Remote Administration Daemon Developer's Guide

Part No: E54825-02
November 2016

ORACLE

Remote Administration Daemon Developer's Guide

Part No: E54825-02

Copyright © 2012, 2016, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS. Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Référence: E54825-02

Copyright © 2012, 2016, Oracle et/ou ses affiliés. Tous droits réservés.

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf stipulation expresse de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, accorder de licence, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est livré sous licence au Gouvernement des Etats-Unis, ou à quiconque qui aurait souscrit la licence de ce logiciel pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique:

U.S. GOVERNMENT END USERS. Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est pas conçu ni n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour ce type d'applications.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d'Advanced Micro Devices. UNIX est une marque déposée d'The Open Group.

Ce logiciel ou matériel et la documentation qui l'accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers, sauf mention contraire stipulée dans un contrat entre vous et Oracle. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l'accès à des contenus, produits ou services tiers, ou à leur utilisation, sauf mention contraire stipulée dans un contrat entre vous et Oracle.

Accessibilité de la documentation

Pour plus d'informations sur l'engagement d'Oracle pour l'accessibilité à la documentation, visitez le site Web Oracle Accessibility Program, à l'adresse `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

Contents

Using This Documentation

- **Overview** – Provides information about how to use the remote administration daemon to provide programmatic access to the administration and configuration functionality of the Oracle Solaris operating system.
- **Audience** – Developers who want to use RAD to create administrative interfaces or to use interfaces published using RAD by others.
- **Required knowledge** – Experience in developing Java or Python based application interfaces.

Product Documentation Library

Documentation and resources for this product and related products are available at http://www.oracle.com/pls/topic/lookup?ctx=E53394.

Feedback

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

♦ ♦ ♦ CHAPTER 1

Introduction to the Remote Administration Daemon

The Remote Administration Daemon (RAD), referred by its acronym and command name, rad, is a standard system service that offers secure, remote administrative access to an Oracle Solaris system. This book helps developers develop client applications using RAD.

Remote Administration Daemon

Remote Administration Daemon (RAD) provides programmable interfaces, that allow 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 allows 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 system. You can use RPC, RMI, CORBA, MPI, and other technologies to build distributed applications.

An 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 allows 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 allows clients to use dynamically-typed languages such as Python.
- Using RAD interfaces, you can define properties and asynchronous event sources.

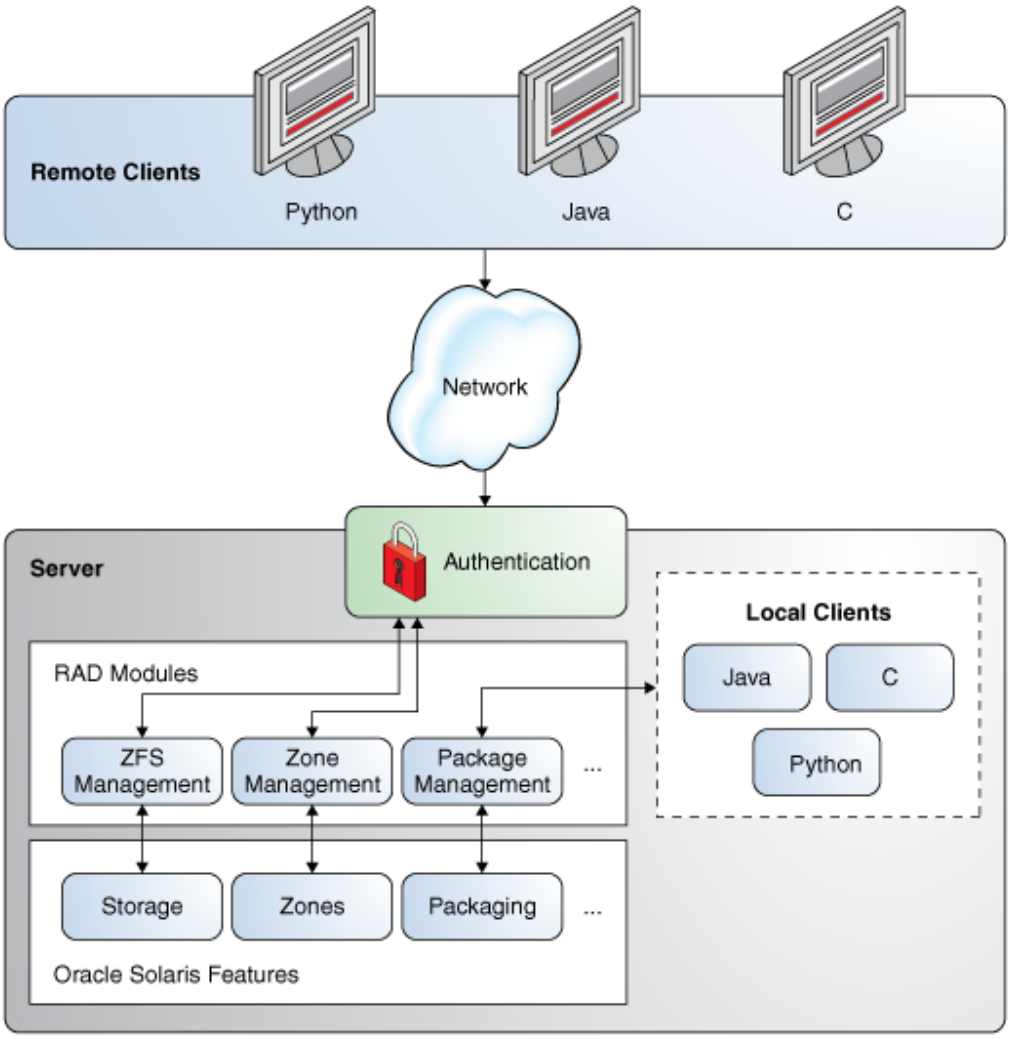
Note - Native protocol of RAD supports asynchronous procedure calls after the client is authenticated. Alternate protocols like the one based on XML-RPC, might not support asynchronous calls because of the limitations of the underlying technology.

How RAD Works

In the 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 clients can be written in C, Java, or Python.

The following figure shows the architecture of RAD.

FIGURE 1 Architecture of RAD



Overview of RAD Features

The main functionality offered by RAD is as follows:

- **Essentials**
  - Managed and configured by two SMF services, `svc:/system/rad:local` and `svc:/system/rad:remote`
  - Structured, browsable namespace.
  - Inspectable, typed, and versioned interfaces.
  - Asynchronous event sources.
  - XML-based IDL ADR supports formally defining APIs. The IDL compiler `radadrgen` generates client language bindings.
- **Security**
  - Full PAM conversation support including use of `pam_setcred`(3PAM) to set the audit context.
  - Authentication via GSSAPI in deployments where `kerberos`(5) is configured.
  - Implicit authentication by using `getpeerucred`(3C) when possible.
  - No non-local network connectivity by default. Preconfigured to use TLS.
  - Most operations automatically delegated to lesser-privileged processes.
  - Defines two authorizations (solaris.smf.manage.rad and solaris.smf.value.rad) and two Rights Profiles (RAD Management and RAD Configuration) to provide fine-grained separation of powers for managing and configuring the RAD SMF services.
    - RAD authorizations
      - `solaris.smf.manage.rad` — Grants the authorization to enable, disable, or restart the RAD SMF services.
      - `solaris.smf.value.rad` — Grants the authorization to change RAD SMF services' property values.
    - RAD rights profiles
      - RAD Management — Includes the `solaris.smf.manage.rad` authorization.
      - RAD Configuration — Includes the `solaris.smf.value.rad` authorization.
  - Generates `AUE_rad_login`, `AUE_logout`, `AUE_role_login`, `AUE_role_logout`, and `AUE_passwd` audit events.
- **Connectivity**
  - Local access via AF_UNIX sockets.
  - Remote access via TCP sockets.
  - Secure remote access via TLS sockets.
  - Captive execution with access through a pipe.
  - Connection points are completely configurable at the command line or via SMF.
- **Client support**
  - A Java language binding provides access to all defined server interfaces.
  - A Python language binding provides access to all defined server interfaces.
  - A C language binding provides access to all defined server interfaces.

- **Extension**
  - A public native C module interface supports addition of third-party content.
  - `radadrgen` can generate server-side type definitions and stubs from IDL input.
  - A native execution system can automatically run modules with authenticated user's privilege and audit context, simplifying authentication and auditing.
  - Private module interfaces permit defining new transports.

Designing RAD Components

The concepts that are fundamental to RAD are *interfaces*, objects that implement those interfaces, and the namespace in which those objects can be found and operated upon.

RAD APIs

An API is the starting point for designing a new RAD component. An API consists of a collection of other subsidiary components: derived types and interfaces. An API is versioned so that a client can specify which version of an API to interact with.

The users of the API fall into two broad categories:

- Administrators
- Developers

Accommodating the desires of consumers in these two categories within one interface is difficult. The first group desire task-based APIs which match directly onto well-understood and defined administrative activities. The second group desire detailed, operation-based interfaces which may be aggregated to better support unusual or niche administrative activities.

For any given subsystem, you can view existing command-line utilities (CLIs) and libraries (APIs) as expressions of the `rad` APIs, which are required. The CLIs represent the task-based administrative interfaces and the APIs represent the operation-based developer interfaces. The goal in using RAD is to provide interfaces that address the lowest-level objectives of the audience. If the audience are administrators (task-based), this effort could translate to matching existing CLIs. If the audience are developers, this effort could mean significantly less aggregation of the lower-level APIs.

APIs are the primary deliverable of a RAD module. The API acts as the name root for all components of the API, defining a namespace which identifies objects to client. APIs are versioned and a single RAD instance is capable of offering multiple major versions of APIs to

different clients. RAD modules are a grouping of interfaces, events, methods, and properties which enable a user to interact with a subsystem.

When exposing the elements of a subsystem consider carefully how existing functions can be grouped together to form an interface. Imperative languages, such as C, tend to pass structures as the first argument to functions, which provides a clear indicator as to how best to group functions into APIs.

Version

A version element is required for all APIs. See “RAD Interface Versioning” on page 24 for more details about API versions.

API Namespace and Restricted Names

An API defines a namespace in which all top-level elements are defined. Names of components must be unique. Names must not begin with “`_rad`” because this string is reserved for toolchain provided functionality.

Synchronous and Asynchronous Invocation

All method invocations in RAD are synchronous. Asynchronous behavior can be obtained by adopting a design pattern that relies on the use of events to provide notifications. For more information, see “Synchronization” on page 81.

Legacy Constraints

Some CLIs contain processing capabilities that are not accessible from an existing API. Such constraints must be considered in the RAD API design.

Do not duplicate the functionality in the new RAD interface, which would introduce redundancy and significantly increase maintenance complexity. One particular area where RAD interface developers need to be careful is to avoid duplication around parameter checking and transformation. This duplication is likely to be a sign that existing CLI functionality should be migrated to an API.

RAD modules must be written in C. Some subsystems, for example, those written in other languages, have no mechanism for a C module to access API functionality. In these cases, RAD module creators must access whatever functionality is available in the CLI or make a potentially significant engineering effort to access the existing functionality, for example, rewriting existing code in C, embedding a language interpreter in their C module, and the like.

Client Library Support

RAD modules are designed to have a language agnostic interface. However, you might want to provide additional language support through the delivery of a language-specific extension. This type of deliverables should be restricted in use. The main reason for their existence is to help improve the fit of an interface into a language idiom.

API Design Examples

Combining the tools described so far in this document to construct an API with a known design can be a challenge. Several possible solutions for a particular problem are often available. The examples in this section illustrate the best practices.

Note - This is only an example. This means it does not reflect the user management modules that is in Oracle Solaris.

