Testing Providers with PyWBEM

Tim Potter <tpot@hp.com>
Hewlett-Packard

Overview

- PyWBEM basics
- Exploring with wbemcli.py
- Unit testing with unittest module
- Async client programming with Twisted
- Using the MOF compiler

- Presentation materials available at
  `http://samba.org/~tpot/mdc2008`

PyWBEM overview

- A pure-Python library for WBEM
  - Sync and async CIM-XML client
  - Provider interface
  - MOF compiler
  - Command line client
- Uses Python language features to provide easy to use interfaces for WBEM

Getting started

- Install PyWBEM
  - Available in some distros
  - `http://pywbem.sourceforge.net`
- Create connection
  - `WBEMConnection()` for CIM-XML over http/https
  - `{Pegasus,OpenWBEM,SFCB}UDSConnection()` for unix domain socket access
- Use connection object to perform operations and return other PyWBEM objects

`pywbem.CIMInstanceName`

- Represents an object path/model path
- Attributes
  - `host`
  - `classname`
  - `namespace`
  - `keybindings`
- Use Python object attributes and dictionary interface

`pywbem.CIMInstance`

- Represents an instance of a class
- Attributes
  - `host`
  - `classname`
  - `namespace`
  - `path` (is a `CIMInstanceName`)
  - `properties`
- Use Python object attributes and dictionary interface

Error handling

- Python exception is thrown when a CIM error occurs

```
try:
    cli.EnumerateInstances('CIM_Foo')
except pywbem.CIMError, arg:
    error = arg[0]  # error number
    description = arg[1] # string
```

- Error constants available, e.g `CIM_ERR_FAILED`

Other useful objects

- `CIMClass`
  - `classname`
  - `properties`, `methods`, `qualifiers`
- `CIMDateTime`
  - Wrapper for `datetime.{datetime,timedelta}`
- CIM numeric types
  - `SintXX`, `UintXX`, `RealXX`
- `CIMQualifier`
  - `name`, `value`, `type`

Demo #1 and #2

- Making connections
- Performing simple WBEM operations
- Accessing CIM object attributes

Testing Instance Providers

- `EnumerateInstanceNames(`*`ClassName`*`)`
  - List of `CIMInstanceName`
- `EnumerateInstances(`*`ClassName`*`)`
  - List of `CIMInstance`
- `GetInstance(`*`InstanceName`*`)`
  - List of `CIMInstance`
- `ModifyInstance(`*`Instance`*`)`
  - `None`
- `DeleteInstance(`*`InstanceName`*`)`
  - `None`
- `CreateInstance(`*`Instance`*`)`
  - `CIMInstanceName`

Testing Association Providers

- `AssociatorNames(`*`InstanceName, Args...`*`)`
  - List of `CIMInstanceName`
- `Associators(`*`InstanceName, Args...`*`)`
  - List of `CIMInstance`
- `ReferenceNames(`*`InstanceName, Args...`*`)`
  - List of `CIMInstanceName`
- `References(`*`InstanceName, Args...`*`)`
  - List of `CIMInstance`

- Arguments are ResultClass, Role, AssociationClass, etc…

Testing Method Providers

- `InvokeMethod(`*`MethodName, InstanceName, InParams`*`)`
  - Tuple of method result and output parameters

```
Result, OutParams = cli.InvokeMethod(
    'SetPowerState', cs,
    PowerState = pywbem.Uint32(8),
    Time = datetime.now())
```

wbemcli.py

- Command line tool for exploration and ad-hoc testing
- Connects to a WBEM server then drops into Python interactive interpreter
- Lots of extra goodies to make life easier for testing and debugging

wbemcli.py (cont)

- Usage: `wbemcli.py HOST [-u USER] [-p PASS] \`
  `[-n NAMESPACE] [--no-ssl] [--port PORT]`

- Features
  - Uses full power of Python interactive interpreter
  - Saves command line history to disk
  - Aliases for common WBEM operations
  - Pretty print of long results

Demo #3

- Using `wbemcli.py` for ad-hoc testing

Unit testing with `unittest.py`

- Built-in unit testing module using xUnit interface
  - `setUp()`, `runTest()`, `tearDown()`
  - Test fixtures created for each test case
- Python version clunky but still usable
- Can run tests individually
  - By named test case
  - By named test method

Demo #4

- Running unit tests based on Python's unittest.py module

Asynchronous client programming

- Uses Twisted Python networking framework
- Event driven programming model – no threads
  - "reactor" is central object in a Twisted program
- Uses callback model to respond to events
  - "defered" is central object for using callbacks
- Go read tutorial and reference documentation at `http://twistedmatrix.com`

Using the PyWBEM Twisted Client

- Basic process for performing a client operation:
  1. Create a "factory" which creates instances of the operation you want to perform
  2. Add callbacks
  3. Call `reactor.connectTCP()`
  4. Enter or return to event loop
- Return "deferred" objects to hang callbacks off
  - *"A deferred is a promise that a function will at some point have a result".*

Example: Create CIM_Indication filter

- CIM operations, twisted style
  - Create CIMInstance object
  - Create a “CreateInstance factory”
  - Add success and failure callbacks
  - Make TCP client connection
  - Enter event loop

- Trigger subsequent operations off callbacks

Creating a CIM Listener

- Basic process for listening for indications
  - Create a `twisted.web.server` listening on port 5988 and port 5989
  - Create a `twisted.web.resource` to handle POST requests and parse received XML
  - Call `reactor.listenTCP()` function or `reactor.listenSSL()`
  - See `irecv.py` file in PyWBEM distribution
- Can have CIM client and server in same process

invent

Using PyWBEM MOF Compiler

- Define a class with a PyWBEM server interface
  - `CreateClass`, `ModifyClass`, `EnumerateQualifiers`, etc
- Create a `mof_compiler.MOFCompiler` instance
- Call `compile_file()` for each MOF file to process

Tricks & Traps

- Use `DeepInheritance = True` when enumerating classes and class names

- Use `LocalOnly = False` when calling `GetClass()` method

- Watch out for host attribute in return values from associators
  - May need to set to `None`