ODataLite

Getting Started

Microsoft Corporation
October 24, 2014

Table of Contents

1 Introduction

This guide explains how to get started with ODataLite. It explains how to:

- Build ODataLite
- Install ODataLite
- Configure ODataLite
- Start and stop the server
- Validate an installation
- Develop OData providers

ODataLite is a plugin that runs under the PHIT web server (included as part of this distribution).

1.1 What is ODataLite?

ODataLite is a software service that implements a subset of the OData 4.0 standard, described here: `http://www.odata.org/documentation/odata-version-4-0`. This stanard defines a protocol that is:

- REST-based - the protocol operations are based on HTTP methods.
- JSON-based - the protocol payload is represented using JSON.

A key goal of ODataLite is to run on resource constrained systems. Accordingly, ODataLite aims at minimal conformance as defined by the OData standard. This standard lists several requirments of a minimally conformant implementation, but the main ODataLite limitations are as follows.

- Only supports the JSON format (no support for the Atom XML format).
- Only supports a subset of query options.

This guide does not cover the OData standard. For a complete description of the OData protocol, please see `http://www.odata.org`.

1.2 What does the ODataLite service do?

The ODataLite service handles requests from clients. These requests utilize HTTP methods to perform the following operations:

- HTTP GET - Gets an entity set or invoke a function.
- HTTP POST - Creates an entity or invokes an action.
- HTTP PUT - Creates an entity.
- HTTP PATCH - Updates an entity.
- HTTP DELETE - Removes an entity.

The server delegates these requests to OData providers. Developers may build new providers to handle these requests for a particular entity type (provider development is covered later in this document).

Here is an example of a simple request-response sequence. The client sends the following request to the ODataLite server.

```
GET /odata/Processes(1) HTTP/1.1
Accept: application/json;odata.metadata=minimal
OData-Version: 4.0
```

The server delegates this request to a provider, which sends the following response.

```
HTTP/1.1 200 OK
Content-Type: application/json;odata.metadata=minimal;charset=utf-8
OData-Version: 4.0
Cache-Control: no-cache
Content-Length: 203

{
  "odata.metadata": "odata/$metadata#Processes/$entity",
  "name": "init",
  "cmdline": "/sbin/init",
  "state": "sleeping",
  "pid": 1,
  "ppid": 0,
  "vsize": 19808256,
  "rss": 136,
  "rsslim": -1
}
```

As noted above, the HTTP payload is always encoded as JSON, even for errors. Here is an example of a request that results in an error.

```
GET /odata/Processes(123456) HTTP/1.1
Accept: application/json;odata.metadata=minimal
OData-Version: 4.0
```

The provider sends back the following error response.

```
HTTP/1.1 404 Not Found
Content-Type: application/json;odata.metadata=minimal;charset=utf-8
OData-Version: 4.0
Cache-Control: no-cache
Content-Length: 128

{
  "error": {
    "code": "3",
    "message": {
      "lang": "en-us",
      "value": "not found: Process(123456)"
    }
  }
}
```

2 Building and Installing

This chapter explains how to configure and build ODataLite.

2.1 Configuring ODataLite

To configure ODataLite, type the following command:

```
# ./configure
```

ODataLite depends on the following packages, which must be installed first:

- pam
- pam-devel
- openssl (future dependency)
- openssl-devel (future dependency)
- expat-devel (XML library, used by the '`csdl2c`' program)

The configure script checks for each of these dependencies and fails if any are missing.

2.2 Building ODataLite

To build ODataLite, simply type the following command:

```
# make
```

This step builds all components including the tests.

2.3 Running the unit tests

To run the unit tests, simply type the following command:

```
# make check
```

This step runs all the unit tests.

2.4 Installing ODataLite

To install ODataLite, type the following command:

```
# make install
```

This installs ODataLite in the default location: '`/opt/odatalite`' (which can be changed with the 'configure –prefix=DIR' option). The installation locations of the various components can be controlled with configure script options.