User Management Example

Object or interface granularity is subjective. For example, imagine an interface for managing a user. The user has a few modifiable properties:

TABLE 1 Example User Properties

Property	Type
name	string
shell	string
admin	boolean

The interface for managing this user might consist solely of a set of attributes corresponding to the above properties. Alternatively, it could consist of a single attribute that is a structure containing fields that correspond to the properties, possibly more efficient if all properties are usually read or written together. The object implementing this might be named as follows:

```
com.example.users:type=TheOnlyUser
```

If instead of managing a single user you need to manage multiple users, you have a couple of choices. One option would be to modify the interface to use methods instead of attributes, and to add a "user" argument to the methods, for example:

```
setUserAttributes(username, attributes) throws UserError
attributes getUserAttributes(username) throws UserError
```

This example is sufficient for a single user, and provides support to other global operations such as adding a user, deleting a user, getting a list of users and so on. You might want to give it a more appropriate name, for example:

```
com.example.users:type=UserManagement
```

However, suppose there were many more properties associated with the user and many more operations you would want to do with a user, for example, sending them email, giving them a bonus and so on. As the server functionality grows, the UserManagement's API grows increasingly cluttered. It would accumulate a mixture of global operation and per-user operations, and the need for each per-user operation to specify a user to operate on, and specify the errors associated with not finding that user, would start looking redundant.

```
username[] listUsers()
addUser(username, attributes)
giveRaise(username, dollars) throws UserError
fire(username) throws UserError
sendEmail(username, message) throws UserError
setUserAttributes(username, attributes) throws UserError
attributes getUserAttributes(username) throws UserError
```

A cleaner alternative would be to separate the global operations from the user-specific operations and create two interfaces. The UserManagement object would use the global operations interface:

```
username[] listUsers()
addUser(username, attributes)
```

A separate object for each user would implement the user-specific interface:

```
setAttributes(attributes)
attributes getAttributes()
giveRaise(dollars)
fire()
sendEmail(message)
```

Note - If fire operates more on the namespace than the user, it should be present in UserManagement where it would need to take a username argument.

Finally, the different objects would be named such that the different objects could be easily differentiated and be directly accessed by the client:

```
com.example.users:type=UserManagement
com.example.users:type=User,name=ONeill
com.example.users:type=User,name=Sheppard
...
```

This example also highlights a situation where the RAD server may not want to enumerate all objects when a client issues a `LIST` request. Listing all users may not be particularly expensive,

but pulling down a list of potentially thousands of objects on every `LIST` call will not benefit the majority of clients.

RAD Interface

An interface defines how a RAD client can interact with an object. An object implements an interface, providing a concrete behavior to be invoked when a client makes a request.

The primary purpose of RAD is to consistently expose the various pieces of the system for administration. Not all subsystems are alike. However, each has a data and state model tuned to the problems they are solving. Although there are major benefits to using a common model across components when possible, uniformity comes with trade-offs. The increased inefficiency and client complexity, and risk of decreased developer adoption, often warrant using an interface designed for problem at hand.

An interface is a formal definition of how a client may interact with a RAD server object. An interface may be shared amongst several objects, for example, when maintaining a degree of uniformity is possible and useful, or may be implemented by only one. A RAD interface is analogous to an interface or pure abstract class in an object oriented programming language. In the case of `rad`, an interface consists of a name, the set of features a client may interact with, optionally a set of derived types referenced by the features, and a version. The features that are supported includes:

- Methods, which are procedure calls made in the context of a specific object
- Properties, which are functionally equivalent to methods but bear different semantics
- Asynchronous event sources

Name

Each interface has a name. This name is used by the toolchain to construct identifier names when generating code. When naming an API, interface, or object, module developers have broad leeway to choose names that make sense for their modules. However, some conventions can help avoid pitfalls that might arise when retrieving objects from the RAD server.

Object Names

The domain portion of RAD object names follows a reverse-dotted naming convention that prevents collisions in `rad`'s flat object namespace. This convention typically resembles a Java package naming scheme:

```
com.oracle.solaris.rad.zonemgr
```

```
com.oracle.solaris.rad.usermgr
org.opensolaris.os.rad.ips
...
```

To distinguish a `rad` API from a native API designed and implemented for a specific language, include a "`rad`." component in the API name.

With the goal of storing objects with names consumers would expect, APIs, and the domains of the objects defined within them, should share the same name. This practice makes the mapping between the two easily identifiable by both the module consumer and module developer.

With the same goal of simplicity, identifying an interface object is made easier by adhering to a "type=interface" convention within the object name.

Applying both conventions, a typical API will look like the following example.

```
    <api  xmlns="http://xmlns.oracle.com/radadr"
       name="com.oracle.solaris.rad.zonemgr">
      <version major="1" minor="0"/>
      <interface name="ZoneInfo"> <!-- Information about the current zone  -->
      <property name="name" access="ro" type="integer"/>

          ...

        </interface>
</api>
```

Within the module, the API appears as follows:

```
int
_rad_init(void)

    {
    ...
          adr_name _t *zname = adr_name_vcreate(MOD_DOMAIN, 1, "type", "ZoneInfo");
          conerr_t cerr = rad_cont_insert_singleton(&rad_container, zname,
 &interface_ZoneInfo_svr);
          adr_name_rele(zname);


         if (cerr != CE_OK) {
             rad_log(RL_ERROR, "failed to insert module in container");
             return(-1);
         }
        return (0);
}
```

On the consumer side (Python), the API appears as follows:

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

```
import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr

# Create a connection and retrieve the ZoneInfo object
with radcon.connect_unix() as rc:
    zinfo = rc.get_object(zonemgr.ZoneInfo())
    print zinfo.name
```

Case

In an effort to normalize the appearance of like items across development boundaries, and to minimize the awkwardness in generated language-specific interfaces, several case strategies have been informally adopted.

Module	The base of the API/domain name. For a module describing an interface *domain.prefix.base*.adr, module spec files should be named *base*.adr, and the resulting shared library mod_*base*.so. Examples: ■ /usr/lib/rad/interfaces/zonemgr/version/1/zonemgr.adr ■ /usr/lib/rad/module/mod_zonemgr.so
API	Reverse-dotted domain, all lowercase. Examples: ■ com.oracle.solaris.rad.usermgr ■ com.oracle.solaris.rad.zonemgr
Interface, struct, union, enum	Non-qualified, camel case, starting with uppercase. Examples: ■ Time ■ NameService ■ LDAPConfig ■ ErrorCode
Enum value and fallback	Non-qualified, uppercase, underscores. Examples: ■ CHAR ■ INVALID_TOKEN ■ REQUIRE_ALL
Interface property and method, struct field, event	Non-qualified, camel case, starting with lowercase. Examples: ■ count ■ addHostName ■ deleteUser

Features

The common thing between the three feature types — methods, attributes, and events — is that they are named. The names of all three feature types exist in the same interface namespace and must therefore be unique. For example, you cannot have both a method and an attribute that is called `foo`. This exclusion avoids the majority of conflicts that could arise when trying to naturally map these interface features to a client environment. As in the API namespace, features must not begin with "`_rad`" because this string is reserved for use by the RAD toolchain.

Note - Enforcing a common namespace for interface features is not always enough. Some language environments place additional constraints on naming. For example, a Java client will see an interface with synthetic methods of the form `get`*function_name*, `set`*function_name*, or `is`*function_name* for accessing attribute *function_name* that must coexist with other method names. Explicitly defining methods with those names might cause a conflict.

Methods

A method is a procedure call made in the context of the object it is called on. In addition to a name, a method may define a return type, can define zero or more arguments, and may declare that it returns an error, optionally with an error return type.

If a method does not define a return type, it returns no value. It is effectively of type void. If a method defines a return type and that type is permitted to be nullable, the return value may be defined to be nullable.

Each method argument has a name and a type. If any argument's type is permitted to be nullable, that argument might be defined to be nullable.

If a method does not declare that it returns an error, it theoretically cannot fail. However, because the connection to RAD could be broken either due to a network problem or a catastrophic failure in RAD itself, all method calls can fail with an I/O error. If a method declares that it returns an error but does not specify a type, the method may fail due to API-specific reasons. Clients will be able to distinguish this failure type from I/O failures.

Finally, if a method also defines an error return type, data of that type may be provided to the client in the case where the API-specific failure occurs. Error payloads are implicitly optional, and must therefore be of a type that is permitted to be nullable.

Note - Method names cannot be overloaded.

The following are the guidelines for methods:

- Methods provide mechanisms for examining and modifying administrative state.

- Consider grouping together existing native APIs into aggregated RAD functions which enable higher order operations to be exposed.
- Follow established good practice for RPC style development. RAD is primarily for remote administration, and avoiding excessive network load is good practice.

Attributes

An attribute is metaphorically a property of the object. Attributes have the following characteristics:

- A name
- A type
- A definition as read-only, read-write, or write-only
- Like a method may declare that accessing the attribute returns an error, optionally with an a error return type

Reading a read-only or read-write attribute returns the value of that attribute. Writing a write-only or read-write attribute sets the value of that attribute. Reading a write-only attribute or writing a read-only attribute is invalid. Clients may treat attempts to write to a read-only attribute as a write to an attribute that does not exist. Likewise, attempts to read from a write-only attribute may be treated as an attempt to read from an attribute that does not exist.

If an attribute's type is permitted to be nullable, its value may be defined to be nullable.

An attribute may optionally declare that it returns an error, with the same semantics as declaring (or not declaring) an error for a method. Unlike a method, an attribute may have different error declarations for reading the attribute and writing the attribute.

Attribute names may not be overloaded. Defining a read-only attribute and a write-only attribute with the same name is not valid.

Given methods, attributes are arguably a superfluous interface feature. Writing an attribute of type X can be implemented with a method that takes one argument of type X and returns nothing, and reading an attribute of type X can be implemented with a method that takes no arguments and returns a value of type X. Attributes are included because they have slightly different semantics.

In particular, an explicit attribute mechanism has the following characteristics:

- Enforces symmetric access for reading and writing read-write attributes.
- Can be easily and automatically translated to a form natural to the client language-environment.
- Communicates more about the nature of the interaction. Reading an attribute ideally should not affect system state. The value written to a read-write attribute should be the value returned on subsequent reads unless an intervening change to the system effectively *writes* a new value.

Events

An event is an asynchronous notification generated by RAD and consumed by clients. A client might subscribe to events by name to register interest in them. The subscription is performed on an object which implements an interface. In addition to a name, each event has a type.

Events have the following characteristics:

- Sequential
- Volatile
- Guaranteed

A client can rely on sequential delivery of events from a server as long as the connection to the server is maintained. If the connection fails, then events will be lost. On reconnection, a client must resubscribe to resume the flow of events.

Once a client has subscribed to an event, event notifications will be received until the client unsubscribes from the event. On receipt of a subscribed event, a client receives a payload of the defined type.

The guidelines for an event are:

- The module is responsible for providing a sequence number. Monotonically increasing sequence numbers are recommended for use, since these will be of most potential use to any clients.
- Consider providing mechanisms for allowing a client to throttle event generation.
- Carefully design event payloads to minimize network load.
- Do not try to replicate the functionality of network monitoring protocols such as SNMP.

RAD Commitment Levels

To solve the problem of different features being intended for different consumers, RAD defines two commitment levels: private and committed. All API components: derived types, interfaces and the various interface sub-components (method, attribute, and event) define their commitment level independently.

Commitment levels provide hints to API consumers about the anticipated use and expected stability of a feature. A feature with a commitment of *committed* can be used reliably. The *private* features, are likely to be subject to change and represent implementation details not intended for public consumption.

