- Imported **ZoneInfo** and `Connection` from the `zonemgr` binding and the `rad.connect` package
- Connected to the local RAD instance
- Obtained a remote object reference directly by using a proxy instance

After you have the remote reference, you can access the properties and the methods directly. In the RAD Java implementation, all properties are accessed using the `getter` or `setter` syntax. Thus, you invoke `getname`() to access the `name` property.

Listing RAD Interface Instances in Java

An interface can contain multiple RAD instances. For example, the `zonemgr` module defines a **Zone** interface and there is an instance for each zone on the system. The `Connection` class provides the `list_objects`() method to list the interface instances as shown in the following example.

Example 2-18 Java Language – Listing RAD Interface Instances

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

for (ADRName name: con.listObjects(new Zone())) {
    System.out.println("ADR Name: " + name.toString());
}
```

Remote Object References and RAD Names in Java

A list of names (`ADRName` is the class name) are returned by the `list_objects`() method from the `Connection` class. After you have a *name*, you can obtain a remote object reference easily as shown in the following example.

Example 2-19 Java Language – Obtaining a Remote Object Reference From a RAD Name

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

for (ADRName name: con.listObjects(new Zone())) {
    Zone zone =  con.getObject(name);
    System.out.println("Name: " + zone.getname());
}
```

Sophisticated RAD Searches in Java

You can search for a zone by its *name* or *ID* or a set of zones by pattern matching. You can extend the definition of a name provided by a proxy. For example, if zones are uniquely identified by a key *name*, then you can find a zone with name *test-0* as shown in the following example. This example uses glob patterns to find a zone.

Example 2-20 Java Language – Using Glob Patterns

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test-0" };
ADRGlobPattern pat = new ADRGlobPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    System.out.println("ADR Name: " + name.toString());
}
```

In this example, the `ADRGlobPattern` class (imported from the `com.oracle.solaris.rad.client` package) is used to refine the search. The `list_objects`() method from the `Connection` class is used, but the search is refined by extending the name definition. The `ADRGlobPattern` class takes an array of keys and an array of values and extends the name used in the search.

Glob Pattern Searching in RAD in Java

You can use a glob pattern to find zones with wildcard pattern matching. Keys or Values in the pattern may contain `*`, which is interpreted as wildcard pattern matching. For example, you can find all zones with a *name* which begins with *test* as follows.

Example 2-21 Java Language – Using Glob Patterns With Wildcards

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test*" };
ADRGlobPattern pat = new ADRGlobPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    System.out.println("ADR Name: " + name.toString());
}
```

Using Maps When Pattern Searching in RAD in Java

It can be simpler to use `Map` rather than arrays of keys and values. This example uses a map of keys and values rather than arrays of keys and values.

Example 2-22 Java Language – Using Maps With Patterns

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

Map<String, String> kvpairs = new HashMap<String, String>();
kvpairs.put("name", "test*");
ADRGlobPattern pat = new ADRGlobPattern(kvpairs);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    System.out.println("ADR Name: " + name.toString());
}
```

Regex Pattern Searching in RAD in Java

You can also use RAD's ERE search capabilities to search a zone. For example, you can find only zones with the name `test-0` or `test-1` as shown in the following example.

Example 2-23 Java Language – Using Regex Patterns

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRRegexPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test-0|test-1" };
ADRRegexPattern pat = new ADRRegexPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    System.out.println("ADR Name: " + name.toString());
}
```

The key and the value must be valid ERE as determined by the instance of RAD to that you are connected. The expression is compiled and executed on the server.

Interface Components for RAD in Java

An API is defined by a module developer. It contains one or more of the following components, each of which performs a task:

- Enumerations
  - Values
- Structures
  - Fields
- Dictionary
- Interfaces
  - Properties
  - Methods
  - Events

These components are defined in an ADR IDL document. The `radadrgen` utility is used to process the document to generate language specific components which facilitates client-server interactions within RAD. For more information about the role of ADR and RAD, see Chapter 2, Abstract Data Representation for RAD in *Remote Administration Daemon Module Developer's Guide*. The following sections describe each component.

RAD Property Enumerations in Java

Enumerations provide a restricted range of choices for a property, an interface method parameter, result, or error.

Using RAD Enumeration Types in Java

To access an enumerated type, import the generated class and interact with the enumeration.

Example 2-24 Java Language – Using RAD Enumerations

```
import com.oracle.solaris.rad.zonemgr.ErrorCode;

System.out.println(ErrorCode.NONE);
System.out.println(ErrorCode.COMMAND_ERROR);
```

RAD Structs in Java

Structs are used to define new types and are composed from existing built-in types and other user defined types. Structs are simple forms of interfaces with no methods or events. They are not included in the RAD namespace.

Using RAD Struct Types in Java

The `zonemgr` module defines a `Property` struct, which represents an individual zone configuration property. The structure has the following members `name`, `type`, `value`,

`listValue`, and `complexValue`. Like enumerations, structs can be interacted directly once the binding is imported.

Example 2-25 Java Language – Using RAD Structs

```
import com.oracle.solaris.rad.zonemgr.Property;

Property prop = new Property();
prop.setName("my name");
prop.setValue("a value");
System.out.println(prop.getName());
System.out.println(prop.getValue());
```

Dictionary Support for RAD in Java

To support the dictionary type, Java client uses the `java.util.Map<K,V>` interface. For more information about dictionary, see Dictionary Definitions in RAD Modules in *Remote Administration Daemon Module Developer's Guide*.

The following example shows how to read and write a property defined in a dictionary. For more information, see Defining a Dictionary for RAD in *Remote Administration Daemon Module Developer's Guide*.

```
//reading a property value
Map<Integer, String> property = o.getDictProp();

//writing a property value
Map<Integer, String> property = new HashMap<Integer, String>();
....
o.setDictProp(property);
```

RAD Interfaces in Java

Interfaces, also known as objects, are the entities, which populate the RAD namespace. They must have a *name*. An interface is composed of events, properties, and methods.

Obtaining a RAD Object Reference in Java

For more information, see RAD Namespace in Java.

Working With RAD Object References in Java

Once you have an object reference, you can use this object reference to interact with RAD directly. All attributes and methods defined in the IDL are accessible directly as attributes and methods of the Java objects that are returned by the `getObject`() function. The attributes are accessed using the automatically generated `getter` or `setter`. For example, if the property is `name`, you would use `getname` or `setname(<value>)`. In this example, you get a reference to a zone and then boot the zone.

Example 2-26 Java Language – Invoking a RAD Remote Method

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());
```

```
String keys[] = { "name" };
String values[] = { "test-0" };
ADRGlobPattern pat = new ADRGlobPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    Zone z = (Zone) con.getObject(name);
    z.boot(null);
}
```

In this example, you have connected to the RAD instance, created a search for a specific object, retrieved a reference to the object, and invoked a remote method on the object.

RAD Remote Property Example in Java

Accessing a remote property is similar to using a remote method.

Example 2-27 Java Language – Accessing a RAD Remote Property

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.*;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test-0" };
ADRGlobPattern pat = new ADRGlobPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    Zone z = (Zone) con.getObject(name);
    Resource filter = new Resource("global", null, null);
    List<Property> props = z.getResourceProperties(filter, null);
    System.out.println("Properties:");
    for (Property prop: props) {
        System.out.printf("\t%s = %s\n",prop.getName(), prop.getValue());
    }
}
```

In this example, you have accessed the list of global resource properties of the `Zone` and printed the name and value of every Property.

RAD Event Handling in Java

This example shows how to subscribe and handle events. The `ZoneManager` instance defines a `stateChange` event, which clients can subscribe for information about changes in the runtime state of a zone.

