Reference Manual
Cloud Blueprints and Blueprint Processor

Jan. 30, 2013

Contents

Introduction

This document is a reference manual for the blueprint processing features of Oracle Enterprise Manager Cloud. It presents concepts, describes how to install and run the blueprint processor, and documents the blueprint language. Also included are sections to help in your use of the blueprint processor, such as use of the blueprint debugger, what to do when errors are diagnosed, and frequently asked questions.

Before reading this document, you should read “Introduction to Blueprints” [Intro], which is a gentler “by example” introduction.

Installing the Blueprint Processor

Installing the blueprint processor is done in 3 steps:

- Install Python 2.7, if not already present.[1]
- “Install” the blueprint processor files.

Detailed instructions are presented below for Linux and Windows.

Linux

(These instructions were tested on Oracle Linux.)

Install Python

- Download a version of Python, version 2.7 or higher (but not 3.x). For instance, use http://www.python.org/ftp/python/2.7.3/Python-2.7.3.tgz.
- Untar it, e.g. by entering:

```
tar xzf Python-2.7.3.tgz
cd Python-2.7.3
```

- Execute these commands:

```
./configure --prefix=$HOME
make
make install
```

Note: during the configure step, you may see output that includes warnings about some modules that could not be created on your platform, but those modules may not be needed; their absence may not adversely affect use of the blueprint processor.

```
Python build finished, but the necessary bits to build these modules were not found:
bsddb185           dl                 imageop
sunaudiodev
To find the necessary bits, look in setup.py in detect_modules() for the module's name.


Failed to build these modules:
_sqlite3
```

- The python interpreter should be installed in your home bin directory. Test it.

```
$HOME/bin/python2.7
```

- You should see the Python banner… something like this:

```
Python 2.7.3 (default, May 25 2012, 11:33:27)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-50)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Install Blueprint Processor From Zip File

- Unzip the blueprint distribution file to a directory, e.g. named bp_installation
- To run the blueprint processor you will…

[1] Python 2.6 may also work, but this has not been tested.

- cd to the directory, e.g. bp_installation
- Enter this command:
  - python bp_processor.py <filename> <options>

Test the Installation

- From the directory in which you placed the blueprint processor and other files…
- Enter this command to confirm you can run the blueprint processor, in this case just to see 'help' text:

```
$HOME/bin/python2.7 bp_processor.py helloWorld.yml -h
```

- You should see the 'help' text, i.e. something like this:

```
Usage:
  bp_processor.py [options] BlueprintFileName

Example:
  bp_processor.py myfile.yml -i "name:Joe" -i "count:2" -u jabauer -c https://...

Options:
  -h, --help            show this help message and exit
  -c CLOUD_URI, --cloud_uri=CLOUD_URI
                        Ex: https://myhost:4473/em/cloud
  -u USER, --user=USER  user id
  -p PASSWORD, --password=PASSWORD
                        password. If not provided, you'll be prompted.
  -t TIMEOUT, --timeout=TIMEOUT
                        timeout (seconds)
  -i INPUT_VALUE, --input_value=INPUT_VALUE
                        <param name>:<param value> e.g. -i "name:Joe Blogs"
  -r REFRESH_FREQUENCY, --refresh_frequency=REFRESH_FREQUENCY
                        # of seconds between dots in timeline. Zero: no dots
  -n INSTANCE_NAME, --instance_name=INSTANCE_NAME
                        name of blueprint instance to create
  -d, --debug           more logging, including http traffic
  -E, --pause_error     Drop into debugger if error occurs
  -I, --pause_input     Pause before processing Input section
  -R, --pause_resource  Pause before processing Resource section
  -O, --pause_output    Pause before processing Output section
  -T, --pause_terminate
                        Pause before termination, but after output processing
                        or error
  -g GRAPHIC_RESULTS, --graphic_results=GRAPHIC_RESULTS
                        Directory for deployed blueprint graphical report
  -G GRAPHIC_BLUEPRINT, --graphic_blueprint=GRAPHIC_BLUEPRINT
                        Directory for undeployed blueprint graphical report
```

- Enter this command to simulate running a blueprint without having to connect to a cloud environment:

```
$HOME/bin/python2.7 bp_processor.py helloWorld.yml
```

  - You should see something like this …

```
[jabauer@zzzzzzzz blueprints]$ $HOME/bin/python2.7 bp_processor.py helloWorld.yml

Blueprint Processor - Invocation Summary
----------------------------------------
   Cloud URI:          sim
   User:               None
```

```
   Blueprint file:     helloWorld.yml
   Timeout:            90 minutes, 0 seconds
   Ellipses frequency: 15 seconds
   Inputs:
   Pause points:       (none)
   Debug logging:      False
   Instance name:      default_instance_name