RAD Interface Versioning

RAD interfaces are versioned for the following reasons:

- APIs change over time.
- A change to an API might be incompatible with existing consumers.
- A change might be compatible with existing consumers but new consumers might not be able to use the API that was in place before the change occurred.
- Some features represent committed interfaces whose compatibility is paramount, but others are private interfaces that are changed only in lockstep with the software that uses them.

RAD Version Numbering

The first issue is measuring the compatibility of a change. RAD uses a simple `major.minor` versioning scheme. When a compatible change to an interface is made, its minor version number is incremented. When an incompatible change is made, its major version number is incremented and its minor version number is reset to 0.

In other words, an implementation of an interface that claims to be version X.Y (where X is the major version and Y is the minor version) must support any client expecting version X.Z, where Z <= Y.

The following interface changes are considered compatible:

- Adding a new event
- Adding a new method
- Adding a new attribute
- Expanding the access supported by an attribute, for example, from read-only to read-write
- A change from nullable to non-nullable for a method return value or readable property, that is, decreasing the range of a feature
- A change from non-nullable to nullable for a method argument or writable property, that is, increasing the domain of a feature

The following interface changes are considered incompatible:

- Removing an event
- Removing a method
- Removing an attribute
- Changing the type of an attribute, method, or event
- Changing a type definition referenced by an attribute, method, or event
- Decreasing the access supported by an attribute, for example, from read-write to read-only
- Adding or removing method arguments
- A change from non-nullable to nullable for a method return value or readable property, that is, increasing the range of a feature
- A change from nullable to non-nullable for a method argument or writable property, that is, decreasing the domain of a feature

Note - An interface is more than just a set of methods, attributes, and events. Associated with those features are well-defined behaviors. If those behaviors change, even if the structure of the interface remains the same, a change to the version number might be required.

A RAD client can access version information from a client binding. The mechanism for accessing the information depends on the client language like C, Java, and Python. For example, in Python, the `rad.client` module contains the `rad_get_version()` function, which may be used to get the version of an API.

RAD Namespace

The namespace acts as RAD's gatekeeper, associating a name with each object, dispatching requests to the proper object, and providing meta-operations that enable the client make queries about what objects are available and what interfaces they implement.

A RAD server may provide access to several objects that in turn expose a variety of different components of the system or even third-party software. A client merely knowing that interfaces exist, or even that a specific interface exists, is not sufficient. A simple, special-purpose client needs some way to identify the object implementing the correct interface with the correct behavior, and an adaptive or general-purpose client needs some way to determine what functionality the RAD server has made available to it.

RAD organizes the server objects it exposes in a namespace. Much like files in a file system, objects in the RAD namespace have names that enable clients to identify them, can be acted upon or inspected using that name, and can be discovered by browsing the namespace. Depending on the point of view, the namespace either is the place one goes to find objects or the intermediary that sits between the client and the objects it accesses. Either way, it is central to interactions between a client and the RAD server.

Naming

Unlike a file system, which is a hierarchical arrangement of simple filenames, RAD adopts the model used by JMX and maintains a flat namespace of structured names. An object's name consists of a mandatory reverse-dotted domain combined with a non-empty set of key-value pairs.

Equality

Two names are considered equal if they have the same domain and the same set of keys, and each key has been assigned the same value.

Patterns

Some situations call for referring to groups of objects. In these situations, a glob style pattern, or a regex style pattern should be used. For more information, see “Sophisticated Searching” on page 35.

Data Types Supported in RAD

All data returned, submitted to, or obtained from RAD APIs adheres to a strong typing system similar to that defined by XDR. For more information about XDR, see the XDR (`http://tools.ietf.org/rfc/rfc4506.txt`) standard. This makes it simpler to define interfaces that have precise semantics, and makes server extensions (which are written in C) easier to develop. Of course, the rigidity of the typing exposed to an API's consumer is primarily a function of the client language and implementation.

Base Types

RAD supports the following base types:

`boolean`	A boolean value (true or false).
`integer`	A 32-bit signed integer value.
`uinteger`	A 32-bit unsigned integer value.
`long`	A 64-bit signed integer value.
`ulong`	A 64-bit unsigned integer value.
`float`	A 32-bit floating-point value.
`double`	A 64-bit floating-point value.
`string`	A UTF-8 string.
`opaque`	Raw binary data.
`secret`	An 8-bit clean “character” array. The encoding is defined by the interface using the type. Client/server implementations may take additional steps, for example, zeroing buffers after use, to protect the contents of secret data.

`time`	An absolute UTC time value.
`name`	The name of an object in the RAD namespace.
`reference`	A reference to an object.

Derived Types

In addition to the base types, RAD supports the following derived types:

- An enumeration is a set of user-defined tokens. Like C enumerations, RAD enumerations may have specific integer values associated with them. Unlike C enumerations, RAD enumerations and integers are not interchangeable. Among other things, this aspect means that an enumeration data value may not take on values outside those defined by the enumeration, which precludes the common but questionable practice of using enumerated types for bitfield values.
- An array is an ordered list of data items of a fixed type. Arrays do not have a predefined size.
- A structure is a record consisting of a fixed set of typed, uniquely named fields. A field's type may be a base type or derived type, or even another structure type.

Derived types offer almost unlimited flexibility. However, one important constraint imposed on derived types is that recursive type references are prohibited. Thus, complex self-referencing data types, for example, linked lists or trees, must be communicated after being mapped into simpler forms.

Optional Data

In some situations, data might be declared as nullable. Nullable data can take on a “non-value”, for example, `NULL` in C, `None` in Python, or `null` in Java. Conversely, non-nullable data cannot be `NULL`. Only data of type opaque, string, secret, array, or structure might be declared nullable. Additionally, only structure fields and certain API types can be nullable. Specifically, array data cannot be nullable because the array type is actually more like a list than an array.

♦ ♦ ♦ CHAPTER 2

Connecting to RAD

RAD provides support for three client language environments: C, Java, and Python.

The examples in this chapter are snippets of the code.

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

There is a list of `#include` statements at the beginning of each example to show the headers required for that specific functionality.

Note - A lot of these examples are based on the example zonemgr interface. Refer to the sample in, Appendix A, "`zonemgr` ADR Interface Description Language Example" for this module to assist in your understanding of the examples.

Connecting to RAD in C

The RAD instances can establish connections 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 two common arguments: a boolean to specify whether the connection must be multithreaded and a *locale* to use for the connection. As a best practice, set the boolean value as *TRUE*. 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 Instance

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 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 boolean, to determine if the connection must be multithreaded
- A string, locale for the connection

If the value of socket path is `NULL`, the default RAD UNIX socket path is used. As a best practice, run the connection in multithreaded mode. If the value of *locale* is `NULL`, the *locale* of the local client system is used.

EXAMPLE 1 Creating a Local Connection

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

// do something with conn
```

Connecting to a Remote Instance and Authenticating

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 using `rc_pam_login()` function. The client application must use `#include <rad/client/1/pam_login.h>` header and links to the pluggable authentication module (PAM) C binding library, `/usr/lib/rad/client/c/libpam_client.so`.

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

EXAMPLE 2 Remote Connection over TCP IPv4 on Port 7777

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

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

if (conn !=NULL) {
```

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

Connecting to a RAD Instance by Using an URI in C

You can use an uniform resource identifier (URI) to connect to a local or remote RAD instance. For more information, see “Connecting to a RAD Instance by Using an URI” on page 66.

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)`
- `rc_credentials_t *rc_alloc_gss_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 on one of the `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 authentication for a remote connection using PAM, 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 using the `rc_connect_uri()` function. This returns a `rc_conn_t()` function, that can be used to establish the connection using `rc_connect_unix()`,

`rc_connect_tcp()`, or other functions. You can use all the other informative functions to interact with the allocated structure and obtain useful information. The various `rc_free_uri()` functions can be used to clean the memory after you finish using the structures.

RAD Namespace in C

Most RAD objects that are represented in the ADR document as *<interfaces>* are named and can be found by searching the RAD namespace. The key point to note is that to access a RAD object, you need a proxy, which is used to search the RAD namespace. This capability is provided by an interface proxy class, which is defined in each interface's binding module. The key point to note is that to access a RAD object, you should use the list and lookup functions provided by a module's client binding library (`<module>_<interface>__rad_list()`, `<module>_<interface>__rad_lookup()`. These functions also provide the option to do either strict or relaxed versioning.

The proxy automatically provides the base name and version details by using functions 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 and these objects might not be named. If the objects do not have a name, they are anonymous. Such objects cannot be looked up in the RAD namespace and the interface itself will provide access mechanisms that make it simple to interact with the anonymous objects.

Creating a Name for an Object

You can create a name for a `zonemgr` Zone instance as follows:

```
adr_glob_vcreate("com.oracle.solaris.rad.zonemgr", 2, "type",
    "Zone", "name", "zone-1");
```

Searching for Objects

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 a interface type.

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

Obtaining a Reference to a Singleton

A module developer creates a singleton to represent an interface and 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 3 Obtaining a Reference to a 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, B_TRUE, 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 this 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. Once you have the remote reference, you can access the properties with the *module_interface*`__get_<property>()` function.

Listing RAD Instances of an Interface

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. A module provides a list function for each of its interfaces in the form, *module_interface*`__rad_list()`.

EXAMPLE 4 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, B_TRUE, 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

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 5 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, B_TRUE, 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 Searching

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

EXAMPLE 6 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

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 search all the zones with a *name* that begins with *test* as follows.

EXAMPLE 7 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

You can also use 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 8 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 RAD instance that you are connected. The expression is compiled and executed in the server.

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:

- Enums
  - Values
- Structs
  - Fields
- Dictionary
- Interfaces
  - Properties
  - Methods
  - Events

These components are all defined in an ADR Interface Description Language 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 3, “Abstract Data Representation”. Brief descriptions of each component follows.

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 3, “Abstract Data Representation”. The following sections describe each component.

Enumerations

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

Using Enumeration Types

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 9 zonemgr ErrorCode Enumeration

```
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;
```

Structs

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 Struct Types

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 10 The 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

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.

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.

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.

Interfaces

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 an Object Reference

See the "RAD Namespace in C" on page 32 section.

Working With Object References

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 11 Working With 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

The following example shows to access a remote property.

EXAMPLE 12 Accessing a Remote Property

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

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, B_TRUE, 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

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

EXAMPLE 13 Subscribing and Handling 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 subscribe to the single event and pass in a handler and a handle for the `ZoneManager` object. The handler is invoked asynchronously by the framework with the various event details and the supplied user data (the user data in this case being `NULL`).

RAD Error Handling

The list of possible errors are defined by the `rc_err_t()` enumeration. RAD delivers a variety of errors, but the error which requires additional handling is `rc_err_t()`, value `RCE_SERVER_OBJECT`. The following snippet shows how it can be used.

EXAMPLE 14 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 - With the `rc_err_t` value of `RCE_SERVER_OBJECT` you might get a payload. This payload is only present if your interface method or property has defined an "error" element, in which case the payload is the content of that error. If there is no "error" element for the interface method (or property), then there is no payload and there will be no error reference argument for the method or property `get` or `set` functions.

Java Client

The public Java interfaces are exported in the following packages:

- `com.oracle.solaris.rad.client` – The client implementation of the RAD protocol plus 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. Refer to the sample in, Appendix A, "`zonemgr` ADR Interface Description Language Example" for this module to assist in your understanding of the examples.