Note: the ODataLite PAM configuration file is not installed as part of this step. Since it affects security, it must be installed manually. This requires manually copying '`phit.pam`' (supplied with the ODataLite distribtion) to the PAM configuration directory. On Linux, this is done as follows (from the root of the ODataLite distribtion).

```
# cp ./src/base/phit.pam /etc/pam.d/phit
```

2.5 Layout of the ODataLite installation

By default, ODataLite installs under '`/opt/odatalite`'. Here is a list of installed files.

```
/opt/odatalite/share/phit/background.gif
/opt/odatalite/share/phit/odata.html
/opt/odatalite/share/phit/preamble.html
/opt/odatalite/share/phit/postamble.html
/opt/odatalite/include/phit.h
/opt/odatalite/include/odata.h
/opt/odatalite/etc/phit/plugins.conf
/opt/odatalite/etc/phit/phitd.conf
/opt/odatalite/etc/phit/providers.conf
/opt/odatalite/etc/phit/roles.conf
/opt/odatalite/var/log
/opt/odatalite/var/phitauth
/opt/odatalite/var/run
/opt/odatalite/bin/phitcli
/opt/odatalite/bin/olcli
/opt/odatalite/bin/phitd
/opt/odatalite/lib/libpingplugin.so
/opt/odatalite/lib/libechoplugin.la
/opt/odatalite/lib/libpingplugin.la
/opt/odatalite/lib/libcheckprovider.so
/opt/odatalite/lib/libpingplugin.so.0.0.0
/opt/odatalite/lib/libodataplugin.a
/opt/odatalite/lib/libhelloplugin.so.0.0.0
/opt/odatalite/lib/libpingplugin.so.0
/opt/odatalite/lib/libechoplugin.so.0.0.0
/opt/odatalite/lib/libechoplugin.so
/opt/odatalite/lib/libhelloplugin.so.0
/opt/odatalite/lib/libhelloplugin.so
/opt/odatalite/lib/libcheckprovider.la
/opt/odatalite/lib/libcheckprovider.so.0.0.0
/opt/odatalite/lib/libcheckprovider.so.0
/opt/odatalite/lib/libechoplugin.so.0
/opt/odatalite/lib/libhelloplugin.la
```

This chapter discusses each of the installation directories.

2.5.1 The '`/opt/odatalite/share/phit`' directory

This directory contains files to support the PHIT server's web interface (recall that PHIT is a web server). This directory contains files needed for that interface, such as HTML, GIF, JPG files.

2.5.2 The '`/opt/odatalite/include`' directory

This directory contains C header files needed for developing PHIT plugins ('`phit.h`') and OData providers ('`odata.h`', '`odatacxx.h`', and '`odatacxxinlines.h`').

2.5.3 The ‘`/opt/odatalite/etc/phit`’ directory

This directory contains all configuration files used by the server, which includes:

- ‘`phitd.conf`’ - configuration file for the PHIT web server.
- ‘`plugins.conf`’ - plugins managed by the PHIT web server.
- ‘`providers.conf`’ - providers managed by the ODataLite plugin.
- ‘`roles.conf`’ - authentication roles and groups.

2.5.4 The ‘`/opt/odatalite/var/log`’ directory

This directory is where the log files are written by the PHIT server. These include:

- ‘`phitd.log`’ - main log file.
- ‘`phitrecv.log`’ - data received from clients (with ’phitd -k’ option).
- ‘`phitsend.log`’ - data sent to clients (with ’phitd -k’ option).

2.5.5 The ‘`/opt/odatalite/var/phitauth`’ directory

This directory is where temporary files are kept during local authentication. Clients connecting on Unix domain sockets use a local authentication scheme in which a secret is stored in a file in this directory. This file is readable only by this user. The client must read the secret from this file and present it to the server to prove its identify.

2.5.6 The ‘`/opt/odatalite/var/run`’ directory

This PHIT server (‘`phitd`’) runs in this directory. This directory also contains the PID file (‘`phitd.pid`’). The ‘`phitd`’ process uses the PID file in three ways.

- To store its PID when it is running.
- To determine whether it is already running during startup.
- To stop the current instance of the process (‘`phid -s`’).

2.5.7 The ‘`/opt/odatalite/bin`’ directory

This directory contains the following programs.

- ‘`phitd`’ - The PHIT web server daemon.
- ‘`phitcli`’ - A command-line client utility for sending HTTP requests to the server.
- ‘`olcli`’ - A command-line client utility for sending OData requests to the server.
- ‘`csdl2c`’ - A command-line utility for converting CSDL XML files to C program files.