17:10:31 WARNING: No Resources specified in blueprint.  Nothing will be created

17:10:31 INFO: Output Processing
17:10:31 INFO: -----------------
17:10:31 INFO:
17:10:31 INFO: Output values specified: 1
17:10:31 INFO:    Value of MyMsg: Hello World
17:10:31 INFO:
17:10:31 INFO: Blueprint Processing Summary
17:10:31 INFO: ---------------------------
17:10:31 INFO:
17:10:31 INFO: Timing Summary (seconds):
17:10:31 INFO:    Client-side CPU time: 0.218
17:10:31 INFO:    Elapsed time:
17:10:31 INFO:       Processing time:    0.0
17:10:31 INFO:       Paused time:        0.0
17:10:31 INFO:       Total elapsed time: 0.0
```

Windows

Install Python

- Go to http://www.python.org/download/releases/2.7.3. You'll see that there are several options for installing Python.
- Download the MSI Installer. Run it. Accept the defaults suggested by the installer.
- Note the directory into which Python was installed, probably c:\python27
- Add the Python directory to your path, so you can run it from the command line:
    - If Windows XP:
        - Start -> My Computer -> Properties
        - Select the 'Advanced' tab
        - Click "Environment variables"
        - You can update the path for 'Current User' and, if you have sufficient privileges, for 'System'. The latter is preferred if you want everybody to be able to run Python on the machine.
        - Add this to the end of the Path string:
          ;c:\python27
    - If Windows 7 (and probably also true for Vista) the instructions are almost identical:
        - Start -> Computer -> Properties
        - Click "Advanced system settings" on the left.
        - Click "Environment variables"

        - You can update the path for 'Current User' and, if you have sufficient privileges, for 'System'. The latter is preferred if you want everybody to be able to run Python on the machine.
        - Add this to the end of the Path string:
          ;c:\python27
        - Click "OK" as needed (i.e. for two dialog boxes)
- Test the Python installation:
    - Open a **new** command window. (Previously opened command windows won't see the new environment variable value.)
    - Type 'python'
    - The Python interpreter should start.
    - Type 'exit()'

Install Blueprint Processor From Zip File

- Unzip the blueprint distribution file to a directory, e.g. named bp_installation.
- To run the blueprint processor you will…
    - cd to bp_installation
    - Enter this command:

```
python bp_processor.py <filename> <options>
```

    - Alternatively, use:

```
bp_processor.py <filename> <options>
```

Test the Installation

- From the directory in which you placed the blueprint processor and other files…
- Enter this command to confirm you can run the blueprint processor, in this case just to see 'help' text:
    - python bp_processor.py helloWorld.yml -h
- You should see the 'help' text
- Enter this command to simulate running a blueprint without having to connect to a cloud environment:
    - python bp_processor.py helloWorld.yml
    - You should see something like this, in particular the output value "Hello World":

```
C:\...>python bp_processor.py interactiveTests\helloWorld.yml

Blueprint Processor - Invocation Summary
----------------------------------------
   Cloud URI:          sim
   User:               None
   Blueprint file:     helloWorld.yml
   Timeout:            90 minutes, 0 seconds
   Ellipses frequency: 15 seconds
   Inputs:
   Pause points:       (none)
   Debug logging:      False
   Instance name:      default_instance_name

17:10:31 WARNING: No Resources specified in blueprint.  Nothing will be created

17:10:31 INFO: Output Processing
17:10:31 INFO: -----------------
17:10:31 INFO:
17:10:31 INFO: Output values specified: 1
```

```
17:10:31 INFO:    Value of MyMsg: Hello World
17:10:31 INFO:
17:10:31 INFO: Blueprint Processing Summary
17:10:31 INFO: ---------------------------
17:10:31 INFO:
17:10:31 INFO: Timing Summary (seconds):
17:10:31 INFO:    Client-side CPU time: 0.218
17:10:31 INFO:    Elapsed time:
17:10:31 INFO:       Processing time:    0.0
17:10:31 INFO:       Paused time:        0.0
17:10:31 INFO:       Total elapsed time: 0.0
```

Optional Components for Graphical Summary Report

The blueprint processor can generate a summary report that includes a graphical depiction of the blueprint. (To generate such reports, use the –g or –G option.) For these options to produce reports, additional third party software is required and you must install it separately.

Install GraphViz

To install GraphViz, see http://www.graphviz.org. Download the software for your platform and follow the instructions.

Install Pydot

To install pydot, see http://code.google.com/p/pydot/. Download the software (zip or tar file). The blueprint processor was tested using pydot version 1.0.28.

PyDot can be installed using setuptools, e.g.

```
sudo easy_install pydot
```

One can also use the setup.py script in the zip/tar file. From the directory into which you unzipped or untarred the file, run that script:

```
python setup.py install
```

Running the Blueprint Processor

To deploy a blueprint, you run the blueprint processor and provide the name of the blueprint file plus any desired command options. For a full set of command options, use the command's –h option:

```
Usage:
  bp_processor.py [options] BlueprintFileName

Example:
  bp_processor.py myfile.yml -i "name:Joe" -i "count:2" -u jabauer -c https://...