Connecting to RAD in Java

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

Connecting to a Local Instance

You can connect to a local instance 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 15 Creating a Local Connection

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

Connection con = Connection.connectUnix();

//do something with con
```

Connecting to a Remote Instance and Authenticating

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 (`RadPamHandler`) 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.

Here is an example for Remote Connection to a TCP instance on port 7777.

EXAMPLE 16 Remote Connection to a TCP Instance on Port 7777

```
import com.oracle.solaris.rad.client.RadPamHandler;
```

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

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

Connecting to a RAD Instance by Using an URI in Java

You can use an URI to connect to a local or remote RAD instance. You can use the class `URIconnection` in Java for connecting using an URI. For more information, see "Connecting to a RAD Instance by Using an URI" on page 66.

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 allows you to use a proxy to search the RAD namespace. 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 and are referred as anonymous objects. Anonymous objects cannot be looked up in the RAD namespace and the interface provides access methods that make it simple to interact with the anonymous objects.

Creating a Name for an Object

The names are changed to be represented by a domain string and a Map<String, String> for the key/value pairs. Thus, 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 Objects

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

Obtaining Reference to a Singleton

A module developer creates a singleton to represent an interface, and 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 17 Obtaining Reference to a 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

Once 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 Instances of an Interface

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 18 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());
}
```

Obtaining a Remote Object Reference From a Name

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

EXAMPLE 19 Obtaining a Remote Object Reference From a 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 Searching

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 20 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

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 21 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

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 22 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

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 23 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 in the server.

Interface Components in Java

An API is defined by a module developer and contains a variety of components designed to accomplish a task. These components are:

- 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 3, “Abstract Data Representation”. Brief descriptions of each component follows.

Enumerations

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

Using Enumeration Types

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

EXAMPLE 24 Using Enumerations

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

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

Structs

Structs are used to define new types and are composed from existing built-in types and other user defined types. In essence, they are simple forms of interfaces. They do not have methods or events and are not present in the RAD namespace.

Using Struct Types

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 25 Using 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 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” on page 72.

The following example shows how to read and write a property defined in Example 49, “Defining a Dictionary,” on page 72.

```
//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);
```

Interfaces

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 an Object Reference

For more information, see "RAD Namespace in Java" on page 46.

Working With Object References

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 26 Invoking a 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 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.

Accessing a Remote Property

Accessing a remote property is just as simple as using a remote method.

EXAMPLE 27 Accessing a 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 accessed the list of `Zone` global resource properties and printed out the name and value of every Property.

RAD Event Handling

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 28 Subscribing and Handling 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.

RAD Error Handling

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 29 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, in which case the payload is the content of that error. If there is no "error" element for the interface method (or property), then there is no payload and it will have a value of null.

Python Client

The public interfaces are exported in the following three modules:

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

Note - A lot of these examples are based on the example zonemgr interface. Refer to the sample in, Appendix A, "`zonemgr` ADR Interface Description Language Example" for this module to assist in your understanding of the examples.

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

EXAMPLE 30 Accessing Help for a Binding Module

```
user@host1:/var/tmp# python
Python 2.6.8 (unknown, Feb 5 2013, 00:27:10) [C] 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 Instance

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 31 Creating a Local Connection

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

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

Connecting to a Remote Instance and Authenticating

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 32 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.pam_login("garypen", "xxxpasswordxxx")
>>> <now authenticated and can use this connection>
>>> rc.close()
>>> print rc
<closed RadConnection>
>>>
```

Connecting to a RAD Instance by Using an URI in Python

You can use an URI to connect to a local or remote RAD instance. You 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 to a RAD Instance by Using an URI” on page 66.

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 which schemes are to be recognised

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>* are named and can be found 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 allows you to use a proxy to search the RAD namespace. 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 and these objects might not be named. These objects are anonymous if they don't have a name. Such objects cannot be looked up in the RAD namespace and the interface itself will provide access mechanisms that make it simple to interact with the anonymous objects.

Creating a Name for an Object

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 and it removes any possible issues in processing names where values contain certain characters (for example, ',' and '=').

Searching for Objects

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

Obtaining a Reference to a Singleton

A module developer creates a singleton to represent an interface, and 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 33 Obtaining a Reference to a 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 using a proxy instance. Once you have the remote reference, you can access properties and method directly as you would with any Python object.

Listing RAD Instances of an Interface

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 34 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 Remote Object Reference From a Name

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 35 Obtaining a 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
>>>
```

Sophisticated Searching

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 a 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 36 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 refining the search by extending the name definition. The `ADRGlobPattern` class takes a *key: value* dictionary and extends the name used for the search.

Glob Pattern Searching

You can use a glob pattern to find zones with wildcard pattern matching. Keys and values in the pattern may contain *, which is interpreted as wildcard pattern matching. The following example shows how to find all zones with a *name* which begins with *test*.

EXAMPLE 37 Using Glob Patterns with Wildcards

```
>>> 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

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 38 Using Regex 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.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 in the server.

Interface Components in Python

An API is defined by a module developer and contains a variety of components designed to accomplish a task. These components are:

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

These components are all defined in ADR Interface Description Language 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 3, “Abstract Data Representation”. Brief descriptions of each component follows.

Enumerations

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

Using Enumeration Types

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

EXAMPLE 39 Using Enumerations

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

Structures

Structures (Structs) are used to define new types and are composed from existing built-in types and other user defined types. In essence, they are simple forms of interfaces: no methods or events and they are not present in the RAD namespace.

Using Struct Types

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 can be interacted with directly once the binding is imported.

EXAMPLE 40 Using 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

You can use the inbuilt dictionary type in Python. For example, the Python code to set the sample dictionary property as defined in Example 49, “Defining a Dictionary,” on page 72 can be as follows:

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

Interfaces

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 an Object Reference

See the “RAD Namespace in Python” on page 58 section.

Working With Object References

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 of the python objects that are returned by the `get_object()` function.

Here is an example which gets a reference to a zone and then boots the zone.

EXAMPLE 41 Working With 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 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. This does not include error handling.

Accessing a Remote Property

The following example shows how to access a remote property.

EXAMPLE 42 Accessing a Remote 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

This example shows how to subscribe 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 43 Subscribing and Handling 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 provided the 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.

RAD Error Handling

Python provides a exception handling mechanism and RAD errors are propagated by using this mechanism. RAD delivers a variety of errors, but the which requires handling is `rad.client.ObjectError`. The following example shows how to handle RAD errors.

EXAMPLE 44 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 is only present if your interface method or property has defined an error element, in which case the payload is the content of that error. If there is no error element for the interface method (or property), then there is no payload and it will have a value of `None`.

Connecting to a RAD Instance by Using an 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 45 Connecting to a RAD Instance by Using an URI

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

```
rad://john@abc.example.com:10000
```

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

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

The following example shows how to open an SSH connection as the current user to the system `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
```

♦ ♦ ♦ CHAPTER 3

Abstract Data Representation

The data model used by RAD is known as the Abstract Data Representation (ADR). This data model defines a formal IDL for describing types and interfaces supplies a toolchain for operating on that IDL and provides libraries used by `rad`, its extension modules, and its clients.

ADR Interface Description Language

The APIs used by RAD are defined by using an XML-based IDL. The normative schema for this language can be found in `/usr/share/lib/xml/rng/radadr.rng.1`. The namespace name is `http://xmlns.oracle.com/radadr`.

Overview

The top-level element in an ADR definition document is an `api`. The `api` element has one mandatory attribute, `name`, which is used to name the output files. The element contains one or more derived type or interface definitions. Because there is no requirement that an interface must use derived types, there is no requirement that any derived types be specified in an API document. To enable consumers to use the data typing defined by ADR for non-interface purposes, there is no requirement that an interface must be defined. However, note that either a derived type or an interface must be defined.

Three derived types are available for definition and use by interfaces: a structured type that can be defined with a `struct` element, an enumeration type that can be defined with an `enum` element, and a dictionary type that can be defined with a `dictionary` element. Interfaces are defined using `interface` elements. The derived types defined in an API document are available for use by all interfaces defined in that document.

The following is an example of an API.

EXAMPLE 46 Skeleton API document

```
<api xmlns="http://xmlns.oracle.com/radadr" name="com.oracle.solaris.rad.example"
 register="true">
```

```
        <version/>
        <struct>...</struct>
        <struct>...</struct>
        <enum>...</enum>
        <dictionary>...</dictionary>
        <interface>...</interface>
        <interface>...</interface>
</api>
```

The `xmlns` is required to indicate the type of the XML document. The `name` attribute is identifying the name of the API, the namespace within which all subsidiary interfaces are to be found. There are additional attributes to assist in the generation of server module code.

The register attribute is a `boolean`, which is optional and true by default. If true, then `radadrgen` automatically generates a `_rad_reg()` function when generating server implementation code. If false, the function is not generated and the module author will need to provide a `_rad_reg()` function. This option is primarily provided for the creation of special types of modules, such as protocol or transport modules, in general it does not need to be specified, since the default generated function is enough for most purposes.

Version

A version element is required for all APIs.

The initial version of an API must always be defined as follows:

```
<version major="1" minor="0"/>
```

This indicates that the module is starting at version 1.0.

Enumeration Definitions

The `enum` element has a single mandatory attribute, `name`. The `name` is used when referring to the enumeration from other derived type or interface definitions. An `enum` contains one or more `value` elements, one for each user-defined enumerated value. A `value` element has a mandatory `name` attribute that gives the enumerated value a symbolic name. The symbolic name is not used elsewhere in the API definition, only in the server and various client environments. The symbolic name that is exposed in these environments are environment-dependent. An environment offering an explicit interface to RAD must provide an interface that accepts the exact string values defined by the `value` elements' `name` attributes.

Some language environments support associating scalar values with enumerated type values, for example C. To provide richer support for these environments, ADR supports this concept as well. By default, an enumerated value has an associated scalar value 1 greater than the preceding enumerated value's associated scalar value. The first enumerated value is assigned a

scalar value of 0. Any enumerated value element may override this policy by defining a `value` with the desired value. A `value` attribute must not specify a scalar value already assigned, implicitly or explicitly, to an earlier value in the enumeration and `value` elements contain no other elements.

EXAMPLE 47 Enumeration Definition

```
<enum name="Colors">
<value name="RED" /> <!-- scalar value: 0 -->
<value name="ORANGE" /> <!-- scalar value: 1 -->
<value name="YELLOW" /> <!-- scalar value: 2 -->
<value name="GREEN" /> <!-- scalar value: 3 -->
<value name="BLUE" /> <!-- scalar value: 4 -->
<value name="VIOLET" value="6" /> <!-- indigo was EOLed -->
</enum>
```

Structure Definitions

Similar to the `enum` element, the `struct` element has a single mandatory attribute, `name`. The name is used when referring to the structure from other derived type or interface definitions. A `struct` contains one or more `field` elements, one for each field of the structure. A `field` element has a mandatory `name` attribute that gives the field a symbolic name. The symbolic name isn't used elsewhere in the API definition, only in the server and various client environments. In addition to a name, each field must specify a type.

You can define the type of a field in multiple ways. If a field is a plain base type, that type is defined with a `type` attribute. If a field is a derived type defined elsewhere in the API document, that type is defined with a `typeref` attribute. If a field is an array of some type (base or derived), that type is defined with a nested `list` element. The type of the array is defined in the same fashion as the type of the field: either with a `type` attribute, a `typeref` attribute, or another nested `list` element.

A field's value might be declared nullable by setting the `field` element's `nullable` attribute to `true`.

Note - The structure fields, methods return values, method arguments, attributes, error return values, and events have types, and in the IDL, use identical mechanisms for defining those types.

EXAMPLE 48 `struct` Definition

```
<struct name="Name">
        <field name="familyName" type="string" />
        <field name="givenNames">
```

```
                <list type="string" />
        </field>
</struct>

<struct name="Person">
        <field name="name" typeref="Name" />
        <field name="title" type="string" nullable="true" />
        <field name="shoeSize" type="int" />
</struct>
```

Dictionary Definitions

You can use dictionaries to add a data structure in which the key-value pair mappings can be stored and retrieved. The following example shows how to use the dictionary tag.