2.5.8 The ‘`/opt/odatalite/lib`’ directory

This directory contains PHIT web server plugins and OData providers.

3 Using the server

This chapter explains how to use the PHIT server. In particular it explains how to:

- Print the help message
- Examine file locations
- Start the server
- Stop the server
- Change the logging level
- Use a web browser to talk to the server
- Send requests to the server

All the examples below assume that the server is installed in the default location ('/opt/odatalite').

3.1 Printing the help message

The default installation location for the PHIT server is '/opt/odatalite/phitd'. To print the help message for this executable, type the following:

```
# /opt/odatalite/phitd -h
```

This prints the following message.

```
Usage: /opt/odatalite/bin/phitd [OPTIONS]

OPTIONS:
    -h                  Print help message
    -f                  Run server in foreground (rather than daemonizing)
    -p PORT             Listen on this port
    -u PATH             Listen on this Unix-domain socket
    -i                  Ignore authentication
    -P                  Print list of path locations
    -s                  Stop the server
    -g USER             Guest user (repeatable)
    -o USER             Operator user (repeatable)
    -a USER             Admin user (repeatable)
    -l LEVEL            Log level: FATAL|ERROR|WARNING|INFO|DEBUG|VERBOSE
    -S                  Send log output to standard output
    -k                  Log soc[k]et I/O (phitsend.log and phitrecv.log)
    -D                  Dump incoming and outgoing messages to stdout
```

3.2 Examing file locations

To print out the locations where the PHIT server expects to find its various files, type the following:

```
# /opt/odatalite/phitd -P
```

For example, if ODataLite is installed in the default location, this prints the following.

```
SOCKFILE{/opt/odatalite/var/run/phitd.sock}
PIDFILE{/opt/odatalite/var/run/phitd.pid}
AUTHDIR{/opt/odatalite/var/phitauth/}
LOGFILE{/opt/odatalite/var/log/phit.log}
RECVLOG{/opt/odatalite/var/log/phitrecv.log}
SENDLOG{/opt/odatalite/var/log/phitsend.log}
PHITD_CONF{/opt/odatalite/etc/phit/phitd.conf}
PLUGINS_CONF{/opt/odatalite/etc/phit/plugins.conf}
ROLES_CONF{/opt/odatalite/etc/phit/roles.conf}
PROVIDERS_CONF{/opt/odatalite/etc/phit/providers.conf}
DATADIR{/opt/odatalite/share/phit}
LIBDIR{/opt/odatalite/lib}
DATADIR{/opt/odatalite/share/phit}
PLUGIN_LIBDIR{/opt/odatalite/lib}
```

From this we can see that the server's main log file can be found here:

```
/opt/odatalite/var/log/phitrecv.log
```

3.3 Starting the server

To start the server in the background, simply type:

```
# /opt/odatalite/bin/phitd
```

To start the server in the foreground, type this:

```
# /opt/odatalite/bin/phitd -f
```

Starting the server in the foreground allows one to see anything printed to standard output by the server, plugins, or providers. This aids debugging.

3.4 Stopping the server

To stop the server, simply type:

```
# /opt/odatalite/bin/phitd -s
```

This stops the server by sending the kill signal to the process id found in the PID file. (‘/opt/odatalite/var/run/phitd.pid’)

3.5 Changing the logging level

The PHIT server supports the following six logging levels (listed from highest priority to lowest).

- ‘FATAL’
- ‘ERROR’
- ‘WARNING’
- ‘INFO’
- ‘DEBUG’
- ‘VERBOSE’

The default logging level is WARNING, meaning that only WARNING log messages and higher are logged. The log level can be set with a command-line option. For example:

```
# /opt/odatalite/bin/phitd -l VERBOSE
```

The log level can also be set by adding a line to the main configuration file (‘`/opt/odatalite/etc/phit/phitd.conf`’). For example:

```
loglevel=VERBOSE
```

3.6 Using a web browser to talk to the server

By default, the PHIT server listens on port 8888. The port can be set with a command-line option. For example:

```
# /opt/odatalite/bin/phitd -p 12345
```

The port can also be set by adding a line to the main configuration file (‘`/opt/odatalite/etc/phit/phitd.conf`’). For example:

```
port=12345
```

The PHIT server provides an HTML web interface for use by web browsers. This interface exposes information about the PHIT server and its plugins. Each plugin may define its own web interface. The ODataLite plugin defines a web interface that lists the registered OData providers and a JSON interface for sending OData requests to those providers.