Options:
  -h, --help            show this help message and exit
  -c CLOUD_URI, --cloud_uri=CLOUD_URI
                        Ex: https://myhost:4473/em/cloud
```

```
  -u USER, --user=USER  user id
  -p PASSWORD, --password=PASSWORD
                        password. If not provided, you'll be prompted.
  -t TIMEOUT, --timeout=TIMEOUT
                        timeout (seconds)
  -i INPUT_VALUE, --input_value=INPUT_VALUE
                        <param name>:<param value> e.g. -i "name:Joe Blogs"
  -r REFRESH_FREQUENCY, --refresh_frequency=REFRESH_FREQUENCY
                        # of seconds between dots in timeline. Zero: no dots
  -n INSTANCE_NAME, --instance_name=INSTANCE_NAME
                        name of blueprint instance to create
  -d, --debug           more logging, including http traffic
  -E, --pause_error     Drop into debugger if error occurs
  -I, --pause_input     Pause before processing Input section
  -R, --pause_resource  Pause before processing Resource section
  -O, --pause_output    Pause before processing Output section
  -T, --pause_terminate
                        Pause before termination, but after output processing
                        or error
  -g GRAPHIC_RESULTS, --graphic_results=GRAPHIC_RESULTS
                        Directory for deployed blueprint graphical report
  -G GRAPHIC_BLUEPRINT, --graphic_blueprint=GRAPHIC_BLUEPRINT
                        Directory for undeployed blueprint graphical report
```

Options that are not self-explanatory are described below:

- Input_value: For each input parameter, you can provide a –i or –input_value string of the form <param name>:<value>. You should use quotation marks around each input parameter specification, e.g. -i "name:Joe Blogs". To provide values for more than one input parameter, use the - i command line option more than once, e.g.

```
bp_processor.py _processor.py myfile.yml -i "DbZone:Zone1" -i "DbPassword:myPw"
```

- If an input parameter defined in the blueprint is not provided on the command line, you will be prompted
- Break_*: Drop into the blueprint debugger, just prior to beginning the Input/Resource/Output processing phases. In the case of break_terminate, the debugger is entered even in the event of an error. Use of the debugger commands is described in “Debugging with the Blueprint Processor”.
- If no CLOUD_URI is specified, the blueprint processing is simulated. See the section titled “Simulation Mode”.
- Timeout: The number of seconds after which the blueprint processor should terminate the deployment process. (If the processor is waiting for completion of a cloud request, e.g. a POST request, termination occurs when that request completes.)

Blueprint Processing Phases

Blueprint processing is done in phases. It’s useful to understand the phases, in order to

- Understand how blueprints are evaluated and resources created.
- Set breakpoints at phase transitions. (This can help with cloud browsing and debugging as explained later.)

The phases are:

- Initialization:
  - Parse the blueprint
  - Connect to the designated Cloud resource

- Input processing: If the blueprint defines input parameters and if some were not provided on the command line, prompt the user for the parameter values.
- Resource Creation: For each resource defined in the blueprint…
    - If the resource has no dependencies on other resources that haven't been created yet, initiate creation.
    - Monitor the process. If creation succeeds, other resources may then become unblocked for creation.
    - If creation fails, terminate the resource creation phase.
- Output processing: Evaluate and display any output values. (If the resource creation phase was terminated due to errors, some output values may not be available.)

Finally a summary of the blueprint deployment process is displayed.

A more detailed description of each phase follows.

Initialization

The blueprint file is read and parsed and a connection to the designated Cloud resource is made. The parsed blueprint content is captured in memory and then augmented. A member named Cloud is added, whose value is the Cloud resource as documented in [CRMA]. Another member named Info, which provides environmental information, is also added. Its value is a set of name/value pairs such as *time* and a string like '13:02:45' representing the time when processing began. Oracle-provided macros are loaded into the Macros section of the blueprint (except for any whose name conflicts with a blueprint-defined macro).

Input Parameter Evaluation

Input parameter processing is then done and the input parameters set. Any parameters specified in the blueprint that were not provided on the command line are prompted for.

Each input parameter value is stored in the *Value* attribute of the input parameter in the blueprint. The values can be accessed via this path expression: "Inputs.<parameterName>.Value". (See "Evaluation Intrinsics".)

Resource Creation

To process the Resources section, each resource is effectively processed in parallel. The following is done for each resource:

First, the expression specified for *Container* is evaluated. When successful and when the cloud resource it identifies is in the READY state, all expressions of the resource's Properties subsection are evaluated. (When evaluation of a resource's Properties or Container section can't be completed and must wait, the resource is marked and the blueprint processor proceeds to process other resources. Periodically, it reattempts to evaluate this and any other marked resources.)

Once all evaluation for a resource definition is complete, the document derived from the Properties section is used to request creation of the cloud resource (i.e. it is POSTed to the container URI). If successful, the URI of the newly created cloud resource is stored in the '_uri'

attribute of the blueprint's resource definition. At this point, that URI represents the resource that is being created.

The blueprint processor then polls the resource being created to track its status. From a state of CREATING, the resource should eventually transition to a success or failure state. If it transitions to a failure state, the blueprint processor diagnoses the situation and terminates. If creation succeeds and the resource enters the READY state, this may enable evaluation of other resource definitions to proceed.

Alternatively, the timeout value specified on the command line may get exceeded, in which case the blueprint processor wraps up its work and terminates. Any cloud resources whose creation was initiated may continue in the 'creating' state for some time before succeeding or failing.

In addition to timing out, other failures may occur such as an error evaluating an expression or an error code returned from the cloud, e.g. in response to a creation request. In all such cases, the blueprint processor diagnoses the situation and terminates.

Outputs

After all resources have been successfully evaluated, the Outputs section is processed, at which time each named output expression is evaluated and their values printed. Then a graphical report that depicts the blueprint and what got created is generated, if it was requested and a summary of blueprint processing is displayed to the user.

Language Specifics

YAML and JSON Concepts

A blueprint is a text file that represents a set of cloud resources one wants to create. The text is formatted in YAML or JSON. Both YAML and JSON are notations for representing data structures of lists and name/value pairs, which can be nested. A blueprint is such a structure. (See [YAML] and [JSON] for concise descriptions.)

You may be more familiar with JSON and are free to write a blueprint purely in JSON, but we recommend using YAML. With YAML you can write blueprints that are more concise and somewhat easier to read. YAML also offers useful capabilities not present in JSON, such as the ability to include comments. For these reasons, we've chosen to use YAML in the examples that follow. (Incidentally, YAML allows the inclusion of JSON notation in YAML documents, i.e. YAML is a superset of JSON.)

Overview of Blueprint Content

At the top level, a blueprint may contain any of the following name/value pairs, which can be viewed as blueprint section types:

- Inputs
- Data
- Macros
- Resources

- Outputs

Order of appearance in the blueprint is not significant. No other sections are allowed and each section can appear only once.

No section is truly *required* in a blueprint, but you must include a Resources section with at least one resource definition if you intend to create a cloud resource via the blueprint. A section may only appear once in a blueprint.

Inputs Section

The Inputs section is used to describe input parameters. Each parameter has a unique name and has the following attributes:

- Type: "String" or "Number". If "Number" is specified, the input value must be numeric. (Default: "String".)
- Prompt: A string to be used when prompting. (Default: The parameter name.)
- DefaultValue: A default value to assume if the user responds to the prompt by simply pressing Enter. (Default: "".)
- Order: A way to specify the order in which values will be prompted for. If not specified, ordering is arbitrary.
- Sensitive: "True" or "False". If "True", what the user enters interactively will not be echoed. (Default: "False".)
- Value: Set at runtime, using the value provided by the user (or the default value)

Processing input parameters starts with those input parameter values provided on the command line. For any not provided, the user is prompted.

Examples

In this blueprint snippet:

```
Inputs:
  UserId:
    DefaultValue: qa_user
    Prompt: User id
    Order: 1
  Password:
    Sensitive: True
    Order: 2
...
```

The author has defined 2 input parameters. Both are of type String. A default user id is specified but the user must provide a password. If neither is provided on the command line, the use of Order assures that UserId is requested first.

In this interaction:

```
C:\work>bp_processor.py test2.yml  -c https://... -u jon -p myPW -t 665

Blueprint Processor - Invocation Summary
----------------------------------------
   Cloud URI:               https://slc01rbw.us.oracle.com:15430/em/cloud
   User:                    ssa_user1
```

```
   Blueprint file:           xyzApp.yml
   Timeout:                  90 minutes, 0 seconds
   Refresh frequency:        15 seconds
   Inputs:
   Pause points:             Inputs
   Debug logging:            False
   Instance name:            default_instance_name
   Graphical report dir:     deployment_report
   Versions:
      Blueprint processor:   12.1.0.5, 10-Oct-2012
      Cloud protocol:        10001

16:19:18 INFO: Connecting to cloud: https://...
User id (qa_user):
Password:
...
```

The user provided no input parameters as part of the command line, so he is prompted for the two values. He pressed "Enter" for the user id, accepting the default of "qa_user". And he entered a password, which was not echoed.

The following interaction is almost the same, except that the user id is provided on the command line:

```
C:\work>bp_processor.py test2.yml  -c https://... -u jon -p myPW -t 665 -i "UserId:joe"

Blueprint Processor - Invocation Summary
----------------------------------------
   Cloud URI:                https://slc01rbw.us.oracle.com:15430/em/cloud
   User:                     ssa_user1
   Blueprint file:           xyzApp.yml
   Timeout:                  90 minutes, 0 seconds
   Refresh frequency:        15 seconds
   Inputs:
   Pause points:             Inputs
   Debug logging:            False
   Instance name:            default_instance_name
   Graphical report dir:     deployment_report
   Versions:
      Blueprint processor:   12.1.0.5, 10-Oct-2012
      Cloud protocol:        10001