```
<dictionary>
  <key type="<key type>">
  <value type="<value type>">
</dictionary>
```

You can use the dictionary type similar to any other RAD type such as a field in a structure, a method argument or a return value, a property, an error payload, or as an event payload.

EXAMPLE 49 Defining a Dictionary

This example shows how to define a dictionary with a key type of integer and value type of string as a read-write property.

```
...
  <property name="DictProp" access="rw" >
     <dictionary>
        <key type="integer" />
        <value type="string" />
     </dictionary>
  </property>
...
```

Values can be of any type except for `list` and `dictionary`. The value can be a derived type or a reference in which case you must use the `"typeref"` tag instead of the `"type"` tag. However, the key must belong to any one of the following basic types:

- boolean
- integer
- unsigned integer
- long
- unsigned long

- float
- double
- time
- string
- name

Interface Definitions

An interface definition has a name, and one or more attributes, methods, or events. An interface's name is defined with the `interface` element's mandatory `name` attribute. This name is used when referring to the inherited interface from other interface definitions, as well as in the server and various client environments. The other characteristics of an interface are defined using child elements of the `interface` element.

Methods

Each method in an interface is defined by a `method` element. The name of a method is defined by this element's mandatory `name` attribute. The other properties of a method are defined by child elements of the `method`.

If a method has a return value, it is defined using a single `result` element. The type of the return value is specified in the same way the type is specified for a structure field. If no `result` element is present, the method has no return value.

If a method can fail for an API-specific reason, it is defined using a single `error` element. The type of an error is specified the same way the type is specified for a structure field. Unlike a structure field, an error need not specify a type. Such a situation is indicated by an `error` element with no attributes or child elements. If no `error` element is present, the method will only fail if there is a connectivity problem between the client and the server.

A method's arguments are defined, in order, with zero or more `argument` elements. Each `argument` element has a mandatory `name` attribute. The type of an argument is specified in the same way the type is specified for a structure field.

EXAMPLE 50 Method Definition

```
<struct name="Meal">...</struct>
<struct name="Ingredient">...</struct>

<method name="cook">
        <result typeref="Meal" />
        <error />
        <argument type="string" name="name" nullable="true" />
```

```
        <argument name="ingredients">
                <list typeref="Ingredient" />
        </argument>
</method>
```

Attributes

Each attribute in an interface is defined by a `property` element. The name of an attribute is defined by this element's mandatory `name` attribute. The types of access permitted are defined by the mandatory `access` attribute, which takes a value of `ro`, `wo`, or `rw`, corresponding to read-only access, write-only access, or read-write access, respectively.

The type of an attribute is specified in the same way the type is specified for a structure field.

If access to an attribute can fail for an API-specific reason, it is defined using one or more `error` elements. An `error` element in a `property` may specify a `for` attribute, which takes a value of `ro`, `wo`, or `rw`, corresponding to the types of access the error return definition applies to. An `error` element with no `for` attribute is equivalent to one with a `for` attribute set to the access level defined on the `property`. Two error elements may not specify overlapping access types. For example, on a read-write property it is invalid for one `error` to have no `for` attribute (implying `rw`) and one to have a `for` attribute of `wo` they both specify an error for writing.

The type of an error is specified the same way the type is specified for a method. It is identical to defining the type of a structure, with the exception that a type need not be defined.

EXAMPLE 51 Attribute Definition

```
<struct name="PrivilegeError">...</struct>

<property name="guestList" access="rw">
        <list type="string" />
        <error for="wo" typeref="PrivilegeError" />
        <!-- Reads cannot fail -->
</property>
```

Events

Each event in an interface is defined by a `event` element. The name of an event is defined by this element's mandatory `name` attribute. The type of an event is specified in the same way the type is specified for a structure field.

EXAMPLE 52 Event Definition

```
<struct name="TremorInfo">...</struct>
```

```
<event name="earthquakes" typeref="TremorInfo" />
```

API Example

EXAMPLE 53 API

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<api xmlns="http://xmlns.oracle.com/radadr" name="com.oracle.solaris.rad.example">

         <version major="1" minor="0"/>

         <struct name="StringInfo">
                <field type="integer" name="length" />
                <field name="substrings">
                        <list type="string" />
                </field>
        </struct>

        <struct name="SqrtError">
                <field type="float" name="real" />
                <field type="float" name="imaginary" />
        </struct>

        <enum name="Mood">
                <value name="IRREVERENT" />
                <value name="MAUDLIN" />
        </enum>

        <struct name="MoodStatus">
                <field typeref="Mood" name="mood" />
                <field type="boolean" name="changed" />
        </struct>

        <interface name="GrabBag" stability="private">

                <method name="sqrt">
                        <result type="integer" />
                        <error typeref="SqrtError" />
                        <argument type="integer" name="x" />
                </method>

                <method name="parseString">
                        <result typeref="StringInfo" nullable="true" />
                        <argument type="string" name="str" nullable="true" />
                </method>

                <property typeref="Mood" name="mood" access="rw">
                        <error for="wo" />
```

```
                </property>

                <event typeref="MoodStatus" name="moodswings" />
        </interface>
</api>
```

radadrgen

`radadrgen` is the ADR IDL processing tool that is used to generate API-specific language bindings for the RAD server and various client environments. See the `radadrgen`(1) man page for details about its options.

♦ ♦ ♦ CHAPTER 4

Module Development

RAD is modular in a variety of ways. Modules may deliver custom protocols, transports, or API definitions and implementations. This section focuses on the custom API definitions and implementations. Although an API can be constructed manually, using `radadrgen` to generate the necessary type definitions is much simpler.

APIs in C

This section describes the APIs that are available for C language.

Entry Points

All entry points take a pointer to the object instance and a pointer to the internal structure for the method or attribute. The object instance pointer is essential for distinguishing different objects that implement the same interface. The internal structure pointer is theoretically useful for sharing the same implementation across multiple methods or attributes, but isn't used and may be removed.

Additionally, all entry reports return a `conerr_t`. If the access is successful, they should return `CE_OK`. If the access fails due to a system error, they should return `CE_SYSTEM`. If the access fails due to an expected error which should be noted in the API definition, they should return `CE_OBJECT`. If an expected error occurs and an error payload is defined, it may be set in `*error`. The caller will unref the error object when it is done with it.

- A method entry point has the type `meth_invoke_f`:

  ```
  typedef conerr_t (meth_invoke_f)(rad_instance_t *inst, adr_method_t *meth,
      adr_data_t **result, adr_data_t **args, int count, adr_data_t **error);
  ```

  `args` is an array of count arguments.

  Upon successful return, `*result` should contain the return value of the method, if any.

  The entry point for a method named `METHOD` in interface `INTERFACE` is named `interface_INTERFACE_invoke_METHOD`.

- An attribute read entry point has the type `attr_read_f`:

  ```
  typedef conerr_t (attr_read_f)(rad_instance_t *inst, adr_attribute_t *attr,
  adr_data_t **value, adr_data_t **error);
  ```

  Upon successful return, `*value` should contain the value of the attribute, if any.

  The read entry point for an attribute named `ATTR` in interface `INTERFACE` is named `interface_INTERFACE_read_ATTR`.

- An attribute write entry point has the type `attr_write_f`:

  ```
  typedef conerr_t (attr_write_f)(rad_instance_t *inst, adr_attribute_t *attr,
      adr_data_t *newvalue, adr_data_t **error);
  ```

  `newvalue` points to the new value. If the attribute is nullable, `newvalue` can be `NULL`.

  The write entry point for an attribute named `ATTR` in interface `INTERFACE` is named `interface_INTERFACE_write_ATTR`.

`rad` explicitly checks the types of all arguments passed to methods and all values written to attributes. Stub implementations can assume that all data provided is of the correct type. Stub implementations are responsible for returning valid data. Returning invalid data results in an undefined behavior.

Global Variables

Variables	Description
`boolean_t rad_isproxy`	A flag to determine if code is executing in the main or proxy `rad` daemon. Only special system modules, which are integral to the operation of RAD, may use this variable.

`rad_container_t`
 	
`*rad_container`
	The `rad` container that contains the object instance.

Module Registration

Function	Description
`int _rad_init(void *handle);`	A module must provide a `_rad_init`. This is called by the RAD daemon when the module is loaded and is a convenient point for module initialization including registration. Return `0` to indicate that the module successfully initialized.

Function	Description
`int rad_module_register(void *handle, int version, rad_modinfo_t *modinfo);`	`rad_module_register` provides a handle, which is the handle provided to the module in the call to `_rad_init`. This handle is used by the RAD daemon to maintain the private list of loaded modules. The version indicates which version of the `rad` module interface the module is using. `modinfo` contains information used to identify the module.

Instance Management

Function	Description
`rad_instance_t *rad_instance_create(rad_object_type *type, void *data, void (*)(void *)freef);`	`rad_instance_create` uses the supplied parameters to create a new instance of an object of type. `data` is the user data to store with the instance and the freef function is a callback which will be called with the user data when the instance is removed. If the function fails, it returns `NULL`. Otherwise, a valid instance reference is returned.
`void * rad_instance_getdata(rad_instance_t *instance);`	`rad_instance_getdata` returns the user data (supplied in `rad_instance_create`) of the RAD instance.
`void rad_instance_notify (rad_instance_t *instance, const char *event, long sequence, adr_data_t *data);`	`rad_instance_notify` generates an event on the supplied `instance`. The `sequence` is supplied in the event as the sequence number and the payload of the event is provided in data.

Container Interactions

Function	Description
`conerr_t rad_cont_insert(rad_container_t *container, adr_name_t *name, rad_instance_t *instance);`	Create a RAD instance, `rad_instance_t`, using the supplied name and object and then insert into container. If the operation succeeds, `CE_OK` is returned.
`conerr_t rad_cont_insert_singleton(rad_container_t *container, adr_name_t *name, rad_object_t *object);`	
`void rad_cont_remove(rad_container_t *container, adr_name_t *name);`	Remove the `instance` from the container.
`conerr_t rad_cont_register_dynamic(rad_container_t *container, adr_name_t *name, rad_modinfo_t *modinfo, rad_dyn_list_t listf, rad_dyn_lookup_t lookupf, void *arg);`	Register a dynamic container instance manager. The container defines the container in which the instances will be managed. The `name` defines the name filter for which this instance manager is responsible. A typical name would
`conerr_t (*rad_dyn_list_t)(adr_pattern_t *pattern, adr_data_t **data, void *arg);`	

Function	Description
`conerr_t (*rad_dyn_lookup_t)(adr_name_t **name, rad_instance_t **inst, void *arg);`	define the type of the instance which are managed. For example, `zname = adr_name_vcreate (MOD_DOMAIN, 1, "type", "Zone")` would be responsible for managing all instances with a type of "Zone". `listf` is a user-supplied function which is invoked when objects with the matching pattern are listed. `lookupf` is a user-supplied function which is invoked when objects with the matching name are looked up. arg is stored and provided in the callback to the user functions.

Logging

Function	Description
`void rad_log(rad_logtype_t type, const char * format, ...);`	Log a message with type and format to the `rad` log. If the type is a lower level than the `rad` logging level, then the message is discarded.
`void rad_log_alloc()`	Log a memory allocation failure with log level RL_FATAL.
`rad_logtype_t rad_get_loglevel()`	Return the logging level.