To use this web interface, point the browser at the root page. For example,

```
http://HOSTNAME:PORT/
```

3.7 Sedning requests to the server

As mentioned above, the ODataLite server provides a web interface, allowing web browsers to interact with the server. In addition, ODataLite provides two command-line tools for sending requests to the PHIT server. These are discussed below.

3.7.1 The ‘`phitcli`’ tool

The ‘`phitcli`’ tool sends any HTTP message (stored in a file) to the server. For example, consider the following file (‘`hello.req`’).

```
M-POST /echo HTTP/1.1
Content-Type: text/plain; charset=utf-8
Content-Length: 13
TE: trailers, chunked

Hello World!
```

The following command sends this request to the server:

```
# phitcli hello.req
```

The ‘`phitcli`’ prints the response to standard output:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 13

Hello World!
```

The request is delegated to the echo plugin, which simply echos back the request payload.

3.7.2 The '`olcli`' tool

The '`olcli`' tool sends OData requests to the PHIT server. The following example sends a get request, which retrieves one instance of the '`Gadgets`' entity.

```
# olcli get '/odata/Gadgets(1)'
HTTP/1.1 200 OK
Content-Type: application/json;odata.metadata=minimal;charset=utf-8
OData-Version: 4.0
Access-Control-Allow-Origin: *
Cache-Control: no-cache
Content-Length: 196

{
  "@odata.context": "odata/$metadata#Gadgets/$entity",
  "Id": 1,
  "Color": "Red",
  "Model": "M3403",
  "WeigthInGrams": 765,
  "ManufacturingDate": "2014-01-13",
  "SerialNumber": 847574657
}
```

4 Server Features

This chapter discusses several PHIT server features.

4.1 HTTP Basic Authentication

The PHIT server supports the HTTP basic authentiation scheme, which accepts an HTTP header like the one below, in which the password is base-64 encoded.

```
Authorization: Basic bWlrYnJhczpwYXNzd29yZA==
```

The server authenticates this password with PAM (Pluggable Authentication Module), which is described in the next section.

4.2 PAM (Pluggable Authentication Module)

Passwords obtained from HTTP clients are passed to the PAM layer for authentication. This feature must be configured by installing a PAM configuration file. On most Linux system, this file is installed in the following location:

```
# /etc/pam.d/phit
```

The source distribution contains a sample of this file, located here in this source tree:

```
# odatalite-0.0.3/src/base/phit.pam
```

Here are its contents:

```
# Sample PAM authentication file (copy to /etc/pam.d)
auth       required     pam_sepermit.so
auth       include      password-auth
account    required     pam_nologin.so
account    include      password-auth
password   include      password-auth
session    include      password-auth
```

4.3 Authorization Roles

ODataLite provides a simple role-based authorization mechanism. Clients authenticate as system users. These users can be assigned to one of three roles:

- Administrator
- Operator
- Guest

These assignments are made either in the 'phit/roles.conf' file or as 'phitd' command-line options. Both styles simply assign users to one of the three roles defined above. Consider the following configuration file:

```
# <USERNAME>:<ROLENAME>
root:admin
jsmith:operator
*:guest
```

This example makes three role assignements:

- The 'root' user is assigned to the 'admin' role.

- The ‘jsmith’ user is assigned to the ‘operator’ role.
- All other users are assigned to the ‘guest’ role.

The following ‘phitd’ command-line options make the same assignments:

```
# phitd -a root -o jsmith -g '*'
```

The server performs assignments from users to roles. The server does not perform the actual authorization, that is it does not prevent users from performing any operations. Restricting operations is left to plugins and providers. Both interfaces provide methods for obtaining the role of the requesting user.

5 Developing plugins

This guide does not explain how to write plugins. To learn more about writing plugins, see the examples under the following directory in the distribution:

```
odatalite-0.0.3/src/plugins
```

6 Developing OData providers

This chapter gives a very quick overview of the OData provider development process. ODataLite does not include any production providers, but it includes several samples under this directory in the distribution:

```
odatalite-0.0.3/src/odata/providers
```

This chapter focuses on writing a provider for the '`Widgets`' entity type. The distribution contains a sample implementation in this directory:

```
odatalite-0.0.3/src/odata/providers/widget
```

The following is a sample skeleton which can be used as a starting point for any provider. It defines the main entry point and the methods needed by all providers.