16:19:18 INFO: Connecting to cloud: https://...
16:19:19 WARNING: No Resources specified in blueprint.  Nothing will be created
16:19:19 INFO: Creating blueprint instance named default_instance_name
Password:
...
```

In this case, the user is only prompted for a password.

Another example can be found at "Example 3 – Default Input Parameter Value via Cloud Lookup", which uses an intrinsic function to perform a runtime lookup for the default value.

Resources Section

The Resources section is used to describe the cloud resources you wish to create. Each resource description has a unique name and has the following attributes:

- Container: The URI of the parent cloud resource.

- Type: The media type of the resource, as defined by the Cloud Resource Model [CRMA]. (If the specified Container is a resource that is a subtype of ServiceTemplate, this value is optional and the default type of the ServiceTemplate is assumed.)
- Properties: A set of name/value pairs used to specify values required per the Cloud Resource Model.

As a blueprint author, you must know for each resource you wish to create, its type, its parent, and the properties you must provide to specify its characteristics. This information is all in the Cloud Resource Model document [CRMA].

For instance, a JavaPlatformInstance must specify:

- A container resource by provide the URI of a JavaPlatformTemplate. (You choose a template by selecting one that creates an instance most suited to your needs.)
- The properties required by the Cloud Resource Model. For JavaPlatformInstance, these are:
    - Name
    - Zone: (the URI of a zone)

Example

In a blueprint, the Resource description of a JavaPlatformInstance might look like this:

```
MyJavaServer1:
  Container:
    f_getTemplateURI:
      - Small WLS
      - jaas
  Properties:
    name: Foo
    zone:
      f_getZoneURI:
      - Zone1
      - jaas
```

The above doesn't specify 'Type:' because, this is optional when the container is a subtype of ServiceTemplate. Had we wanted to explicitly specify the media type, we could have written:

```
MyJavaServer1:
  Type: application/oracle.com.cloud.jaas.JavaPlatformInstance
  Container:
  ...
```

Our example uses the f_getTemplateURI and f_getZoneURI intrinsic functions to look up the required URI's. Resource definitions have access to numerous such functions as described in "Intrinsic Functions". Resource definitions can also use user-defined macros, as described in "Macros Section".

Outputs Section

The Outputs section is used to describe the set of "outputs" of a blueprint. In this release, outputs are just used by the blueprint author to specify information to display at the end of a successful deployment. For instance, he may define an output that displays the URL of an application deployed to a JavaPlatformInstance.

Each output description has a unique name and one required attribute named “Value”. That attribute generally specifies a blueprint expression whose value would be of interest to the user who instantiates a blueprint. (One may also include a “Description” attribute for each output definition.)

Example

The following Outputs section specifies one output to display:

```
Outputs:
  Application_URL:
    Description: URL of the deployed app
    Value:
      f_getResourceAttr:
        - MyApp
        - http_application_invocation_url
```

In this example, the output definition uses the f_*getResourceAttr* intrinsic function to retrieve the ‘http_application_invocation_url’ attribute of a newly created ApplicationInstanceDeployment. (See [CRMA] for more information on that attribute.)

Data Section

The Data section contains arbitrary YAML text. The data defined in this section can be accessed via intrinsic functions and can be used in various ways during blueprint deployment. For instance, the author of a blueprint may wish to use the concept of named literals to improve the readability and maintainability of a blueprint. He may want to use short descriptive names instead of long literal values such as cryptic URI’s. He may also anticipate needing to change a value used by a blueprint, in which case it can be specified once in the Data section and referenced throughout the blueprint.

Example

The following blueprint excerpts show a Data section that contains an item named *db_conn_str*, a long JDBC connect string that is referenced later in the a DataSource resource definition.

```
Data:
  db_conn_str: 'jdbc:oracle:thin:sysman/sysman@hostname.zzz.com:15044:smay16'
Resources:
  ...
  MyDatasource:
    Type: application/oracle.com.cloud.jaas.DataSource
    ...
    Properties:
      name: jbTest
      jdbc_driver: oracle.jdbc.OracleDriver
      database_type: Oracle
      database_connect_string:
        f_path:
          - Data.db_conn_str
      ...
```

Macros Section

The notation used in blueprints can be verbose. If you have a sequence of constructs that you tend to repeat, you can use macro expansion to improve the readability of your blueprint. Macros also enable you to encapsulate logic e.g. so that you can need only modify the logic in one place to affect all code that refers to it.

The Macros section is used to define macros that can be invoked/expanded from elsewhere in the blueprint. A macro invocation may occur wherever a function invocation is allowed, indeed, the notation is identical and one can't tell from the invocation whether it is an intrinsic function or a macro that is being invoked. (In this way, a macro can be used to override an intrinsic function. Also, some Oracle-provided intrinsic functions are implemented as macros.)

Each macro definition has a unique name and its definition specifies two values:

- The number of arguments it uses.
- The textual representation of the macro expansion.

When a macro is invoked, its textual representation replaces the invocation. Wherever the textual representation specifies a value of arg_<integer>, the value of that argument is used instead.

Example

Consider this somewhat contrived blueprint:

```
Macros:
  # Return a string that describes a resource being created
  # The one argument is a 'name' string
  f_myDescriptiveName:
    - 1
    - f_concat:
      - "Resource "
      - arg_1
      - " created for blueprint instance "
      - f_path:
        - "Info.instance_name"
      - " on "
      - f_path:
        - "Info.date"
Resources:
  MyJavaServer:
    Container:
      f_getTemplateURI:
        - Small WLS
        - jaas
    Properties:
      name:
        f_myDescriptiveName:
          - MyFirstJavaServer
      zone:
        f_getZoneURI:
        - Zone1
        - jaas
Outputs:
  NameOfServer:
    Value:
      f_path:
        - "Resources.MyJavaServer.Properties.name"
```

The Macros section defines one macro named f_myDescriptiveName, which takes one string argument and constructs a larger string adding descriptive information.

The macro is then invoked as part of the MyJavaServer resource definition. It is invoked with a value of "MyFirstJavaServer" and the resource will be created with a 'name' property whose

value is “Resource MyFirstJavaServer created for blueprint instance myQAInstance on 5/9/2012”.

```
Blueprint Processor - Invocation Summary
----------------------------------------
   Cloud URI:                https://...
   User:                     sysman
...
13:32:22 INFO:
13:32:22 INFO: Output values specified: 1
13:32:22 INFO:    Value of NameOfServer: Resource MyFirstJavaServer created for blueprint
instance myQAInstance on 5/9/2012
...
```

Expressions

A blueprint uses ‘expressions’ to compute values at deployment time. In many cases, the expressions are simple literal string values such as a *user_name* attribute whose value is ‘app_user’. In other cases, values are constructed via user-defined macros (see “Macros Section”) and intrinsic functions (see “Intrinsic Functions”).

Two intrinsic functions, *f_path* and *f_eval*, are provided to evaluate the two types of expression string:

- ‘Path’ expression
- ‘Eval’ (or ‘Blueprint’) expression

These expression types are described below. They can be used not only in calls to *f_path* and *f_eval* but also in the blueprint debugger.

‘Path’ Expressions

Path expressions are similar to JSONPath [JSONpath] and XPath expressions and are used to extract values from blueprints and cloud resources as well as to traverse URI’s that link you to other resources.

The starting point for evaluating a path expression is generally the in-memory blueprint. Recall that it contains all the information from your blueprint plus these attributes:

- **Cloud:** the cloud resource, which is defined in [CRMA] and has attributes that describe the overall cloud as well as those that enable you to traverse to all other cloud resources (to which you have access).
- **Info**: a section that contains runtime information you may wish to refer to such as the instance name specified when blueprint processing was initiated, the current date, etc.
- Other blueprint-processor-computed values such as the input parameters provided by the end-user and the URI’s of resources once they have been created.

All this information can be accessed via path expressions. The rest of this section summarizes syntax and semantics and provides examples. As you've seen in earlier sections, the *f_path* intrinsic function provides one way to evaluate path expressions. In the examples that follow, we use another mechanism, the blueprint debugger, described further in "Simulation Mode

The blueprint processor simulation mode can be used to aid in developing and testing blueprints. In this mode, the requests normally sent to the cloud server are simulated as well as the results returned by the server. Otherwise the blueprint processing logic is the same. To run the blueprint processor in this mode, you simply don't specify a cloud URI, i.e. don't use the *-c* option on the command line.

One benefit of simulation mode is the speed with which you can run a blueprint and try variations. Normal running of blueprints involves cloud requests for which the processing may be quite time consuming. When in simulation mode, the default behavior is that requests to create each resource consume 2 seconds and then succeed.

Another benefit is the ability to test various possibilities. For each resource, you can specify the simulated processing time as well as whether the request succeeds or fails. To do this for a given resource, use the Simulation attribute when defining a resource, e.g.

```
MyJavaServer1:
  Container:
    f_getTemplateURI:
      - Small WLS
      - jaas
  Properties:
    destination_zone:
      f_getZoneURI:
      - MyZone
      - jaas
    params:
      user: app_user
      password: pw_you_should_change
  Simulation:
     delay: 3
     result: f
```

In the above example, creation of MyJavaServer1 will fail after 3 seconds.
Debugging with the Blueprint Processor". As you'll see, you can use this not only to explore the contents of your blueprint runtime data but also to explore the contents of the cloud.

In the following descriptions, *<doc>* refers to the YAML data structure on which the operator acts. If I write:

```
Cloud.zones.elements[0]
```

Then the value of the runtime blueprint's *Cloud* attribute is the first *<doc>*. The dot-operator in '.zones' is applied to that *<doc>,* yielding a new *<doc>* whose value is that of the zones attribute of the first *<doc>*. Numerous examples are provided to illustrate the use of the operators.

Operator Summary

Dot Operator ("Member of"): <doc>.<name>

As seen in previous examples, the dot operator selects a value from a document. If <doc> has an attribute named <name>, the value of <name> is returned. Otherwise, expression evaluation fails.

Square Bracket ("List Indexing"): <doc>[<integer>]

If <doc> is a list of values, the value of the <integer>th element is returned. Otherwise, expression evaluation fails. Indexing is zero-based, i.e. the first element is specified as "<doc>[0]".

Dollar Sign ("Literal string prefix"): <doc> $ <string>

To begin a path expression with a string literal, use the '$' prefix, for instance:

```
$"/em/cloud/jaas/zone/A1B44A4EBCC4563125D9D0A3AAE4FD51" ->
```

In the above example, you know the URI of a specific resource and use the arrow operator to view its contents.

This syntax is only useful at the beginning of a path expression, but for consistency with the overall path notation, one can place the $ operator anywhere in a path expression. Other path expression operators operate on the <doc> to the left. The $ operator simply replaces the left value with the value of the literal string to the right.

In short, the $ operator returns the literal string value. If the right operand is not a literal string, expression evaluation fails.

Arrow ("URI Traversal"): <doc> ->

If <doc> is a URI, traverse to the identified resource and return its document. In other words, perform a GET on the URI specified by <doc>. If <doc> is not a URI or the GET fails, expression evaluation fails.

For instance, in this path expression…

```
Cloud.zones.elements[0].uri->
```

the expression to the left of the arrow operator returns the URI of a zone. The arrow operator is used to traverse to the zone, i.e. it performs a GET on the URI and displays the contents:

```
context_id: A1B44A4EBCC4563125D9D0A3AAE4FD51
description: Zone for Physical Pool
```

```
media_type: application/oracle.com.cloud.jaas.Zone+json
name: Zone1
resource_state:
  state: READY
service_family_type: jaas
service_instances:
  elements: []
  media_type: application/oracle.com.cloud.common.ServiceInstance+json
  total: '0'
uri: /em/cloud/jaas/zone/A1B44A4EBCC4563125D9D0A3AAE4FD51
```

In most cases the traversal operator is sufficient, but you can also specify traversal *qualifiers*. In particular, you can specify a media type and request parameters. These are optional and enclosed in square brackets. Multiple qualifiers can be specified, separated by commas.

To specify a media type, use a string (enclosed in quotes or double quotes).

To specify a request parameter there are two styles:

- Identifier
- Identifier = quotedValue

Overall there are 3 forms of traversal qualifier:

- *quotedValue*:
  media type
- *Identifier*:
  a request parameter that has no value
- *Identifier = quotedValue*
  a request parameter that *has* a value

For instance, the following path expression traverses to the cloud URI and specifies three qualifiers (a media type and two request parameters).

```
Cloud.uri-> ["application/oracle.com.cloud.common.ServiceTemplateFinds+json",
 filters='{"filters": {"service_family_type":"jaas"}', name]
```

The first parameter is the media type to be used to request that the cloud process a filter/query as described in [CRMA]. The remaining two are request parameters, also described in [CRMA]. The second is named 'filters' with a value that specifies the filtering to perform (expressed as a JSON string). The last is named 'name' and has no value. This specifies that the name attribute is to be returned.

Example: Viewing all values of Info

As noted above, the Info section contains environmental information that may be of use when constructing your blueprint. This example shows how to see those values that are currently available in the Info section.

Suppose you are looking for a value you can use to construct a unique name. You run the blueprint processor and enter debug mode. (You can do this by specifying "-I" on the command line.

```
C:\Users\jabauer\Dropbox\Code\blueprints>bp_processor.py helloWorld.yml
-c https://hostname.us.oracle.com:15430/em/cloud -u sysman -p sysman -I
...

Blueprint Processor - Invocation Summary
----------------------------------------
   Cloud URI:                https://hostname.us.oracle.com:15430/em/cloud
   User:                     sysman
   Blueprint file:           helloWorld.yml
...

...Pause point, prior to Input processing...
For command info, enter (h)elp

Paused: Info
date: 1/11/2013
date_suffix: '1_11_2013'
instance_name: default_instance_name
time: '16:47:7'
time_suffix: '16_47_7'
uuid: 81dcaf6895fa4fb881e82d1c16ef7025
```

Here, you see that there are 6 values stored in the Info section. There is one named *uuid* that is a universally unique hexadecimal string. You may prefer to go with *time_suffix,* as being sufficiently unique and more readable.

More values may be added between the time of writing this document and when the blueprint processor ships, so you can use this technique to see what's available in the version you are using.

Examples: Viewing Blueprint Values

Suppose you have a blueprint that begins with…

```
Inputs:
  DbPassword:
    Type: String
    DefaultValue: welcome1
    Prompt: Password to use for db
    Sensitive: True
```

Say you run the blueprint processor and specify command line options to pause just before processing the Inputs and Resources section, e.g. you specify "-RI".