Using Threads

Function	Description
`void *rad_thread_arg(rad_thread_t *tp);`	Return the `arg` referenced by the thread `tp`.
`void rad_thread_ack(rad_thread_t *tp, rad_moderr_t error);`	This function is intended to be used from a user function previously supplied as an argument to `rad_thread_create`. It should not be used in any other context. Acknowledge the thread referenced by `tp`. This process enables the controlling thread, from which a new thread was created using `rad_thread_create`, to make progress. The error is used to update the return value from `rad_thread_create` and is set to `RM_OK` for success.
`rad_moderr_t rad_thread_create(rad_threadfp_t fp, void *arg);`	Create a thread to run `fp`. This function will not return until the user function (`fp`) calls `rad_thread_ack. arg` is stored

Function	Description
	and passed into `fp` as a member of the `rad_thread_t` data. It can be accessed using `rad_thread_arg`.
`rad_moderr_t rad_thread_create_async( rad_thread_asyncfp_t fp, void *arg);`	Create a thread to run `fp. arg` is stored and passed into `fp`.

Synchronization

Function	Description
`void rad_mutex_init(pthread_mutex_t *mutex);`	Initialize a `mutex.abort()` on failure.
`void rad_mutex_enter(pthread_mutex_t *mutex);`	Lock a `mutex. abort()` on failure.
`void rad_mutex_exit(pthread_mutex_t *mutex);`	Unlock a `mutex. abort()` on failure.
`void rad_cond_init(pthread_cond_t *cond);`	Initialize a condition variable, `cond. abort()`, on failure.

Subprocesses

Function	Description
`exec_params_t *rad_exec_params_alloc`	Allocate a control structure for executing a subprocess.
`void rad_exec_params_free(exec_params_t *params);`	Free a subprocess control structure, params.
`void rad_exec_params_set_cwd(exec_params_t *params, const char *cwd);`	Set the current working directory, `cwd`, in a subprocess control structure, `params`.
`void rad_exec_params_set_env(exec_params_t *params, const char **envp);`	Set the environment, `envp`, in a subprocess control structure, `params`.
`void rad_exec_params_set_loglevel( exec_params_t *params, rad_logtype_t loglevel);`	Set the RAD log level, `loglevel`, in a subprocess control structure, `params`.
`int rad_exec_params_set_stdin(exec_params_t *params, int fd);`	Set the stdin file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_exec_params_set_stdout(exec_params_t *params, int fd);`	Set the stdout file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_exec_params_set_stderr(exec_params_t *params, int fd);`	Set the stderr file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_forkexec(exec_params_t *params, const char **argv, exec_result_t *result);`	Use the supplied subprocess control structure, `params`, to fork and execute (`execv`) the supplied args, `argv`. If result is not `NULL`, it is updated with the subprocess pid and file descriptor details.

Function	Description
`int rad_forkexec_wait(exec_params_t *params, const char **argv, int *status);`	Use the supplied subprocess control structure, `params`, to fork and execute (`execv`) the supplied args, `argv`. If status is not `NULL`, it is updated with the exit status of the subprocess. This function will wait for the subprocess to terminate before returning.
`int rad_wait(exec_params_t *params, exec_result_t *result, int *status);`	Use the supplied subprocess control structure, params, to wait for a previous invocation of `rad_forkexe` to complete. If result is not `NULL`, it is updated with the subprocess pid and file descriptor details. If status is not `NULL`, it is updated with the exit status of the subprocess. This function will wait for the subprocess to terminate before returning.

Utilities

Function	Description
`void *rad_zalloc(size_t size);`	Return a pointer to a zero-allocated block of size bytes.
`char *rad_strndup(char *string, size_t length);`	Create and return a duplicate of string that is of size, length bytes.
`int rad_strccmp(const char * zstring, const char * cstring, size_t length);`	Compare two strings, up to a maximum size of length bytes.
`int rad_openf(const char *format, int oflag, mode_tmode, ...);`	Open a file with access mode, oflag, and mode, mode, whose path is specified by calling `sprintf()` on format.
`FILE *rad_fopenf(const char *format, const char *mode, ...);`	Open a file with mode, whose path is specified by calling `sprintf()` on format.

Locales

Function	Description
`int rad_locale_parse(const char *locale, rad_locale_t **rad_locale);`	Update `rad_locale` with locale details based on locale. If locale is `NULL`, then attempt to retrieve a locale based on the locale of the RAD connection. Returns 0 on success.
`void rad_locale_free(rad_locale_t *rad_locale);`	Free a locale, `rad_locale`, previously obtained with `rad_locale_parse`.

Transactional Processing

There is no direct support for transactional processing within a module. If a transactional model is desirable, then it is the responsibility of the module creator to provide the required building blocks, start_transaction, commit, rollback, and other related processes.

Asynchronous Methods and Progress Reporting

Asynchronous methods and progress reporting is achieved using threads and events. The pattern is to return a token from a synchronous method invocation which spawns a thread to do work asynchronously. This worker thread is then responsible for providing notifications to interested parties events.

For example, an interface has a method which returns a Task object. The method is called `installpkg` and takes one argument, the name of the package to install.

```
Task installpkg(string pkgname);
```

The Task instance returned by the method, contains enough information to identify a task. Prior to invoking `installpkg`, the client subscribes to a task-update event. The worker thread is responsible for issuing events about the progress of the work. These events contain information about the progress of the task.

In a minimal implementation, the worker thread would issue one event to notify the client that the task was complete and what the outcome of the task was. A more complex implementation would provide multiple events documenting progress and possibly also provide an additional method that a client could invoke to interrogate the server for a progress report.

APIs in Python

This section describes the APIs that are available for Python language.

The exported Python interfaces are as follows:

- `rad.server` – RAD Server module.
- `RADInstance` – RAD instance base class. All the generated interfaces inherit from the `RADInstance` class. Thus, the interfaces inherit a set of useful behaviours. All the inherited attributes are prepended with `_rad` to both prevent name collisions and clearly indicate that these attributes are protected.
- `RADContainer` – Container base class. Represents a container into which the RAD instances are inserted.

- `RADException` – RAD exception base class. Represents an exception, which will be propagated back to the client as a `CE_OBJECT` exception. If an invocation fails, the error is declared in the ADR. See Example 54, “Using `RADException`,” on page 85.

The following functions must be provided by an implementation module:

- `rad_reg`
- `rad_init`
- `rad_fini`

`rad.server` Module

The following tables provide information about the server module functions and attributes.

TABLE 2 RAD Server Module Functions

Function	Description
`rad_log`	
`_rad_locale_get()`	
`_rad_locale_free()`	
`_rad_locale_parse()`	

TABLE 3 RAD Server Module Attributes

Attribute	Description
`rad_container`	
`rad_log_lvl`	

`RADInstance` Class

The following tables provide information about the methods and properties for the RAD instance base class, `RADInstance`.

TABLE 4 `RADInstance` Methods

Method	Description
`_rad_notify(self, event, payload)`	
`_rad_insert_singleton(cls, ctr)`	
`_rad_hold(self)`	
`_rad_release(self)`	

TABLE 5 `RADInstance` Properties

Property	Description
`_rad_name`	
`_rad_user_data`	

`RADContainer` Class

The following table provides information about the methods for `RADContainer`.

TABLE 6 `RADContainer` Methods

Method	Description
`insert(self, inst)`	
`insert_singleton(self, cls)`	
`remove(self, inst)`	
`supercede(self, inst)`	
`register(self, listf, lookupf)`	
`find_by_name(self, name)`	
`find_by_id(self, id)`	
`list(self, pat)`	

`RADException` Class

This section provides an example that shows how to use `RADException`.

EXAMPLE 54 Using `RADException`

The ADR type definition is as follows:

```
<struct name="pair" stability="private">
   <field name="first" type="integer"/>
   <field name="second" type="integer"/>
</struct>
```

The ADR method definition is as follows:

```
<method name="raiseError" stability="private">
  <doc>
  Raise an exception to test exception handling.
  </doc>
```

```
  <error typeref="pair"/>
</method>
```

In the method definition, you are specifying that an exception must be raised when the initialization of the struct `"pair"` fails.

The implementation is as follows:

```
def raiseError(self):
    raise radser.RADException(snake_iface.pair(3, 6))
```

In this example, you are raising a `RADException` and providing a payload that matches the definition in the ADR document.

The exceptions that occur as a consequence of "other" errors such as divide by zero are propagated back to the client as a `CE_SYSTEM` error representing the general RAD failure code for systemic failure.

RAD Namespaces

Objects in the RAD namespace can be managed either as a set of statically installed objects or as a dynamic set of objects that are listed or created on demand.

Static Objects

`rad_modapi.h` declares two interfaces for statically adding objects to a namespace.

`rad_cont_insert()` adds an object to the namespace. In turn, objects are created by calling `rad_instance_create()` with a pointer to the interface the object implements, a pointer to object-specific callback data and a pointer to a function to free the callback data. For example:

```
adr_name _t *uname = adr_name_vcreate("com.oracle.solaris.rad.user", 1, "type", "User");
rad_instance_t *inst = rad_instance_create(&interface_User_svr, kyle_data, NULL);
(void) rad_cont_insert(&rad_container, uname, inst);
adr_name_rele(uname);
```

`rad_cont_insert_singleton()` is a convenience routine that creates an object instance for the specified interface with the specified name and adds it to the namespace. The callback data is set to `NULL`.

```
adr_name _t *uname = adr_name_vcreate("com.oracle.solaris.rad.user", 1, "type", "User");
(void) rad_cont_insert_singleton(&rad_container, uname, &interface_User_svr);
adr_name_rele(uname);
```

Dynamic Handlers

A module can register a dynamic handler for each interface that is implemented by the module. This allows efficient searching within a module by limiting a listing to a matching subset of the instances that the module is managing. Note that you can register a single dynamic handler for a module's entire namespace. Additionally, when you register a dynamic handler, you need to specify a lookup function pointer.

The following example shows the usage of dynamic handlers in the zones module.

```
cerr = rad_cont_register_dynamic(rad_container, aname,
            &modinfo, zone_listf, zone_lookupf, NULL);
```

rad Module Linkage

Modules are registered with the RAD daemon in the `_rad_reg()`. This is automatically generated from the information contained within the IDL defining the module.

Each module is required to provide a function, `_rad_init()`, for initializing the module. This function is called before any other function in the module. Similarly, the `_rad_fini()` in the module is called by the RAD daemon just prior to unloading the module.

EXAMPLE 55 Module Initialization

```
#include <rad/rad_modapi.h>

int
_rad_init(void)
{
 adr_name _t *uname = adr_name_vcreate("com.oracle.solaris.rad.user", 1, "type",
 "User");
    conerr_t cerr = rad_cont_insert_singleton(&rad_container, uname,
 &interface_User_svr);
    adr_name_rele(uname);

 if (cerr != CE_OK)
 {
       rad_log(RL_ERROR, "failed to insert module in container");
    return(-1);
    }
 return (0);
}
```

♦ ♦ ♦ CHAPTER 5

REST APIs for RAD Clients

In addition to supporting RPC-based interfaces, RAD now supports REST (Representational State Transfer), which facilitates integration with the web-based frameworks and the cloud. You can invoke the RESTful RAD APIs to build client applications that need to access administrative RAD interfaces.

This chapter describes REST APIs for RAD clients.

Interacting With RAD by Using REST

The RESTful interface can be accessed by any HTTP client that supports UNIX domain sockets. The `rad:local-http` RAD service instance that is enabled by default facilitates communication with the HTTP clients.

Note - By design, connections over UNIX domain sockets are local only.

The following example provides a quick look at interaction with RAD by using REST.