Example 2-28 Java Language – Subscribing to and Handling RAD Events

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.RadEvent;
import com.oracle.solaris.rad.client.RadEventHandler;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.*;

ZoneManager zmgr = con.getObject(new ZoneManager());
    con.subscribe(zmgr, "statechange", new StateChangeHandler());
    Thread.currentThread().sleep(100000000);
```

```
class StateChangeHandler extends RadEventHandler {

    public void handleEvent(RadEvent event, Object payload) {
        StateChange obj = (StateChange) payload;
        System.out.printf("Event: %s", event.toString());
        System.out.printf("\tcode: %s\n", obj.getZone());
        System.out.printf("\told: %s\n", obj.getOldstate());
        System.out.printf("\tnew: %s\n", obj.getNewstate());
       }
}
```

To handle an event, implement the `RadEventInterface` class. The `com.oracle.solaris.rad.client` package provides a default implementation (`RadEventHandler`) with limited functions. This class can be extended to provide additional event handling logic as in the example above.

In this example, you have subscribed to a single event by passing a handler and a reference to the `ZoneManager` object. The handler is invoked asynchronously by the framework with various event details and provided the user data.

Java Error Handling in RAD

Java provides a exception handling mechanism and RAD errors are propagated using this method. RAD delivers a variety of errors, but the error that requires handling is `RadObjectException`. The following example shows how to handle RAD errors.

Example 2-29 Java Language – Handling RAD Errors

```
<imports..>

Connection con = Connection.connectUnix();
    for (ADRName name: con.listObjects(new Zone())) {
        Zone zone =  con.getObject(name);
        try {
            zone.boot(null);
        }catch (RadObjectException oe) {
            Result res = (Result) oe.getPayload();
            System.out.println(res.getCode());
            if (res.getStdout() != null)
                System.out.println(res.getStdout());
            if (res.getStderr() != null)
                System.out.println(res.getStderr());
       }
}
```

> **Note:**
>
> With `RadException` exceptions, you might get a payload. This payload is only present if your interface method or property has defined an `error` element, where the payload is the content of that error. If the interface method or property defines no `error` element for the interface method or property, then no payload exists and `error` has a value of null.

RAD TLS Client in Java

The Java implementation has the `Connection.connectTLS(hostname, port, certfiles, locale, keystorefname, keystorepassfname)` method. This method enables you to specify a PKCS #12 archive (*keystorefname*) and the file that holds the password (*keystorepassfname*) used to unlock the PKCS #12 archive.

The following example test code verifies the functionality of the RAD TLS client X.509 authentication implementation:

Because Java cannot use the default X.509 certificates and their corresponding key file like C and Python can, a Java user must first create a PKCS #12 archive from the certificate and key files. The following example test code fragment shows how to create the archive:

```
# Create a PKCS#12 keystore that Java can use

pkcs12_password=$(od -An -N6 -x /dev/urandom | nawk '{print $1$2$3;}')
echo "$pkcs12_password" > ${CERT_HOST_PKCS12_PASS}
openssl pkcs12 -export \
        -password file:${CERT_HOST_PKCS12_PASS} \
        -in ${CERT_HOST_CRT} \
        -inkey ${CERT_HOST_KEY} \
-out ${CERT_HOST_PKCS12}
```

RAD Python Client

The public interfaces are exported in the following three modules:

- `rad.auth` – Useful functions or classes for performing authentication
- `rad.client` – The client implementation of the RAD protocol and associated useful functionality
- `rad.connect` – Useful functions or classes for connecting to a RAD instance

> **Note:**
>
> Most of the examples are based on the **zonemgr** interface. To understand the examples for this module better, see Appendix A, zonemgr ADR Interface Description Language Example in *Remote Administration Daemon Module Developer's Guide*.

Alternatively, you can import the module and examine the module help.

Example 2-30 Accessing Help for a Binding Module

```
user@host1:/var/tmp# python3.7
Python 3.7.5 (default, Aug 20 2020, 02:25:50)
[GCC 9.3.0] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr_1 as zonemgr
>>> help(zonemgr)
```

Connecting to RAD in Python

The RAD instances can communicate through the `RADConnection` class. There are various mechanism to get different types of connections to RAD. Each mechanism returns a `RADConnection` instance, which provides a standard interface to interact with RAD.

The preferred method for managing a connection is to use the `with` keyword. The connection uses the system resources and this ensures that the resource is closed correctly when the object goes out of scope. If the system resources are not used, the system resources can be reclaimed explicitly with the `close`() method.

> **Note:**
>
> If you print the `RADConnection` object, it displays the state of the connection and lets you know if the connection is closed.

Connecting to a Local RAD Instance in Python

You can connect to a local instance using the `radcon.connect_unix`() function. An implicit authentication is performed against your user ID and most RAD tasks you request with this connection are performed with the privileges available to your user account.

Example 2-31 Python Language – Creating a RAD Local Connection

```
>>> import rad.connect as radcon

>>> with radcon.connect_unix() as rc:
```

Connecting to a RAD Remote Instance and Authenticating in Python

When connecting to a remote instance, no implicit authentication is performed. The connection is not established until you authenticate. The `rad.auth` module provides a utility class (`RadAuth`), which may be used to perform a PAM login. If you provide a username and password, authentication is non-interactive. If you do not provide username and password, you will receive a console prompt for the missing information.

Example 2-32 Python Language – Creating a RAD Remote Connection Over TLS

```
>>> import rad.connect as radcon
>>> import rad.auth as rada

>>> rc=radcon.connect_tls("host1")
>>> # Illustrate examining RadConnection state.
>>> print rc
<open RadConnection >
>>> auth = rada.RadAuth(rc)
>>> auth.authenticate("jdoe", "xxxpasswordxxx")
>>> <now authenticated and can use this connection>
>>> rc.close()
>>> print rc
<closed RadConnection >
>>>
```

RAD is deployed as two cooperating processes. A `proxy` process is responsible for authentication and establishing communications. A `slave` process is created by the proxy and handles module processing. A slave is created for each client connection.

Connecting to a RAD Instance by Using a URI in Python

You can use a URI to connect to a local or remote RAD instance. You can use the class `RadURI()` to connect to a RAD instance. The methods or functions are not required in Python because you can read the attributes of the RAD instances that you create instead of using defined methods. For more information, see Connecting in Python to a RAD Instance by Using a URI.

The following constructor is supported.

```
def __init__(self, src, schemes = RAD_SCHEMES):
```

src
String, which is the URI of a RAD instance

schemes
List of strings that specify the schemes to be recognized

The following method is supported:

```
def connect(self, cred = None):
```

cred
Credentials that are required for authentication

You can use `PAMCredentials` class to create PAM credentials for PAM authentication or you can use `def get_pam_cred(passw)` function, which returns a `PAMCredentials` object for use in the `RadURI.connect`() method.

RAD Namespace in Python

Most RAD objects that are represented in the ADR document as *<interfaces>*. You can find RAD objects by searching the RAD namespace. To access a RAD object, you need a proxy, which is used to search the RAD namespace. An interface proxy class enables you to use a proxy to search the RAD namespace. The interface proxy is defined in the binding module of each interface.

The proxy provides the base name and version details for interface instances and is structured as follows:

```
<domain name>:type=<interface name>[,optional additional key value pairs]
```

The `<domain name>` and the `<interface name>` are automatically derived from the ADR IDL definition and are stored in the module binding.

Certain interfaces return or accept object references directly to or from clients. These objects might not be named. Objects that are not named are **anonymous**. Anonymous objects cannot be looked up in the RAD namespace, but the interface provides access methods that make it simple to interact with them.

Creating a Name for a RAD Object in Python

The RAD object names are structured, consisting of a domain and one or more key-value pairs.

For example, you can create a name for a `zonemgr` zone instance as follows:

```
>>> ADRName("com.oracle.solaris.rad.zonemgr", { "type": "Zone",
               "name" : "radtest-zone", "id" : "1" })
```

When you create a name, you can handle key-value pairs. This removes any issues in processing names where values contain special characters (for example, commas (`,`), and equal signs (`=`).

The `RADConnection` class provides methods for listing objects by name and for obtaining a remote object reference.

RAD Singletons in Python

A module developer creates a singleton to represent an interface. This interface can be accessed easily. For example, the `zonemgr` module defines a singleton interface, `ZoneInfo`. It contains information about the zone that contains the RAD instance with which you are communicating.

Example 2-33 Python Language – Obtaining a Reference to a RAD Singleton

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
...     zi = rc.get_object(zonemgr.ZoneInfo())
...     print zi.name
...
global
>>>
```

In this example, you have imported the RAD bindings and the `rad.connect` module, and connected to the local RAD instance. After connecting to the local RAD instance, obtain a remote object reference directly by using a proxy instance. After you have the remote reference, you can access properties and methods directly as you would with any Python object.

Listing RAD Instances of an Interface in Python

An interface can contain multiple RAD instances. For example, the `zonemgr` module defines a Zone interface and there is an instance for each zone on the system. The `RADConnection` class provides the `list_objects`() method to list the interface instances as shown in the following example.