```
#include <odata/odata.h>

typedef struct _Provider /* Extends OL_Provider */
{
    OL_Provider base;
}
Provider;

static void _Load(
    OL_Provider* self,
    OL_Scope* scope)
{
}

static void _Unload(
    OL_Provider* self,
    OL_Scope* scope)
{
    free(self);
}

static void _Get(
    OL_Provider* self_,
    OL_Scope* scope,
    const OL_URI* uri)
{
    OL_Scope_SendResult(scope, OL_Result_NotSupported);
}

static void _Post(
    OL_Provider* self,
    OL_Scope* scope,
    const OL_URI* uri,
    const OL_Object* object)
{
    OL_Scope_SendResult(scope, OL_Result_NotSupported);
}

static void _Put(
    OL_Provider* self,
    OL_Scope* scope,
    const OL_URI* uri,
    const OL_Object* object)
{
    OL_Scope_SendResult(scope, OL_Result_NotSupported);
}

static void _Patch(
    OL_Provider* self,
    OL_Scope* scope,
    const OL_URI* uri,
    const OL_Object* object)
{
    OL_Scope_SendResult(scope, OL_Result_NotSupported);
}

static void _Delete(
    OL_Provider* self,
    OL_Scope* scope,
    const OL_URI* uri)
{
    OL_Scope_SendResult(scope, OL_Result_NotSupported);
}

static OL_ProviderFT _ft =
{
    _Load,
    _Unload,
    _Get,
    _Post,
    _Put,
    _Patch,
    _Delete
};

OL_EXPORT OL_Provider* OL_ProviderEntryPoint()
{
    return NULL;
}
```

6.1 Implementing the entry point

Starting with this skeleton, the first step is to implement the main entry point. A typical implementation allocates and returns a structure that points to the function table of provider methods. For example:

```
OL_EXPORT OL_Provider* OL_ProviderEntryPoint()
{
    Provider* self;

    /* Allocate the Provider structure */
    if (!(self = (Provider*)calloc(1, sizeof(Provider))))
        return NULL;

    /* Set the function table */
    self->base.ft = &_ft;

    return &self->base;
}
```

The '`Provider`' structure is defined locally and extends the '`OL_Provider`' structure. Here is its definition:

```
typedef struct _Provider /* Extends OL_Provider */
{
    OL_Provider base;
    /* Define provider-specific fields here */
}
Provider;
```

The first field is a structure of type '`OL_Provider`'. This allows the '`Provider`' structure to be casted to and treated as a structure of type '`OL_Provider`' by the server.

Provider developers may define provider-specific fields. This structure is cast to '`OL_Provider`' and passed as the first parameter to all the provider methods. The provider developer may cast this parameter to '`Provider`' in order to access the extended fields.

The '`Provider.base.ft`' field must point to a function table of the provider's methods. This function table is defined locally, just above the '`OL_ProviderEntryPoint`' function, as follows.

```
static OL_ProviderFT _ft =
{
    _Load,
    _Unload,
    _Get,
    _Post,
    _Put,
    _Patch,
    _Delete,
};
```

The operations performed by the functions are defined as follows.

- '`Load`' - Called when the provider is loaded.
- '`Unload`' - Called when the provider is unloaded.
- '`Get`' - Gets an entity set or a single entity or invokes a function.
- '`Post`' - Creates an entiy or invokes an action.
- '`Put`' - Updates an entiy.
- '`Patch`' - Updates an entity.
- '`Delete`' - Deletes an entiy.

6.2 The '`Unload`' method

The '`Unload`' method is responsible for releasing the structure allocated by the '`OL_ProviderEntryPoint`' function. This example defines the '`Unload`' function as follows.