EXAMPLE 56 Interacting With RAD by Using REST

As a prerequisite, ensure that the `curl` utility is installed on your system.

1. Authenticate.

   # **curl -X POST -c cookie.txt -b cookie.txt \ --header 'Content-Type:application/json' --data '{"username":"***username***","password":"***password***", "scheme":"pam","timeout":-1, "preserve":true}' \ localhost/api/com.oracle.solaris.rad.authentication/1.0/Session/ \ --unix-socket /system/volatile/rad/radsocket-http**

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

   ---

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

```
# curl -H 'Content-Type:application/json' -X GET \ localhost/api/
com.oracle.solaris.rad.zonemgr/1.0/Zone?_rad_detail \ --unix-socket /system/volatile/
rad/radsocket-http -b cookie.txt
```

If your system has zones, you will get a response similar to the following:

```
{
        "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"
                        }
                }
        ]
}
```

URI Specification 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:

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

```
http://host:port/api/com.oracle.solaris.rad.zonemgr/1.2/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 using 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, `/api/com.oracle.solaris.rad.zonemgr/1.0/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 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/1.0/Zone/testzone1
```

URI for a 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/1.0/Zone
```

Invoking 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 57 Listing the `anet` Properties of a Zone

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 \ localhost/api/
com.oracle.solaris.rad.zonemgr/1.0/Zone/testzone1/_rad_method/getResourceProperties \ --
data '{"filter": {"type": "anet"}}' \ --unix-socket /system/volatile/rad/radsocket-http
-b cookie.txt
```

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`) manpage.

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 58 Creating a Resource

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 \ localhost/api/
com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/create_filesystem \ --
```

```
unix-socket /system/volatile/rad/radsocket-http -b cookie.txt \ --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 59 Updating a Resource

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 \ localhost/
api/com.oracle.solaris.rad.dlmgr/1.0/Datalink/net0/_rad_method/
setPropety \ --unix-socket /system/volatile/rad/radsocket-http \ --data
'{"properties":"maxbw=300","flags":1}' -b cookie.txt
```

Sample response:

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

EXAMPLE 60 Querying a Resource

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 \ localhost/api/
com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/get_filesystems \ --unix-
socket /system/volatile/rad/radsocket-http -b cookie.txt \ --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 61 Deleting a Resource

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

Sample request:

```
# curl -H 'Content-Type:application/json' -X PUT \ localhost/api/
com.oracle.solaris.rad.usermgr/1.0/UserMgr/_rad_method/deleteUser \ --data
'{"username":"tuser4"}' --unix-socket /system/volatile/rad/radsocket-http \ -b
cookie.txt
```

Sample response:

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

REST Response

All REST responses have the following basic JSON structure:

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

HTTP Status Codes

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 does 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 Response

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)
```

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 with each request.

Instead of having to re-authenticate for every request, RAD provides a token authentication mechanism. When a client connects to RAD and successfully authenticates, RAD generates a unique token for the clients 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.

Authenticating Local Clients

This section provides an example that shows how to authenticate a local client.

```
# curl -X POST -c cookie.txt -b cookie.txt --header 'Content-Type:application/json'
\ --data '{"username":"username","password":"password", "scheme":"pam","timeout":-1,
"preserve":true}' \ localhost/api/com.oracle.solaris.rad.authentication/1.0/Session/ \
--unix-socket /system/volatile/rad/radsocket-http
```

Running this command establishes a session and generates a token. 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 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 `cookie.txt` file and a new request would result in RAD processing the new request as part of the initial session.

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

```
curl -v -X GET -c cookie.txt -b cookie.txt \
localhost/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 - A session token may be used across multiple connections, which allows an authenticated client to make multiple concurrent requests.

Authenticating Remote Clients

The default configuration for REST accepts connections only from a local system on a UNIX socket. However, you can open a public port and provide secure transport over TLS for remote RAD clients. This section provides information about the three general steps required:

1. Create a service instance to handle requests from remote clients.
2. Test the remote connection.
3. Set up a RAD connection.

▼ How to Create a Service Instance to Handle Requests from Remote Clients

1. Create a SMF service manifest by copying the `/lib/svc/manifest/system/rad.xml` manifest and modifying it.

The following example shows a sample SMF manifest after modification.

`cat rad-remote-http.xml`

```
<?xml version="1.0" ?>
<!DOCTYPE service_bundle
 SYSTEM '/usr/share/lib/xml/dtd/service_bundle.dtd.1'>

<service_bundle type="manifest" name="site/rad">
   <service version="1" type="service" name="site/rad">
       <dependency restart_on="none" type="service"
           name="multi_user_dependency" grouping="require_all">
           <service_fmri value="svc:/milestone/multi-user"/>
       </dependency>
       <exec_method name='start' type='method' exec='/usr/lib/rad/rad -sp'
 timeout_seconds='0'/>
       <exec_method name='stop' type='method' exec=':kill' timeout_seconds='0'/>
       <instance name='remote-http' enabled='false' complete='true'>
         <property_group name='ssl_port' type='xport_tls'>
           <propval name='certificate' type='astring' value='/etc/rad/cert.pem'/>
           <propval name='generate' type='boolean' value='true'/>
           <propval name='localonly' type='boolean' value='false'/>
           <propval name='pam_service' type='astring' value='rad-tls'/>
           <propval name='port' type='integer' value='12303'/>
           <propval name='privatekey' type='astring' value='/etc/rad/key.pem'/>
           <propval name='proto' type='astring' value='rad_http'/>
           <propval name='value_authorization' type='astring' value='solaris.smf.value.
rad'/>
         </property_group>
         <property_group name='config' type='application'>
           <property name='moduledir' type='astring'>
```

```
            <astring_list>
              <value_node value='/usr/lib/rad/transport'/>
              <value_node value='/usr/lib/rad/protocol'/>
              <value_node value='/usr/lib/rad/module'/>
              <value_node value='/usr/lib/rad/site-modules'/>
            </astring_list>
          </property>
        </property_group>
      </instance>
      <template>
          <common_name>
                <loctext xml:lang="C">Remote RAD HTTP</loctext>
          </common_name>
          <description>
              <loctext xml:lang="C">RAD connections over REST (HTTP)</loctext>
          </description>
      </template>
   </service>
</service_bundle>
```

Note the following items in the example manifest:

- The `port` property defines the port number on which RAD accepts remote connection, which in this example is set to 12303.
- The `proto` property defines the protocol to use. In this example, it is set to HTTP as indicated by `rad-http`.

2. **Copy the manifest to the site-wide SMF manifest location at `/lib/svc/manifest/site`.**

   # **`cp rad-remote-http.xml /lib/svc/manifest/site`**

3. **Restart the `svc:/system/manifest-import:default` service instance.**

   # **`svcadm restart manifest-import`**

4. **Enable the newly created `rad:remote-http` service instance.**

   # **`svcadm enable rad:remote-http`**

▼ How to Test the Remote Connection

- **Make the following request in a browser on a remote client.**

  `https://`*hostname*`:12303`

  The request returns a response similar to the following example:

```
{
        "status": "illegal access",
        "payload": {
```

```
                "Message": "Response content type 'text/html,application/xhtml+xml,
application/xml;q=0.9,*/*;q=0.8' requested
                yet only 'application/json' is supported by the origin server.",
                "HTTP Method": "GET",
                "URI": "/",
                "RAD Operation": null
        }
}
```

How to Set Up a RAD Connection

After you have created a service instance to handle remote requests, the next step is to authenticate and establish a RAD connection from a client. To authenticate:

1. **Save the user credentials a JSON file to authenticate.**

   For example:

```
# cat body.json
{
      "username": "testuser",
      "password": "testpassword",
      "scheme": "pam",
      "preserve": true,
      "timeout": -1
}
```

2. **Copy the SSL certificate and the key files to the client from which you want to connect to the RAD server.**

   For example, you can now use the `curl` command on the client with slightly different set of options that point to the SSL key and certificate file.

```
# curl -H "Content-type: application/json" -X POST \ --data-binary @body.json \ --cert
solaris-cert.pem --key solaris-key.pem -k \ https://testserver.com:12303/api/com.oracle.
solaris.rad.authentication/1.0/Session \ -v -c cookie.txt -b cookie.txt
```

   If the authentication is successful, output similar to the following example is displayed:

```
*   Trying 192.168.1.125...
* Failed to set TCP_KEEPALIVE on fd 4
* Connected to testserver.com (192.168.1.125) port 12303 (#0)
* successfully set certificate verify locations:
*   CAfile: none
  CApath: /etc/openssl/certs
* TLSv1.2, TLS handshake, Client hello (1):
* TLSv1.2, TLS handshake, Server hello (2):
* TLSv1.2, TLS handshake, CERT (11):
* TLSv1.2, TLS handshake, Server finished (14):
* TLSv1.2, TLS handshake, Client key exchange (16):
* TLSv1.2, TLS change cipher, Client hello (1):
```