Example 2-34 Python Language – Listing RAD Interface Instances

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone())
... print zonelist
...
[Name: com.oracle.solaris.rad.zonemgr:type=Zone,name=test-0,id=-1 Version: (1.0),
```

```
Name:
com.oracle.solaris.rad.zonemgr:type=Zone,name=test-1,id=-1 Version: (1.0), Name:
com.oracle.solaris.rad.zonemgr:type=Zone,name=NOT-TEST,id=-1 Version: (1.0)]
>>>
```

Obtaining a RAD Remote Object Reference From a Name in Python

Names (`ADRName` is the class name) are returned by the `RADConnection list_objects` method. Once you have a *name*, you can obtain a remote object reference easily.

Example 2-35 Python Language – Obtaining a RAD Remote Object Reference

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone())
... zone = rc.get_object(zonelist[0])
... print zone.name
...
test-0
>>>
```

You can get values of individual components of an ADRName object by using the dot notation `nameObj.key` or by using the `getattr(nameObj, key)` function call.

```
zonelist[0].name
```

Sophisticated RAD Searches in Python

You can search for a zone by its *name* or *ID* or a set of zones by pattern matching. You can extend the basic definition of a name provided by a proxy. For example, if zones are uniquely identified by the key *name*, then you can find a zone with the name `test-0` as shown in the following example. The example uses glob patterns to find a zone.

Example 2-36 Python Language – Using Glob Patterns

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone(), radc.ADRGlobPattern({"name" :
"test-0"}))
... print zonelist
...
[Name: com.oracle.solaris.rad.zonemgr:type=Zone,name=test-0,id=-1 Version: (1.0)]
>>>
```

In this example, the `ADRGlobPattern` class (imported from the `rad.client` module) is used to refine the search. The `list_objects`() method from the `RADConnection` class is used, but the search is refined by extending the name definition. The `ADRGlobPattern` class takes a *key:value* dictionary and extends the name used for the search.

Glob Pattern Searching in RAD in Python

You can use a glob pattern to find zones with wildcard pattern matching. Keys and values in the pattern may contain an asterisk (`*`), which is interpreted as wildcard

pattern matching. The following example shows how to find all zones with a *name* which begins with *test*.

Example 2-37 Python Language – Using Glob Patterns With Wildcards in RAD

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone(), radc.ADRGlobPattern({"name" :
"test*"}))
...  print zonelist
...
[Name: com.oracle.solaris.rad.zonemgr:type=Zone,name=test-0,id=-1 Version: (1.0), Name:
com.oracle.solaris.rad.zonemgr:type=Zone,name=test-1,id=-1 Version: (1.0)]
>>>
```

Regex Pattern Searching in RAD in Python

You can also use ERE search capabilities of RAD. The following example shows how to find only zones with name *test-0* or *test-1*.

Example 2-38 Python Language – Using Regex Patterns in RAD

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone(), radc.ADRRegexPattern({"name" : "test-0|
test-1"}))
...  print zonelist...
[Name: com.oracle.solaris.rad.zonemgr:type=Zone,name=test-0,id=-1 Version: (1.0), Name:
com.oracle.solaris.rad.zonemgr:type=Zone,name=test-1,id=-1 Version: (1.0)]
>>>
```

The key and the value must be valid EREs as determined by the instance of RAD to which you are connected. The expression is compiled and executed on the server.

RAD Interface Components in Python

An API is defined by a module developer. It contains one or more of the following components, each of which performs a task:

- Enumerations
  - Values
- Structures
  - Fields
- Dictionary
- Interfaces
  - Properties
  - Methods
  - Events

These components are defined in an ADR IDL document. The `radadrgen` utility is used to process the document to generate language specific components which facilitate client-

server interactions within RAD. For more information about the role of ADR and RAD, see Chapter 2, Abstract Data Representation for RAD in *Remote Administration Daemon Module Developer's Guide*. Brief descriptions of each component follows.

RAD Enumerations in Python

Enumerations are primarily used to offer a restricted range of choices for a property, an interface method parameter, result, or error.

Using RAD Enumeration Types in Python

To access an enumerated type, import the binding and interact with the enumeration.

Example 2-39 Python Language – Using RAD Enumerations

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> print zonemgr.ErrorCode.NONE
NONE
>>> print zonemgr.ErrorCode.COMMAND_ERROR
COMMAND_ERROR
>>>
```

RAD Structure Types in Python

Structures, or "structs", are used to define new types and are composed from existing built-in types and other user defined types. Structs are simple form of interfaces with no methods or events. They are not included in the RAD namespace.

Using RAD Structs in Python

The `zonemgr` module defines a Property struct which represents an individual zone configuration property. The structure has the following members: `name`, `type`, `value`, `value`, `listValue`, and `complexValue`. Like enumerations, structures can be interacted with directly once the binding is imported.

Example 2-40 Python Language – Using RAD Structs

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> prop = zonemgr.Property("autoboot", "false")
>>> print prop
Property(name = 'autoboot', value = 'false', type = None, listvalue = None,
complexvalue = None)
>>> prop.name = "my name"
>>> prop.value = "a value"
>>> print prop.name
my name
>>> print prop.value
a value
>>>
```

Dictionary Support in Python for RAD

You can use the built-in dictionary type in Python. For example, the following Python code sets the sample dictionary property as defined in Defining a Dictionary for RAD in *Remote Administration Daemon Module Developer's Guide*:

```
object.DictProp = {1: 'value1', 2: 'value2'}
```

RAD Interfaces in Python

Interfaces, also known as objects, are the entities which populate the RAD namespace. They must have a `name`. An interface is composed of events, properties, and methods.

Obtaining a RAD Object Reference in Python

See the RAD Namespace in Python section.

Working With RAD Object References in Python

Once you have an object reference, you can use this object reference to interact with RAD directly. All attributes and methods defined in the IDL are accessible directly as Python object attributes that are returned by the `get_object`() function.

The following example gets a reference to a zone and then boots the zone.

Example 2-41 Python Language – Working With RAD Object References

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
...     patt = radc.ADRGlobPattern({"name" : "test-0"})
...     zone = rc.get_object(zonemgr.Zone(), patt)
...     print zone.name
...     zone.boot(None)
>>>
```

In this example, you have connected to the RAD instance, created a search for a specific object, retrieved a reference to the object, and accessed a remote property on it. No error handling occurred.

Accessing a RAD Remote Property in Python

The following example shows how to access a remote property.

Example 2-42 Python Language – Accessing a Remote RAD Property

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
...     name = rc.list_objects(zonemgr.Zone(), radc.ADRGlobPattern({"name" :
"test-0"}))
...     zone = rc.get_object(name[0])
...     for prop in zone.getResourceProperties(zonemgr.Resource("global")):
...         if prop.name == "brand":
...             print "Zone: %s, brand: %s" % (zone.name, prop.value)
...             break
...
Zone: test-0, brand: solaris
>>>
```

In this example, you have accessed the list of global resource properties of the `Zone` and searched the list of properties for the `brand` property. When you find it, print the value of the `brand` property and then terminate the loop.

RAD Event Handling in Python

This example shows how to subscribe to and handle events. The `ZoneManager` instance defines a `stateChange` event, which clients can subscribe to for information about changes in the runtime state of a zone.

Example 2-43 Python Language – Subscribing to and Handling RAD Events

```
import rad.connect as radcon
import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
import signal

def handler(event, payload, user):
   print "event: %s" % str(event)
   print "payload: %s" % str(payload)
   print "zone: %s" % str(payload.zone)
   print "old: %s" % str(payload.oldstate)
   print "new: %s" % str(payload.newstate)

with radcon.connect_unix() as rc:
    zm = rc.get_object(zonemgr.ZoneManager())
    rc.subscribe(zm, "stateChange", handler, zm)
    signal.pause()
```

In this example, you have subscribed to a single event by passing a handler and a reference to the `ZoneManager` object. The handler is invoked asynchronously by the framework with various event details and user data. The user data is an optional argument at subscription. If the user data is not provided, the handler receives `None` as the user parameter.

Python Error Handling in RAD

Python provides a exception handling mechanism and propagates RAD errors by using this mechanism. RAD delivers a variety of error codes that you can handle with `rad.client.ObjectError`. The following example shows how to handle RAD errors.

Example 2-44 Python Language – Handling RAD Errors

```
import rad.client as radc
import rad.conect as radcon
import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
import logging
import sys

logging.basicConfig(filename='/tmp/example.log', level=logging.DEBUG)
with radcon.connect_unix() as rc:
    patt = radc.ADRGlobPattern({"name" : "test-0"})
    test0 = rc.get_object(zonemgr.Zone(), patt)
    print test0.name
    try:
        test0.boot(None)
    except radc.ObjectError as ex:
        error = ex.get_payload()
        if not error:
            sys.exit(1)
        if error.stdout is not None:
            logging.info(error.stdout)
        if error.stderr is not None:
```