```
static void _Unload(
    OL_Provider* self,
    OL_Scope* scope)
{
    free(self);
}
```

The '`Unload`' method should also release any other resources allocated by the provider.

6.3 Implementing the '`Get`' method

As mentioned above, the '`Get`' method can do one of three things:

- Get an entity set
- Get a single entity
- Invoke a function

The example shows how to do the first two. The signature of the '`Get`' method is defined as follows.

```
void Get(
    OL_Provider* self,
    OL_Scope* scope,
    const OL_URI* uri);
```

The '`self`' parameter refers to the '`Provider`' structure allocated by the '`OL_ProviderEntryPoint`' entry point function.

The '`scope`' parameter acts as the invocation context for the operation. This object is used to allocate objects of various kinds and to post results back to the server. We will see an example of this below.

The '`uri`' parameter is a structural representation of the URI passed in the original request. In the following example, it represents the URI.

```
GET odata/Widgets(1001)
```

Methods are defined for accessing various parts of the URI. For example, the following obtains the name of the '`Widgets`' segment (where 0 is the index of the '`Widgets`' segment).

```
if (!(name = OL_URI_GetName(uri, 0)))
{
    OL_Scope_SendResult(scope, OL_Result_Failed);
    return;
}
```

The number of segments can be determined by calling the 'OL_URI_Count' method. Note that the 'odata' segment is discarded, so 'OL_URI_Count' return one.

The example above has one key ('1001'). The 'OL_URI_KeyCount' method determines how many keys a given segment has. If it has any keys, then the providersshould return a single instance. Otherwise, the provider should return an entity set. The following shows the code required to return an entity set.

```
if (OL_URI_KeyCount(uri, 0) == 0)
{
    OL_Object* obj;

    OL_Scope_SendBeginEntitySet(scope);

    /* Send Widgets(1001) */
    {
        if (!(obj = MakeWidget(scope, 1001, "Red", "A1", 55)))
        {
            OL_Scope_SendResult(scope, OL_Result_OutOfMemory);
            return;
        }

        OL_Scope_SendEntity(scope, obj);
        OL_Object_Release(obj);
    }

    /* Send Widgets(1002) */
    {
        if (!(obj = MakeWidget(scope, 1002, "Green", "A2", 65)))
        {
            OL_Scope_SendResult(scope, OL_Result_OutOfMemory);
            return;
        }

        OL_Scope_SendEntity(scope, obj);
        OL_Object_Release(obj);
    }

    OL_Scope_SendEndEntitySet(scope);
    OL_Scope_SendResult(scope, OL_Result_Ok);
    return;
}
```

This provider sends two instances. The ‘`MakeWidget`’ method uses the ‘`OL_Object`’ interface to make an instances of ‘`Widget`’. It is defined as follows.

```
OL_Object* MakeWidget(
    OL_Scope* scope,
    long long id,
    const char* color,
    const char* model,
    unsigned int weight)
{
    OL_Object* obj;

    if (!(obj = OL_Scope_NewObject(scope)))
        return NULL;

    OL_Object_AddInt64(obj, "Id", id);
    OL_Object_AddString(obj, "Color", color);
    OL_Object_AddString(obj, "Model", model);
    OL_Object_AddInt32(obj, "Weigth", weight);

    return obj;
}
```

Objects created by ‘`OL_Scope_NewObject`’ should eventually be released by the ‘`OL_Scope_Release`’ method.

If the URI has any keys, then the provider should return a single entity. Here is the code that handles that case.

```
if (OL_URI_KeyCount(uri, 0) == 1)
{
    OL_Int64 id;
    OL_Object* obj;

    /* Get the key from the URI (e.g., 'odata/Widgets(1001)') */
    if (OL_URI_GetKeyInt64(uri, 0, "$0", &id) != OL_Result_Ok)
    {
        OL_Scope_SendResult(scope, OL_Result_NotSupported);
        return;
    }

    /* Send Widgets(1001) */
    if (id == 1001)
    {
        if (!(obj = MakeWidget(scope, 1001, "Red", "A1", 55)))
        {
            OL_Scope_SendResult(scope, OL_Result_OutOfMemory);
            return;
        }
```

```
            OL_Scope_SendEntity(scope, obj);
            OL_Object_Release(obj);
            OL_Scope_SendResult(scope, OL_Result_Ok);
            return;
        }