```
* TLSv1.2, TLS handshake, Finished (20):
* TLSv1.2, TLS change cipher, Client hello (1):
* TLSv1.2, TLS handshake, Finished (20):
.
.
.
.
.
< Set-Cookie: _rad_token=613891db-9cd8-44fe-90a2-d6d8d94c2fb3;
Path=/api; Max-Age=3600; HttpOnly
< Date: Mon, 27 Jul 2015 00:59:48 GMT
<
{
        "status": "success",
        "payload": {
                "href": "/api/com.oracle.solaris.rad.authentication/1.0/Session/
_rad_reference/3328"
        }
* Connection #0 to host testserver.com left intact
}
```

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 7 APIs for Authentication – `com.oracle.solaris.rad.authentication`

Resource URI	Description	Sample Request
`POST /api/com.oracle.solaris.rad.authentication/1.0/Session/`	Authenticate a RAD session.	`curl -X POST -c cookie.txt -b cookie.txt --header 'Content-Type:application/json' --data '{"username":"testuser","password":"testpassword", "scheme":"pam","timeout":-1, "preserve":true}' localhost/api/com.oracle.solaris.rad.authentication/1.0/Session/ --unix-socket /system/volatile/rad/radsocket-http`
`GET /api/com.oracle.solaris.rad.authentication/1.0/Session/{Session-Id}?_rad_detail`	Get the details of a particular session.	`curl -X GET -c cookie.txt -b cookie.txt --header 'Content-Type:application/json' localhost/api/com.oracle.solaris.rad.authentication/1.0/Session/2048?_rad_detail --unix-socket /system/volatile/rad/radsocket-http`

TABLE 8 APIs 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 localhost/api/`

Resource URI	Description	Sample Request
		com.oracle.solaris.rad.usermgr/1.0/ UserMgr/_rad_method/getUser --data '{"username":"testuser"}' --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt
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 localhost/api/ com.oracle.solaris.rad.usermgr/1.0/ UserMgr/shells?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt
PUT /api/com.oracle.solaris.rad.usermgr/1.0/ UserMgr/_rad_method/addUser	Add a user.	curl -H 'Content-Type:application/ json' -X PUT localhost/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"}' --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt
PUT localhost/api/com.oracle.solaris.rad. usermgr/1.0/UserMgr/_rad_method/deleteUser	Delete a user.	curl -H 'Content-Type:application/ json' -X PUT localhost/api/ com.oracle.solaris.rad.usermgr/1.0/ UserMgr/_rad_method/deleteUser -- data '{"username":"tuser4"}' --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt

TABLE 9 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 localhost/api/ com.oracle.solaris.rad.smf/1.0? _rad_detail --unix-socket /system/ volatile/rad/radsocket-http -b cookie.txt
GET /api/com.oracle.solaris.rad.smf/1.0/Instance/ network%2Fhttp,apache22/state	Get the status of the apache22 service.	curl -H 'Content-Type:application/ json' -X GET localhost/api/ com.oracle.solaris.rad.smf/1.0/ Instance/network%2Fhttp,apache22/state --unix-socket /system/volatile/rad/ radsocket-http -b cookie.txt
PUT /api/com.oracle.solaris.rad.smf/1.0/Instance/ network%2Fhttp,apache22/_rad_method/enable	Enable the apache22 service.	curl -H 'Content-Type:application/ json' -X PUT localhost/api/ com.oracle.solaris.rad.smf/1.0/ Instance/network%2Fhttp,apache22/ _rad_method/enable --unix-socket /system/volatile/rad/

Resource URI	Description	Sample Request
		`radsocket-http -b cookie.txt --data '{"temporary": true}'`
`PUT /api/com.oracle.solaris.rad.smf/1.0/Instance/network%2Fhttp,apache22/_rad_method/disable`	Disable the `apache22` service.	`curl -H 'Content-Type:application/json' -X PUT localhost/api/com.oracle.solaris.rad.smf/1.0/Instance/network%2Fhttp,apache22/_rad_method/disable --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt --data '{"temporary": true}'`

TABLE 10 APIs for ZFS Management – `com.oracle.solais.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 localhost/api/com.oracle.solaris.rad.zfsmgr/1.0?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt`
`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 localhost/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/get_filesystems --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt --data '{"recursive":true}'`
`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 localhost/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/get_snapshots --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt --data '{"recursive":true}'`
`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 localhost/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsUtil/_rad_method/valid_zfs_name --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt --data '{"name":"test@test"}'`
`PUT localhost/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 localhost/api/com.oracle.solaris.rad.zfsmgr/1.0/ZfsDataset/rpool/_rad_method/create_filesystem --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt --data '{"name":"rpool/export/home/testuser/p2"}'`

TABLE 11 APIs for Zone Management – `com.oracle.solais.rad.zonemgr`

Resource URI	Description	Sample Request
`GET /api/com.oracle.solaris.rad.zonemgr/1.2?_rad_detail`	List the details of all the interfaces available for Zone management.	`curl -H 'Content-Type:application/json' -X GET localhost/api/com.oracle.solaris.rad.zonemgr/1.2?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt`

Resource URI	Description	Sample Request
GET /api/com.oracle.solaris.rad.zonemgr/1.2/Zone/{zone-name}?_rad_detail	Get the details of a zone.	curl -H 'Content-Type:application/json' -X GET localhost/api/com.oracle.solaris.rad.zonemgr/1.2/Zone/testzone1?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt
GET /api/com.oracle.solaris.rad.zonemgr/1.0/ZoneInfo?_rad_detail	Get the details of the zone for which the interface is executing.	curl H 'Content-Type:application/json' -X GET localhost/api/com.oracle.solaris.rad.zonemgr/1.0/ZoneInfo?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt

TABLE 12 APIs for Datalink Management – `com.oracle.solaris.rad.dlmgr`

Resource URI	Description	Sample Request
GET /api/com.oracle.solaris.rad.dlmgr/1.0?_rad_detail	List the details of all the interfaces available for datalink management.	curl -H 'Content-Type:application/json' -X GET localhost/api/com.oracle.solaris.rad.dlmgr/1.0?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt
PUT /api/com.oracle.solaris.rad.dlmgr/1.0/Datalink/net0/_rad_method/getProperty	Get the details of the priority property from net0.	curl -H 'Content-Type:application/json' -X PUT localhost/api/com.oracle.solaris.rad.dlmgr/1.0/Datalink/net0/_rad_method/getProperty --unix-socket /system/volatile/rad/radsocket-http --data '{"properties":"priority"}' -b cookie.txt

TABLE 13 APIs for Kernel Statistics – `com.oracle.solaris.rad.kstat`

Resource URI	Description	Sample Request
GET /api/com.oracle.solaris.rad.kstat/1.0?_rad_detail	List the details for all the interfaces available for kernel statistics.	curl -H 'Content-Type:application/json' -X GET localhost/api/com.oracle.solaris.rad.kstat/1.0?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt
GET /api/com.oracle.solaris.rad.kstat/1.0/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 localhost/api/com.oracle.solaris.rad.kstat/1.0/Kstat/misc,cpu_info0,cpu_info,0?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt
GET /api/com.oracle.solaris.rad.kstat/1.0/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 localhost/api/com.oracle.solaris.rad.kstat/1.0/Kstat/misc,vm,cpu,0?_rad_detail --unix-socket /system/volatile/rad/radsocket-http -b cookie.txt

Tips

- Man pages for each of the RAD modules are available in section 3RAD. For example, to view the man page of the `com.oracle.solaris.rad.kstat` module, type `man -s 3RAD kstat`.
- Some operations such as adding a new user requires additional privileges. You must ensure that the user has appropriate profiles to execute the operation.
- By default, RAD log messages are available in the `/var/svc/log/system-rad:local.log` file. To enable logging of debug messages, type the following commands:

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

♦ ♦ ♦ APPENDIX A

zonemgr ADR Interface Description Language Example

The example in this appendix shows some APIs used in the zonemgr ADR Interface Description Language. It does not reflect the actual full implementation of the zonemgr APIs in Oracle Solaris.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<api xmlns="http://xmlns.oracle.com/radadr" name="com.oracle.solaris.rad.zonemgr">
  <version major="1" minor="0"/>
  <enum name="ErrorCode">
    <value name="NONE" value="0"/>
    <value name="FRAMEWORK_ERROR"/>
    <value name="SNAPSHOT_ERROR"/>
    <value name="COMMAND_ERROR"/>
    <value name="RESOURCE_ALREADY_EXISTS"/>
    <value name="RESOURCE_NOT_FOUND"/>
    <value name="RESOURCE_TOO_MANY"/>
    <value name="RESOURCE_UNKNOWN"/>
    <value name="ALREADY_EDITING"/>
    <value name="PROPERTY_UNKNOWN"/>
    <value name="NOT_EDITING"/>
    <value name="SYSTEM_ERROR"/>
    <value name="INVALID_ARGUMENT"/>
    <value name="INVALID_ZONE_STATE"/>
  </enum>
  <struct name="Result" stability="private">
    <field typeref="ErrorCode" name="code" nullable="true"/>
    <field type="string" name="str" nullable="true"/>
    <field type="string" name="stdout" nullable="true"/>
    <field type="string" name="stderr" nullable="true"/>
  </struct>
  <struct name="ConfigChange">
    <field type="string" name="zone"/>
  </struct>
  <struct name="StateChange">
    <field type="string" name="zone"/>
    <field type="string" name="oldstate"/>
    <field type="string" name="newstate"/>
```

```
</struct>
<enum name="PropertyValueType">
  <value name="PROP_SIMPLE"/>
  <value name="PROP_LIST"/>
  <value name="PROP_COMPLEX"/>
</enum>
<struct name="Property">
  <field name="name" type="string"/>
  <field name="value" type="string" nullable="true"/>
  <field name="type" typeref="PropertyValueType" nullable="true"/>
  <field name="listvalue" nullable="true">
    <list type="string"/>
  </field>
  <field name="complexvalue" nullable="true">
    <list type="string"/>
  </field>
</struct>
<struct name="Resource">
  <field type="string" name="type"/>
  <field name="properties" nullable="true">
    <list typeref="Property"/>
  </field>
  <field name="parent" type="string" nullable="true"/>
</struct>
<interface name="ZoneManager">
  <method name="create">
    <result typeref="Result"/>
    <error typeref="Result"/>
    <argument name="name" type="string"/>
    <argument name="path" type="string" nullable="true"/>
    <argument name="template" type="string" nullable="true"/>
  </method>
  <method name="delete">
    <result typeref="Result"/>
    <error typeref="Result"/>
    <argument name="name" type="string"/>
  </method>
  <method name="importConfig">
    <result typeref="Result"/>
    <error typeref="Result"/>
    <argument name="noexecute" type="boolean"/>
    <argument name="name" type="string"/>
    <argument name="configuration">
      <list type="string"/>
    </argument>
  </method>
  <event typeref="StateChange" name="stateChange"/>
</interface>
<interface name="ZoneInfo">
  <property name="brand" access="ro" type="string"/>
  <property name="id" access="ro" type="integer"/>
```

```
  <property name="uuid" access="ro" type="string" nullable="true">
    <error typeref="Result"/>
  </property>
  <property name="name" access="ro" type="string"/>
  <property name="isGlobal" access="ro" type="boolean"/>
</interface>
<interface name="Zone">
  <name key="name" primary="true"/>
  <name key="id"/>
  <property name="auxstate" access="ro" nullable="true">
    <list type="string"/>
    <error typeref="Result"/>
  </property>
  <property name="brand" access="ro" type="string"/>
  <property name="id" access="ro" type="integer"/>
  <property name="uuid" access="ro" type="string" nullable="true">
    <error typeref="Result"/>
  </property>
  <property name="name" access="ro" type="string"/>
  <property name="state" access="ro" type="string"/>
  <method name="cancelConfig">
    <error typeref="Result"/>
  </method>
  <method name="exportConfig">
    <result type="string"/>
    <error typeref="Result"/>
    <argument name="includeEdits" type="boolean" nullable="true"/>
    <argument type="boolean" name="liveMode" nullable="true"/>
  </method>
  <method name="update">
    <error typeref="Result"/>
    <argument name="noexecute" type="boolean"/>
    <argument name="commands">
      <list type="string"/>
    </argument>
  </method>
  <method name="editConfig">
    <error typeref="Result"/>
    <argument type="boolean" name="liveMode" nullable="true"/>
  </method>
  <method name="commitConfig">
    <error typeref="Result"/>
  </method>
  <method name="configIsLive">
    <result type="boolean"/>
  </method>
  <method name="configIsStale">
    <result type="boolean"/>
    <error typeref="Result"/>
  </method>
  <method name="addResource">
```

```
  <error typeref="Result"/>
  <argument name="resource" typeref="Resource"/>
  <argument name="scope" typeref="Resource" nullable="true"/>
</method>
<method name="reloadConfig">
  <error typeref="Result"/>
  <argument type="boolean" name="liveMode" nullable="true"/>
</method>
<method name="removeResources">
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="false"/>
  <argument name="scope" typeref="Resource" nullable="true"/>
</method>
<method name="getResources">
  <result>
    <list typeref="Resource"/>
  </result>
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="true"/>
  <argument name="scope" typeref="Resource" nullable="true"/>
</method>
<method name="getResourceProperties">
  <result>
    <list typeref="Property"/>
  </result>
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="false"/>
  <argument name="properties" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="setResourceProperties">
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="false"/>
  <argument name="properties" nullable="false">
    <list typeref="Property"/>
  </argument>
</method>
<method name="clearResourceProperties">
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="false"/>
  <argument name="properties" nullable="false">
    <list type="string"/>
  </argument>
</method>
<method name="apply">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
```

```
</method>
<method name="attach">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="boot">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="clone">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="detach">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="halt">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="install">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="mark">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="move">
```

```
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="rename">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="ready">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="reboot">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="savecore">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="shutdown">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="suspend">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="uninstall">
  <result typeref="Result"/>
  <error typeref="Result"/>
```

```
      <argument name="options" nullable="true">
        <list type="string"/>
      </argument>
    </method>
    <method name="verify">
      <result typeref="Result"/>
      <error typeref="Result"/>
      <argument name="options" nullable="true">
        <list type="string"/>
      </argument>
    </method>
    <method name="getManager">
      <result typeref="ZoneManager"/>
    </method>
    <event typeref="ConfigChange" name="configChange"/>
  </interface>
</api>
```

Index