```
        logging.info(error.stderr)
    sys.exit(1)
```

> **Note:**
>
> With `ObjectError` exceptions, you might get a payload. This payload is present only if your interface method or property has defined an `error` element, where the payload is the content of that error. If no `error` element for the interface method (or property) is declared, then no payload exists and `error` will have a value of `None`.

Connecting in Python to a RAD Instance by Using a URI

You can use the standard URI format to connect to a RAD instance. The URI format is as follows:

scheme`://`*user*`?@`*host*`:`*port*`?auth=`*value*

scheme
(Mandatory) The supported schemes are `unix`, `rad`, `rads`, and `ssh`.

user
(Optional) The user who is connecting to the remote RAD instance. If you do not specify the user, the current user is assumed.

host
(Mandatory) The system that contains the remote RAD instance.

port
(Optional) The port number. The default port is 12302 (RAD IANA port).

auth
(Optional) The authentication method that is used to connect to the remote RAD instance. The supported values are `pam` and `gss`. If you do not specify the authentication, then `pam` is assumed. If you are using SSH as the transport protocol, you must not specify the authentication mechanism.

Example 2-45 Python Language – Connecting to a RAD Instance by Using a URI

The following example shows how to open a TCP connection as `jdoe` to the host `abc` at port 10000 with default authentication.

```
rads://jdoe@abc.example.com:10000
```

The following example shows how to open a TLS connection as `hg` to the host `abc` at the default RAD port with `gss` authentication.

```
radg://hg@abc.example.com?auth=gss
```

The following example shows how to open an SSH connection as the current user to the host `abc` at the default SSH port.

```
ssh://abc.example.com
```

The following example shows how to open a connection to a local RAD instance.

```
unix:///path
```

RAD TLS Client in Python

The Python implementation has the `rad.connect.connect_tls(host, port=RAD_PORT_TLS, locale=None, ca_certs=RAD_DEFAULT_CERT_PATHS, client_cert=None, client_key=None)` function. The `client_cert` and `client_key` arguments can point to `/etc/certs/localhost/host.crt` and `host.key`, respectively.

Generic Security Services API Transport in RAD

RAD includes a Generic Security Services API (GSS-API) transport, which allows secure administrative communication between the client and the server.

The following examples shows the configuration parameters when configuring GSS-API transport in RAD:

```
DNS domain name = example.com
RAD client = client.example.com
RAD server = server.example.com
Kerberos realm name = EXAMPLE.COM
Kerberos administrative principal = adjdoe/admin
User/principal = jdoe
```

Securing Messages Using G-RAD

The GSS-API transport allows secure messaging for RAD applications.

The GSS-API transport leverages environments that have deployed secure authentication protocols, such as Kerberos. A URI transport element is added to the existing schema to indicate the GSS-API RAD (G-RAD) transport as follows:

`radg://[`*user*`@]`*host*`[:`*port*`]`

`radg` specifies the transport indicating G-RAD support.

If the *user* is not specified, then the invoking user is utilized. In the case of Kerberos, the user is mapped into a user principal.

If the *host* is not specified, then failure is returned.

If the *port* is not specified, then the default port is 6789.

G-RAD Applications Using Kerberos

The RAD server must be configured to host Kerberos and to utilize the G-RAD transport for Kerberos. Configuring the RAD server includes creating a `rad` service principal for the system and adding the associated keys to its key table. For more information about configuring the RAD server with Kerberos, see Configuring Kerberos Clients in *Managing Kerberos in Oracle Solaris 11.4*.

After the system is configured for Kerberos, create the `rad` service principal, such as `rad/server.example.com`, on the RAD server. You can authenticate as a RAD user on the RAD client by using the `kinit` command or by authenticating through PAM with `pam_krb5`.

Using the RAD client's initial authentication through the system key table file (`/etc/krb5/krb5.keytab`), the `root` user can also be configured as a RAD user in Kerberos. The *host* service principal is used in this scenario, therefore the client must be configured as a Kerberos system. For more information, see Configuring Kerberos Clients in *Managing Kerberos in Oracle Solaris 11.4*.

To authorize RAD requests as `root`, the RAD server must also map the authenticated *host* service principal of the client to the local `root` user. For example, on the RAD server, the `/etc/krb5/krb5.conf` file is updated to include `auth_to_local_names` in the realms section as follows:

```
server# cat /etc/krb5/krb5.conf
...
[realms]
    EXAMPLE.COM = {
        ...
        auth_to_local_names = {
            host/client.example.com = root
        }
    }
```

Example 2-46 G-RAD Application Example

The following example shows G-RAD transport utilization with live zone migration as a privileged user:

```
client$ id
uid=1234567(mre) gid=1(other)
client $ profiles
Zone Configuration
Zone Migration
Basic Solaris User
All
client$ auths
solaris.admin.wusb.read,solaris.mail.mailq,solaris.network.autoconf.read,
solaris.zone.config/zone1,solaris.zone.migrate/zone1
client$ kinit
Password for mre@EXAMPLE.COM:
client$ pfexec /usr/sbin/zoneadm -z zone1 migrate radg://server
zoneadm: zone 'zone1': Using existing zone configuration on
destination.
zoneadm: zone 'zone1': Attaching zone.
zoneadm: zone 'zone1': Booting zone in 'migrating-in' mode.
zoneadm: zone 'zone1': Checking migration compatibility.
zoneadm: zone 'zone1': Performing initial copy (total 8192MB).
...
zoneadm: zone 'zone1': Migration successful.
```

3
REST APIs for RAD Clients

RESTful Interface and RAD

The RESTful interface can be accessed by any HTTP client that supports either normal TCP (or TLS) connections or UNIX domain sockets. Two RAD SMF service instances provide the access:

`rad:local`
Enables communication with local HTTP clients that can communicate over UNIX domain sockets. Enabled by default. For more information, see the `--unix-socket` option in the `curl`(1) man page.

`rad:remote`
Enables communication over normal TCP sockets with HTTPS clients. Disabled by default.

Oracle Solaris 11.4 introduces RAD HTTP authentication API version 2.0. This updated API exposes the entire PAM (Pluggable Authentication Module) conversation. Developers can take advantage of the exposed PAM conversation to improve the user experience. The Oracle Solaris Analytics Web UI application uses this updated API.

> **Note:**
>
> Communication with RAD over HTTP uses the `application/json` content type only. The RAD server will refuse to communicate if a request states a payload content type or an `Accept` content type from the client that is not `application/json`.

The following example illustrates the REST interaction with RAD. Interacting With RAD by Using the REST Authentication Module 2.0 is a similar example that uses the API that handles multiple authentication requests, such as requests from an OTP application. These examples assume that you have installed the `web/curl` developer package.

For more information about the two distinct authentication APIs, see RAD Authenticating Remote Clients.

Example 3-1 Interacting With RAD by Using the REST Authentication Module 1.0

This example assumes that you have installed the `web/curl` developer package. The example illustrates a password-based PAM authentication stack, and shows how you can use RAD with non-global zones.

1. Create a new authentication session.

   Replace the *username* and *password* with values for any user on your system.

```
# curl -X POST -c cookiejar -b cookiejar \
--header 'Content-Type:application/json'
--data '{"username":"username","password":"password","scheme":"pam","timeout":-1,
```

```
"preserve":true}' \
https://radserver.example.com/api/com.oracle.solaris.rad.authentication/1.0/
Session/
```

2. Request a list of all the zones running on your system.