        /* Send Widgets(1002) */
        if (id == 1002)
        {
            if (!(obj = MakeWidget(scope, 1002, "Green", "A2", 65)))
            {
                OL_Scope_SendResult(scope, OL_Result_OutOfMemory);
                return;
            }

            OL_Scope_SendEntity(scope, obj);
            OL_Object_Release(obj);
            OL_Scope_SendResult(scope, OL_Result_Ok);
            return;
        }

        OL_Scope_SendResult(scope, OL_Result_NotFound);
        return;
    }
```

6.4 Registering the ‘Widgets’ provider

To register the provider, add the following line to the ‘providers.conf’ file:

```
widgetprovider:/Widgets:
```

Where the first field (‘widgetprovider’) is the name root name of the library (the fullname might be something like ‘libwidgetprovider.so’. The second field is a comma-separated lists of segments that the provider handles (‘/Widgets’ in this case).

6.5 Testing the provider

To test the provider, restart the PHIT server and use the ‘olcli’ command-line program to send a get request to the server as follows.

```
# olcli get 'odata/Widgets'
HTTP/1.1 200 OK
Content-Type: application/json;odata.metadata=minimal;charset=utf-8
OData-Version: 4.0
Access-Control-Allow-Origin: *
Cache-Control: no-cache
Content-Length: 255

{
  "@odata.context": "odata/$metadata#Widgets",
  "value": [
    {
      "Id": 1001,
      "Color": "Red",
      "Model": "A1",
      "Weigth": 55
    },
    {
      "Id": 1002,
      "Color": "Green",
      "Model": "A2",
      "Weigth": 65
    }
  ]
}
```

This request obtains an entity set. To request a single entity, try this:

```
# olcli get 'odata/Widgets(1001)'
HTTP/1.1 200 OK
Content-Type: application/json;odata.metadata=minimal;charset=utf-8
OData-Version: 4.0
Access-Control-Allow-Origin: *
Cache-Control: no-cache
Content-Length: 122

{
  "@odata.context": "odata/$metadata#Widgets/$entity",
  "Id": 1001,
  "Color": "Red",
  "Model": "A1",
  "Weigth": 55
}
```

Also, recall that web browsers can be used to send OData requests. Simply browse to the following URL.

```
http://HOSTNAME[:PORT]/odata/webpage
```

6.6 Providing Metadata

Providers may provide metadata through the ‘`Get`’ method. These providers may dynamically generate metadata or they may use the ‘`csdl2c`’ to convert CSDL XML into static C definitions. To see an example of this, look at the following example:

```
odatalite-0.0.3/src/odata/providers/metadata
```

7 Development tips

This chapter discusses some development tips that make development a little easier.

7.1 Running PHIT from the source directory

To run PHIT form the source directory, set the 'PHIT_PREFIX' environment variable to refer to the root of the source distribution. For example:

```
export PHIT_PREFIX=/home/jdoe/odatalite
```

This makes the PHIT server look for all its configuration files and other ocmponetns under the source directory. This allows the developer to maintain the server, the plugins, and the providers without having to repeatedly install. When running out of the source directory, the server listens on port '12345'.

Note: be sure this environment variable is not defined when attempting to run the server from the installation directory (else it looks for its configuration in the source directory).

7.2 Running PHIT in the foreground

To run the PHIT server in the foreground (without daemonization), use this option:

```
# ./phitd -f
```

This allows the developer to see what is written to standard output by the server and by the provider.

7.3 Disabling authentication

Constantly authenticating is somewhat of a nuisance during development, so we suggest disabling it with the following option during development:

```
# ./phitd -i
```

7.4 Increasing the logging level

To increase the looging level to 'VERBOSE', use the following option:

```
# ./phitd -l VERBOSE
```

This allows capturing of all log messages.

7.5 Logging to standard output

To log to standard output, use the following option:

```
# ./phitd -S
```

7.6 Logging socket I/O

To log all socket I/O to log files, use the '-k' option. Input is written to 'phitsend.log' and output is written to 'phitrecv.log'. For example:

```
# ./phitd -k
```

7.7 Dump requests and responses to standard output

Use the '`-D`' option to dump HTTP requests and responses to standard output. For example:

```
# ./phitd -D
```