```
# curl -H 'Content-Type:application/json' -X GET -b cookiejar \
https://radserver.example.com/api/com.oracle.solaris.rad.zonemgr/1.0/Zone?
_rad_detail
```

Sample response:

```
{
        "status": "success",
        "payload": [
                {
                        "href": "api/com.oracle.solaris.rad.zonemgr/1.2/Zone/
testzone1",
                        "Zone": {
                                "auxstate": [],
                                "brand": "solaris",
                                "id": 1,
                                "uuid": "b54e20c1-3ecb-407f-ad26-
befed9221860",
                                "name": "testzone1",
                                "state": "running"
                        }
                },
                {
                        "href": "api/com.oracle.solaris.rad.zonemgr/1.2/Zone/
testzone2",
                        "Zone": {
                                "auxstate": [],
                                "brand": "solaris",
                                "id": 2,
                                "uuid": "358b43ba-32f9-4f27-9efa-
de15ae4100a6",
                                "name": "testzone2",
                                "state": "running"
                        }
                }
        ]
}
```

Example 3-2 Mapping the Connection to the `root` User

Create a Python script to connect to RAD, map the connection to the `root` user, and list all non-global and kernel zones that are present on the specified system, *hostname*. Run the script as the `root` user.

Note that the `map_host_certificate_to_root` property value must be `true`. See Configuring RAD Transports to Accept X.509 Client Certificates.

Ensure that your script contains the following Python 3 example code fragment:

```
import requests
import json
from pprint import pprint
r = requests.get('https://hostname>:6788/api/com.oracle.solaris.rad.zonemgr/1.8/
Zone?_rad_detail', cert=('/etc/certs/localhost/host.crt', '/etc/certs/localhost/
host.key'))
pprint(json.loads(r.text))
```

> **Note:**
>
> RAD over HTTP/REST permits a client to provide a certificate and hence permits it to perform non-interactive authentication. This behavior requires that the HTTP client support the sending of TLS client certificates.

URI Specifications for RAD Resources

In a RESTful architecture, RAD objects are modeled as resources. A resource is an entity with a type, associated data, relationships to other resources, and a set of methods that operate on it. A URI is used to identify a resource. Resources can exist individually or as a collection. And a collection can be nested within an individual resource.

The URI format to access RAD resources can include a variety of parameters, for example:

- `https://host:port/api/{namespace}/[{version}]/{collection}[?query-params]`
- `https://host:port/api/{namespace}/[{version}]/{collection}/{coll-ID}[?query-params]`
- `https://host:port/api/{namespace}/[{version}]/{collection}/{coll-ID}/{property}[?query-params]`
- `https://host:port/api/{namespace}/[{version}]/{collection}/{coll-ID}/{sub-collection}[?query-params]`
- `https://host:port/api/{namespace}/[{version}]/{collection}/{coll-ID}/{sub-collection}/{sub-coll-ID}[?query-params]`
- `https://host:port/api/{namespace}/[{version}]/{collection}/{coll-ID}/{sub-collection}/{sub-coll-ID}/{property}[?query-params]`
- `https://host:port/api/{namespace}/[{version}]/{collection}/{coll-ID}/{sub-collection}/{sub-coll-ID}/{sub-collection}/{sub-coll-ID}/{sub-collection}......[?query-params]`

The components in the URIs are as follows:

- *namespace* – Name associated with a RAD module, generally the module API name or domain name of the RAD module.
- *version* – Optional version number that specifies the RAD module version.
- *collection* – Collection resource.
- *coll-ID* – Identifier or path to an individual resource within a collection that identifies a specific RAD instance.
- *sub-collection* – Collection nested within an individual resource. It is an interface property of type struct, a list, a dictionary, or a reference.
- *sub-coll-ID* – Identifier or path to an individual resource within a subcollection. It consists of a struct field, a list index, a dictionary key, or a reference property.
- *property* – An interface property within a specific individual resource.

Sample URI:

```
https://host:port/api/com.oracle.solaris.rad.zonemgr/1.6/Zone/testzone1?_rad_detail
```

All REST requests take the optional `_rad_detail` query parameter. If this query parameter is set to `true`, you will get the full details of an object in the response. The default setting is `false`.

In some cases, a server object does not have a name and the use of a standard URI to refer to a RAD instance does not make sense. This situation might occur when a reference to a RAD instance is returned as an error or as the result from a method. In this case, the server generates a URI path that includes a `_rad_reference` field. For example, the following is a possible URI:

```
/api/com.oracle.solaris.rad.zonemgr/1.6/Zone/_rad_reference/1234
```

The URI is valid to use in the remainder of the session but is valid only for the lifetime of a session.

URI for an Individual RAD Resource

Individual resources are identified by a URI that includes a comma-separated list of all primary keys. For example, an individual zone object might be represented by the following URI:

```
/api/com.oracle.solaris.rad.zonemgr/1.6/Zone/testzone1
```

URI for a RAD Resource Collection

Collections are identified by a URI which includes the name of the collection. For example, a zone collection object is represented by the following URI:

```
/api/com.oracle.solaris.rad.zonemgr/1.6/Zone
```

Invoking RAD Interface Methods

To invoke a method supported by an interface in a URI, include the method name and an ordered list of arguments in the request. The response includes any results or errors returned by the interface method.

Example 3-3 Listing the `anet` Properties of a Zone in RAD

The following example shows how to get the `anet` properties of a zone and the sample response.

```
# curl -H 'Content-Type:application/json' -X PUT \
https://radserver.example.com/api/com.oracle.solaris.rad.zonemgr/1.6/Zone/
testzone1/_rad_method/getResourceProperties \
--data '{"filter": {"type": "anet"}}'
```

Sample output:

```
{
        "status": "success",
        "payload": [
                {
                        "name": "linkname",
                        "value": "net0",
                        "type": "PROP_SIMPLE",
                        "listvalue": null,
                        "complexvalue": null
                },
```

```
                {
                        "name": "lower-link",
                        "value": "auto",
                        "type": "PROP_SIMPLE",
                        "listvalue": null,
                        "complexvalue": null
                },.....
....]}
```

In this example, the `getResourceProperties` method of the `Zone` interface is invoked. For more information about the methods supported by the `Zone` interface, see the `zonemgr`(3RAD) man page.

> **Note:**
>
> When using the `_rad_method` parameter, the request should be of type `PUT`.

REST Requests

A REST request is associated with an HTTP operation and can use any of the following HTTP operations based on the type of request:

- `GET` – Retrieve a resource or a collection of resources
- `POST` – Create a new resource
- `PUT` – Update a resource
- `DELETE` – Delete a resource

Because REST for RAD supports only JSON as the content type, you must include one of the following values in the HTTP header of a REST request:

- Set the value of the `Content-Type` field to `application/json`.
- Set the value of the `Accept` field to `*/*` or `application/json`.

REST Request Examples

The following examples show how to use REST to create, read, update, and delete RAD resources.

Example 3-4 Creating a Resource by Using REST

This example shows how to create a ZFS file system named `p2` in `rpool/export/home/testuser`.

Sample request:

```
# curl -H 'Content-Type:application/json' -X PUT -b cookiejar \
https://radserver.example.com/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/
_rad_method/create_filesystem \
--data '{"name":"rpool/export/home/testuser/p2"}'
```

Sample response:

```
{
        "status": "success",
```

```
        "payload": {
                "href": "/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/
_rad_reference/5889"
        }
}
```

Example 3-5 Updating a Resource by Using REST

This example shows how to update the value of the `maxbw` property for the `net0` interface.

Sample request:

```
# curl -H 'Content-Type:application/json' -X PUT -b cookiejar \
https://radserver.example.com/api/com.oracle.solaris.rad.dlmgr/1.1/Datalink/net0/
_rad_method/setProperty \
--data '{"properties":"maxbw=300","flags":1}'
```

Sample response:

```
{
        "status": "success",
        "payload": null
}
```

Example 3-6 Querying a Resource by Using REST

This example shows how to get a list of all the ZFS file systems available in `rpool`.

Sample request:

```
# curl -H 'Content-Type:application/json' -X PUT -b cookiejar \
https://radserver.example.com/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/
rpool/_rad_method/get_filesystems \
--data'{"recursive":true}'
```

Sample response:

```
{
        "status": "success",
        "payload": [
                "rpool/ROOT",
                "rpool/ROOT/solaris",
                "rpool/ROOT/solaris/var",
                "rpool/VARSHARE",
                "rpool/VARSHARE/zones",
                "rpool/VARSHARE/pkg",
                "rpool/VARSHARE/pkg/repositories",
                "rpool/export",
                "rpool/export/home",
                "rpool/export/home/testuser"
        ]
}
```

Example 3-7 Deleting a Resource by Using REST

This example shows how to delete a user named `tuser4`.

Sample request:

```
# curl -H 'Content-Type:application/json' -X PUT -b cookiejar \
https://radserver.example.com/api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/
```

```
_rad_method/deleteUser \
--data '{"username":"tuser4"}'
```

Sample response:

```
{
        "status": "success",
        "payload": null
}
```

REST Responses

All REST responses have the following basic JSON structure:

```
{
            "status":    "success" ||
            "object-specific error" ||
            "not found error" ...
            "payload":    null || <resource specific>
}
```

HTTP Status Codes and REST

Because REST requests are made over HTTP, the client receives HTTP status codes in a response. Some of the common HTTP return codes and their corresponding meaning in the context of RAD are as follows:

- `200 OK` – Request succeeded.
- `201 Created` – Request succeeded and a new resource is created.
- `204 No Content` – Request succeeded but the server did not return a message body.
- `400 Bad Request` – Request did not succeed, possibly because of a data-type mismatch or a illegal access.
- `401 Unauthorized` – Insufficient privileges.
- `404 Not found` – Specified resource was not found.

Error Responses to a RAD Request

For non-fatal errors, the server responds with information about the issue. The basic JSON structure of an error response is as follows:

```
{
         "status": text-of-the-RAD error,
         "payload": payload
}
```

In case of a `HTTP 503` error, the value is defined and returned by the RAD module that is mentioned in the request. For all other errors, the payload value has the following format:

```
{
     "Message": description-of-error,

     "HTTP Method": ("HEAD"|"GET"|"POST"|"PUT"|"DELETE"),

     "URI": full-URI-with-all-query-parameters,
```

```
    "RAD Operation": ("INVOKE"|"GETATTR"|"SETATTR"|"LOOKUP"|"LIST"|null),

    "Request payload": { Arguments provided by the client },

    "Method": name of a method for INVOKE operation,

    "Attribute": name of an attribute/property for GETATTR, SETATTR ops,

    "Pattern": URI translated to RAD list pattern for LIST operation,

    "Name": URI translated to RAD name for LOOKUP operation,

    "Object": { URI translated to RAD object (any operation but LIST)

    "Name": name of a module, also known as domain

    "Interface": name of the interface,
    },

    "Reference": RAD reference ID found in the special _rad_reference URIs,

    "Version": { Module version as found in the URI

    "Major": int,

    "Minor": int
    }
  }
```

Information that is not provided in the request or could not be decoded from the request is included in the response as a JSON `null`.

Some of the examples of possible error messages are as follows:

```
Decoding request body as JSON failed: too big integer near '18446744073709551615'

Invalid (array) argument 'arg'='[true,false,true,false]' - element [0]
- integer out of bounds (-2147483648, 2147483647)
```

RAD Authentication

With RAD, all communications between client and server are encapsulated within a connection. When a connection closes, all state associated with the connection is reclaimed by the RAD daemon. However, because RESTful interactions that happen over HTTP are stateless, a client must establish a connection and authenticate each request.

RAD authentication is performed using the `authentication` module API. This module has some special features, but in general one interacts with it much as with any other module. Oracle Solaris 11.4 delivers two major, incompatible `authentication` module versions.

RAD Authentication Module Version 1.0
Authenticates the client in one request where the client provides both username and password.
This version assumes that the server-side PAM configuration only ever requires a username and password. Use this version if that is a safe assumption, or if the same RAD client needs to interoperate with Oracle Solaris 11.3 and earlier releases.

RAD Authentication Module Version 2.0
Authenticates the client in multiple request-responses, thus fully exposing the underlying PAM conversation
Where the server-side PAM configuration may prompt the user for multiple credentials, such as OTP or RADIUS requests, you must use this version. You should use the `authentication` module version 2.0 API when the client is an interactive application rather than scripted automation. The Oracle Solaris WebUI SMF service uses the `authentication` module version 2.0 API.

Instead of having to re-authenticate for every request, RAD provides a token. When a client connects to RAD and successfully authenticates, RAD generates a unique token for the client and then services the request. At the same time, RAD stores the token and details about the client connection. On subsequent requests, if a token is supplied, RAD uses the token to retrieve the previously authenticated connection, associates it with the incoming request, and processes the request.

Because a token is generated when a client connects to RAD for the first time, the token is absent from the request. Tokens have the following characteristics:

- Tokens are a 256-bit opaque value constructed from a random number, which provides security and minimizes the likelihood of collisions.
- Tokens have a finite, configurable lifetime of up to a maximum of 24 hours. The default lifetime is 1 hour. The lifetime is configured as part of the initial authentication request. The expiry time of the token is reset or extended whenever an authenticated request is received.
- If the token received in a request is invalid or has expired, an error is returned and the client must re-authenticate.
- If the RAD slave is killed or if RAD is terminated, all tokens and their corresponding sessions are destroyed.

RAD Authenticating Remote Clients

Unlike in Oracle Solaris 11.3, RAD in Oracle Solaris 11.4 accepts REST connections locally with the `rad:local` SMF instance and remotely with the `rad:remote` SMF instance.

The `rad:remote` SMF instance is delivered disabled by default. When enabled, `rad:remote` expects RPC connections on port 12302 and HTTP/REST connections on port 6788. All remote communications with RAD are transported over TLS/HTTPS.

How to Enable a RAD Remote Client Connection

The `rad:remote` service must be enabled, and the RAD server's certificate must be trusted on the client.

1. **Enable the `rad:remote` instance.**

   ```
   # svcadm enable rad:remote
   ```

2. **Ensure that the RAD server's certificate will be trusted.**

   Perform this procedure once per client. Choose the steps for your certificate source.

   - RAD server is using a self-signed certificate.

     a. From the client, safely copy the server certificate to the client.

```
RADclient # scp user@RADserver:/etc/certs/localhost/host-ca/
hostca.crt /etc/certs/CA
```

b. Restart the `ca-certificates` service on the client.

```
RADclient # svcadm restart ca-certificates
```

- RAD server is using a CA certificate.

a. Ensure that the certificate for the issuing CA is in the `/etc/certs/CA` directory on the client.

```
RADclient # ls /etc/certs/CA
...
Example-Security_EV_RootCA1.pem
Example-Security_RootCA2.pem
Example-Security_Root_CA.pem
...
```

b. If necessary, restart the `ca-certificates` service.

```
RADclient # svcadm restart ca-certificates
```

Example 3-8 RAD Authenticating Using Version 1.0 of the Authentication Module

This example shows how to use the `authentication` module version 1.0 to connect to a remote server.

1. Establish a session and generate a token.

2. ```
   # curl -X POST -c cookiejar -b cookiejar \
   --header 'Content-Type:application/json' \
   --data
   '{"username":"username","password":"password","scheme":"pam","timeout":-1,
   "preserve":true}' \
   https://radserver.example.com/api/com.oracle.solaris.rad.authentication/1.0/
   Session/
   ```

3. If the username and password credentials are valid, you will get a response similar to the following:

   ```
   Set-Cookie: _rad_instance=26368; Path=/api; Max-Age=3600
   Set-Cookie: _rad_token=9432a53c-8034-4729-8cac-fb713a56827b; Path=/api;Max-
   Age=3600

   {
           "status": "success",
           "payload": {
                   "href": "/api/com.oracle.solaris.rad.authentication/1.0/
   Session/_rad_reference/2304"
           }
   }
   ```

   As the `Set-Cookie` implies, to resume a session, a client must present this cookie in the HTTP header as part of each future request. Because the `Set-Cookie` directive instructs the client to include this cookie in future requests, the session resumes automatically. In this example, invoking the `curl` command again with the same `cookiejar` file and a new request would result in RAD processing the new request as part of the initial session.

4. For subsequent requests, you would use the token, as shown in the following example:
   ```

```
# curl -v -X GET -c cookiejar -b cookiejar \
https://radserver.example.com/api/com.oracle.solaris.rad.zonemgr/1.0/Zone?
_rad_detail
```

The `_rad_token` cookie contains a string token that is the external representation of the session. If the token needs to be directly accessed, you can obtain the string token by reading the session's token property. This value may be used to later gain access to the session by writing the token to the session's token property.

Only the owner of a session may delete and thus invalidate the session.

Note that a session token can be used across multiple connections, which allows an authenticated client to make multiple concurrent requests.

Example 3-9 RAD Authenticating Using Version 2.0 of the Authentication Module

This example shows how to use the `authentication` module version 2.0 to connect to a remote server.

1. Establish a session and generate a token that is sent back to the client in the form of an HTTP cookie.

```
# curl -b cookiejar -c cookiejar -X POST -H 'Content-Type:application/json' \
--data '{"username":"jdoe", "preserve":true}' \
https://radserver.example.com/api/authentication/2.0/Session/
```

2. The HTTP cookie is similar to the following:

```
Set-Cookie: _rad_token=9432a53c-8034-4729-8cac-fb713a56827b; Path=/api;Max-Age=3600

{
    "status": "success",
    "payload": {
            "href": "/api/com.oracle.solaris.rad.authentication/2.0/Session/
_rad_reference/2560"
    }
}
```

Unlike the version 1.0 API, the session is not authenticated at this point. The server expects further interaction with client and user.

3. To determine the session state on the server, the client inspects it as follows:

```
# curl -b cookiejar -c cookiejar -X GET \
https://radserver.example.com/api/com.oracle.solaris.rad.authentication/2.0/
Session/_rad_reference/2560/state
```

Sample response:

```
 {
    "status": "success",
    "payload": {
            "scheme": "PAM",
            "pam": {
                    "state": "CONTINUE",
                    "messages": [
                            {
                                    "style": "PROMPT_ECHO_OFF",
                                    "message": "Password: "
                            }
                    ],
                    "responses": null
            },
```

```
            "error": null,
            "generation": 1
    }
}
```

This response tells the client that the server (PAM) expects to continue the authentication conversation (string `CONTINUE` in the JSON structure). Note that the string is under the keys `payload`, `pam`, and `state`. Also, under the key `messages`, the server tells the client application to display the message `"Password: "` and to hide the user response `"style":"PROMPT_ECHO_OFF"`.

The `messages` key can contain multiple messages. The client application must collect and respond to all of those in one subsequent HTTP request.

The possible values for `"state"` are:

`CONTINUE`

Server will send another set of messages to process

`SUCCESS`

Server successfully authenticated the session

`ERROR`

An error occurred when authenticating the session

The possible values for `style` are:

`PROMPT_ECHO_OFF`

`PROMPT_ECHO_ON`

`TEXT_INFO`

`ERROR_MSG`

For more information, see the `pam_start`(3PAM) man page.

4. The client application gathers the requested user input:

```
# curl -b cookiejar -c cookiejar -X PUT -H "Content-type: application/json" \
--data '{"value": {"pam": {"responses": ["secret"]}, "generation": 1}}' \
https://radserver.example.com/api/com.oracle.solaris.rad.authentication/2.0/
Session/_rad_reference/2560/state
```

5. The client responds to the server by updating the `state` resource similar to the following:

```
{
      "status": "success",
      "payload": {
              "scheme": "PAM",
              "pam": {
                      "state": "SUCCESS",
                      "messages": null,
                      "responses": null
              },
              "error": null,
              "generation": 2
      }
}
```

Note that the server sends a generation number with each of its JSON payload responses. The client must repeat the same key and value in any of its subsequent requests to update the `state` resource.

This example shows that the RAD server used PAM to authenticate the user successfully in just one generation. However, because of the configuration of the PAM stack, the client must repeat the `GET` and `PUT` loop: (`GET ... /state`, gather the user's unput and `PUT ... /state`) for as long as the value of the `state` key remains `CONTINUE`.

Example 3-10 Interacting With RAD by Using the REST Authentication Module 2.0

1. Create a new authentication session.

```
# curl -b cookiejar -c cookiejar -X POST --header 'Content-Type:application/json' \
--data '{"username":"jdoe", "preserve":true}' \
https://radserver.example.com/api/authentication/2.0/Session/
```

Sample response:

```
{
     "status": "success",
     "payload": {
          "href": "/api/com.oracle.solaris.rad.authentication/2.0/Session/
_rad_reference/2560"
     }
}
```

The preceding command creates a new authentication session for user `jdoe`, as shown in the response. Along with the response, the server sends HTTP cookies for the client to store and send back with every subsequent request to the server during this particular session.

2. Determine the state of the session.

The `state` key shows that PAM requires more input for successful session authentication.

```
# curl -b cookiejar -c cookiejar -X GET \
https://radserver.example.com/api/com.oracle.solaris.rad.authentication/2.0/
Session/_rad_reference/2560/state
```

Sample response:

```
{
     "status": "success",
     "payload": {
             "scheme": "PAM",
             "pam": {
                     "state": "CONTINUE",
                     "messages": [
                             {
                                     "style": "PROMPT_ECHO_OFF",
                                     "message": "Password: "
                             }
                     ],
                     "responses": null
             },
             "error": null,
             "generation": 1
     }
}
```

For information about the meaning of the response, see RAD Authenticating Remote Clients.

3. In this example, PAM is configured for password-based authentication. Therefore, the client needs to send to the server the password that the client program gathered from the user:

```
# curl -b cookiejar -c cookiejar -X PUT -H "Content-type: application/json" \
--data '{"value": {"pam": {"responses": ["secret"]}, "generation": 1}}' \
https://radserver.example.com/api/com.oracle.solaris.rad.authentication/2.0/
Session/_rad_reference/2560/state
```

Sample response:

```
{
        "status": "success",
        "payload": {
                "scheme": "PAM",
                "pam": {
                        "state": "SUCCESS",
                        "messages": null,
                        "responses": null
                },
                "error": null,
                "generation": 2
        }
}
```

The `SUCCESS` value of the `state` key indicates that the session is fully authenticated and the client can proceed with sending other requests to interact with different modules. Further success depends on the client sending back the session cookie(s), and the session cannot have expired.

4. If your system has non-global zones, send the following request.

```
# curl -H 'Content-Type:application/json' -X GET \
https://radserver.example.com/api/com.oracle.solaris.rad.zonemgr/1.6/Zone?
_rad_detail
```

Sample response:

```
{
     "status": "success",
     "payload": [
             {
                     "href": "api/com.oracle.solaris.rad.zonemgr/1.6/Zone/
testzone1",
                     "Zone": {
                             "auxstate": [],
                             "brand": "solaris",
                             "id": 1,
                             "uuid": "b54e20c1-3ecb-407f-ad26-befed9221860",
                             "name": "testzone1",
                             "state": "running"
                     }
             },
             {
                     "href": "api/com.oracle.solaris.rad.zonemgr/1.6/Zone/
testzone2",
                     "Zone": {
                             "auxstate": [],
                             "brand": "solaris",
```

```
                                        "id": 2,
                                        "uuid": "358b43ba-32f9-4f27-9efa-de15ae4100a6",
                                        "name": "testzone2",
                                        "state": "running"
                                    }
                                }
                            ]
                        }
```

REST API Reference

Although all RAD modules support REST, from a client perspective only some of the modules will be accessed over REST. The tables in this section list some of the URIs for commonly-accessed RAD modules along with sample requests.

Table 3-1 REST APIs for Datalink Management – `com.oracle.solaris.rad.dlmgr`

Resource URI	Description	Sample Request
`GET /api/com.oracle.solaris.rad.dlmgr/1.1?_rad_detail`	List the details of all the interfaces available for datalink management.	`curl -H 'Content-Type:application/json' -X GET https://radserver.example.com/api/com.oracle.solaris.rad.dlmgr/1.1?_rad_detail -b cookiejar`
`PUT /api/com.oracle.solaris.rad.dlmgr/1.1/Datalink/net0/_rad_method/getProperty`	Get the details of the `priority` property from `net0`.	`curl -H 'Content-Type:application/json' -X PUT https://radserver.example.com/api/com.oracle.solaris.rad.dlmgr/1.1/Datalink/net0/_rad_method/getProperty --data '{"properties":"priority"}' -b cookiejar`

Table 3-2 REST APIs for Kernel Statistics – `com.oracle.solaris.rad.kstat`

Resource URI	Description	Sample Request
`GET /api/com.oracle.solaris.rad.kstat/2.0?_rad_detail`	List the details for all the interfaces available for kernel statistics.	`curl -H 'Content-Type:application/json' -X GET https://radserver.example.com/api/com.oracle.solaris.rad.kstat/2.0?_rad_detail -b cookiejar`
`GET /api/com.oracle.solaris.rad.kstat/2.0/Kstat/misc,cpu_info0,cpu_info,{CPU number}?_rad_detail`	Get the information for a particular CPU on a system.	`curl -H 'Content-Type:application/json' -X GET https://radserver.example.com/api/com.oracle.solaris.rad.kstat/2.0/Kstat/misc,cpu_info0,cpu_info,0?_rad_detail -b cookiejar`
`GET /api/com.oracle.solaris.rad.kstat/2.0/Kstat/misc,vm,cpu,{CPU number}?_rad_detail`	Get the VM statistics for a particular CPU on a system.	`curl -H 'Content-Type:application/json' -X GET https://radserver.example.com/api/com.oracle.solaris.rad.kstat/2.0/Kstat/misc,vm,cpu,0?_rad_detail -b cookiejar`

Table 3-3 REST APIs for SMF Management – `com.oracle.solaris.rad.smf`

Resource URI	Description	Sample Request
`GET /api/com.oracle.solaris.rad.smf/1.0?_rad_detail`	List the details of all the interfaces available for SMF management.	`curl -H 'Content-Type:application/json' -X GET https://radserver.example.com/api/com.oracle.solaris.rad.smf/1.0?_rad_detail -b cookiejar`
`GET /api/com.oracle.solaris.rad.smf/1.0/Instance/network%2Fhttp,apache24/state`	Get the status of the `apache24` service.	`curl -H 'Content-Type:application/json' -X GET https://radserver.example.com/api/com.oracle.solaris.rad.smf/1.0/Instance/network%2Fhttp,apache24/state -b cookiejar`
`PUT /api/com.oracle.solaris.rad.smf/1.0/Instance/network%2Fhttp,apache24/_rad_method/enable`	Enable the `apache24` service.	`curl -H 'Content-Type:application/json' -X PUT https://radserver.example.com/api/com.oracle.solaris.rad.smf/1.0/Instance/network%2Fhttp,apache24/_rad_method/enable -b cookiejar --data '{"temporary": true}'`
`PUT /api/com.oracle.solaris.rad.smf/1.0/Instance/network%2Fhttp,apache24/_rad_method/disable`	Disable the `apache24` service.	`curl -H 'Content-Type:application/json' -X PUT https://radserver.example.com/api/com.oracle.solaris.rad.smf/1.0/Instance/network%2Fhttp,apache24/_rad_method/disable -b cookiejar --data '{"temporary": true}'`

Table 3-4 REST APIs for User Management – `com.oracle.solaris.rad.usermgr`

Resource URI	Description	Sample Request
`PUT /api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/_rad_method/getUser`	Get the information of a particular user on the system.	`curl -H 'Content-Type:application/json' -X PUT -b cookiejar https://radserver.example.com/api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/_rad_method/getUser --data '{"username":"testuser"}'`
`GET /api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/shells?_rad_detail`	Get the list of all the shells on the system.	`curl -H 'Content-Type:application/json' -X GET -b cookiejar https://radserver.example.com/api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/shells?_rad_detail`

Table 3-4 (Cont.) REST APIs for User Management – `com.oracle.solaris.rad.usermgr`

Resource URI	Description	Sample Request
`PUT /api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/_rad_method/addUser`	Add a user.	`curl -H 'Content-Type:application/json' -X PUT -b cookiejar https://radserver.example.com/api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/_rad_method/addUser --data '{"user":{"username":"tuser4", "userID": 9992, "groupID": 10, "inactive": 0, "min": -1, "max": -1, "warn": -1},"password":"test123"}'`
`PUT https://radserver.example.com/api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/_rad_method/deleteUser`	Delete a user.	`curl -H 'Content-Type:application/json' -X PUT -b cookiejar https://radserver.example.com/api/com.oracle.solaris.rad.usermgr/1.0/UserMgr/_rad_method/deleteUser --data '{"username":"tuser4"}'`

Table 3-5 REST APIs for ZFS Management – `com.oracle.solaris.rad.zfsmgr`

Resource URI	Description	Sample Request
`GET /api/com.oracle.solaris.rad.zfsmgr/1.0?_rad_detail`	List the details of all the interfaces available for ZFS management.	`curl -H 'Content-Type:application/json' -X GET https://radserver.example.com/api/com.oracle.solaris.rad.zfsmgr/1.0?_rad_detail -b cookiejar`
`PUT /api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/get_filesystems`	List all the ZFS file systems in `rpool`.	`curl -H 'Content-Type:application/json' -X PUT https://radserver.example.com/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/get_filesystems --data '{"recursive":true}' -b cookiejar`
`PUT /api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/get_snapshots`	List all the ZFS snapshots.	`curl -H 'Content-Type:application/json' -X PUT https://radserver.example.com/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/get_snapshots --data '{"recursive":true}' -b cookiejar`
`PUT /api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsUtil/_rad_method/valid_zfs_name`	Check whether a specified string can be used as a ZFS name.	`curl -H 'Content-Type:application/json' -X PUT https://radserver.example.com/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsUtil/_rad_method/valid_zfs_name --data '{"name":"test@test"}' -b cookiejar`

Table 3-5 (Cont.) REST APIs for ZFS Management – `com.oracle.solaris.rad.zfsmgr`

Resource URI	Description	Sample Request
`PUT https://radserver.example.com/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/create_filesystem`	Create a ZFS file system.	`curl -H 'Content-Type:application/json' -X PUT -b cookiejar https://radserver.example.com/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/create_filesystem --data '{"name":"rpool/export/home/testuser/p2"}'`

Table 3-6 REST APIs for Zone Management – `com.oracle.solaris.rad.zonemgr`

Resource URI	Description	Sample Request
`GET /api/com.oracle.solaris.rad.zonemgr/1.6?_rad_detail`	List the details of all the interfaces available for Zone management.	`curl -H 'Content-Type:application/json' -X GET -b cookiejar https://radserver.example.com/api/com.oracle.solaris.rad.zonemgr/1.6?_rad_detail`
`GET /api/com.oracle.solaris.rad.zonemgr/1.6/Zone/{zone-name}?_rad_detail`	Get the details of a zone.	`curl -H 'Content-Type:application/json' -X GET -b cookiejar https://radserver.example.com/api/com.oracle.solaris.rad.zonemgr/1.6/Zone/testzone1?_rad_detail`
`GET /api/com.oracle.solaris.rad.zonemgr/1.6/ZoneInfo?_rad_detail`	Get the details of the zone for which the interface is executing.	`curl -H 'Content-Type:application/json' -X GET -b cookiejar https://radserver.example.com/api/com.oracle.solaris.rad.zonemgr/1.6/ZoneInfo?_rad_detail`

A

RAD Module Descriptions

RAD Modules in Oracle Solaris 11.4

The following RAD modules are available in this release.

> **Note:**
>
> See Tips for Using RAD Modules for how to display the man page for a RAD module. For more information, see the `rad`(8) man page.

`com.oracle.solaris.rad.archivemgr`
Provides functionality for creating and interacting with a Unified Archive (UA).

`com.oracle.solaris.rad.authentication`
Provides functionality for authentication using PAM and session management through the use of secure tokens.

`com.oracle.solaris.rad.autoinstall`
Provides functionality for administering an Automated Install (AI) Server.

`com.oracle.solaris.rad.bemgr`
Provides functionality for getting information about Boot Environments (BE) on the system.

`com.oracle.solaris.rad.compliance_mgr`
Provides functionality for administering compliance operations, including remote assessments and central storage for reports.

`com.oracle.solaris.rad.dlmgr`
Exposes common operations for configuring and administering datalinks and data flows.

`com.oracle.solaris.rad.ips`
Exposes common package operations in IPS to RAD clients.

`com.oracle.solaris.rad.kstat`
Exposes v2 kernel statistics (kstat2) to RAD clients.

`com.oracle.solaris.rad.labelmgr`
Provides functionality for selecting and combining labels and clearances that are used for mandatory access control.

`com.oracle.solaris.rad.modules`
Provides enumeration of RAD modules.

`com.oracle.solaris.rad.odocprovider`
Provides functionality for accessing and searching Oracle Solaris online documentation.

com.oracle.solaris.rad.smf
Exposes common SMF configuration, actions, and state to RAD clients.

com.oracle.solaris.rad.sstore
Exposes `libsstore` interfaces for reading data, info, or namespaces for different SSIDs and for enabling or disabling the persistent recording of statistics and events. It also provides a batch interface to perform multiple read operations in a single request.

com.oracle.solaris.rad.sysmgr
Provides functionality for controlling system state.

com.oracle.solaris.rad.usermgr
Exposes user, group, and role administration to RAD clients.

com.oracle.solaris.rad.webuiprefs
Provides functionality for the setting and viewing of Oracle Solaris WebUI preferences.

com.oracle.solaris.rad.zfsmgr
Provides functionality for managing ZFS storage pools, ZFS datasets and ZFS snapshots.

com.oracle.solaris.rad.zonemgr
Provides functionality for the configuration and administration of zones.

com.oracle.solaris.rad.zonesbridge
Provides connectivity to a non-global zone through its global zone for RAD clients.

Index

Symbols

A

B

C

D

E

G

R

S

U

W

Z