```
...Pause point, prior to Input processing...
For command info, enter (h)elp

Paused: Inputs.DbPassword
DefaultValue: welcome1
Prompt: Password to use for db
Sensitive: true
Type: String
```

At the first pause point above, you enter the expression 'Inputs.DbPassword' and see that it has the attributes you specified in your blueprint. That includes *DefaultValue, Prompt*, and *Sensitive*. Note that it doesn't have an attribute named Value because input processing has not been performed yet.

```
Paused: c
...continuing...

Input Parameter Value Entry
---------------------------
   Password to use for db (welcome1):


...Pause point, prior to processing Resources section...
For command info, enter (h)elp

Paused: Inputs.DbPassword.Value
welcome1

Paused: Inputs.DbPassword
DefaultValue: welcome1
Prompt: Password to use for db
Sensitive: true
Type: String
Value: welcome1
```

You then enter 'c' to continue and are prompted for a password. You enter one, which is not echoed because you specified that it was 'Sensitive'.

Then the second (prior to processing the Resources section) pause point is reached. You enter 'Inputs.DbPassword.Value' to see the value of your password and then enter 'Inputs.DbPassword' to see the value of all attributes for the DbPassword input parameter.

Examples: Browsing Your Cloud

Path expressions also offer an easy way to explore the resources in the cloud and their attributes. That is because, at the beginning of blueprint processing, the cloud resource (as defined in [CRMA]) is read and placed into the in-memory blueprint structure. By starting your path expression with 'Cloud.', you can browse attributes of the Cloud resource and navigate via URI's to any other resource to which you have access.

To start, we look at the 'description' of the cloud to which you connected:

```
Paused: Cloud.description
This represents the Cloud resource of the Oracle Enterprise Manager Cloud Management
solution
Paused:
```

Now let's do something more useful, e.g. look at the cloud's 'zones' attribute:

```
Paused: Cloud.zones
elements:
- media_type: application/oracle.com.cloud.jaas.Zone+json
  name: Zone1
  service_family_type: jaas
  uri: /em/cloud/jaas/zone/A1B44A4EBCC4563125D9D0A3AAE4FD51
```

```
- description: Zone for Physical Pool
  media_type: application/oracle.com.cloud.common.DbZone+json
  name: Zone1
  type: self_service_zone
  uri: /em/cloud/dbaas/zone/A1B44A4EBCC4563125D9D0A3AAE4FD51
- media_type: application/oracle.com.cloud.opc.OpcZone+json
  name: OPC Zone
  service_family_type: opc
  type: opc
  uri: /em/cloud/opc/opczone
media_type: application/oracle.com.cloud.common.Zone+json
total: '3'
```

Here, we see that the 'zones' attribute contains three attributes, *elements, media_type,* and *total*. Their meanings are described in [CRMA].

We wish to focus on the first zone listed, so we use the square bracket (list indexing) syntax:

```
Paused: Cloud.zones.elements[0]
media_type: application/oracle.com.cloud.jaas.Zone+json
name: Zone1
service_family_type: jaas
uri: /em/cloud/jaas/zone/A1B44A4EBCC4563125D9D0A3AAE4FD51
```

We can further specify that we wish to focus on the 'uri' attribute by adding another dot-operator:

```
Paused: Cloud.zones.elements[0].uri
/em/cloud/jaas/zone/A1B44A4EBCC4563125D9D0A3AAE4FD51
```

To view the resource to which that URI refers, we add the arrow (traversal) operator:

```
Paused: Cloud.zones.elements[0].uri->
context_id: A1B44A4EBCC4563125D9D0A3AAE4FD51
description: Zone for Physical Pool
media_type: application/oracle.com.cloud.jaas.Zone+json
name: Zone1
resource_state:
  state: READY
service_family_type: jaas
service_instances:
  elements: []
  media_type: application/oracle.com.cloud.common.ServiceInstance+json
  total: '0'
uri: /em/cloud/jaas/zone/A1B44A4EBCC4563125D9D0A3AAE4FD51
```

As you'd expect, you can continue to add to your path expressions, for instance you can write "multi-hop" expressions that traverse multiple URIs, e.g. …

```
Paused: Cloud.service_templates.elements[0].uri->zones.elements[0].name
Zone1

Paused: Cloud.service_templates.elements[0].uri->zones.elements[0].uri->
context_id: A1B44A4EBCC4563125D9D0A3AAE4FD51
description: Zone for Physical Pool
media_type: application/oracle.com.cloud.jaas.Zone+json
name: Zone1
resource_state:
  state: READY
service_family_type: jaas
```

```
service_instances:
  elements: []
  media_type: application/oracle.com.cloud.common.ServiceInstance+json
  total: '0'
uri: /em/cloud/jaas/zone/A1B44A4EBCC4563125D9D0A3AAE4FD51

Paused:
```

In the above, we use the service_templates attribute of the Cloud to identify the first service template. We then traverse its URI to get to the template, where we identify the first zone in its list of supported zones. We then traverse its URI to get to the full definition of the zone.

In most cases the traversal operator is sufficient, but if there is no default media-type defined for the URI, you may need to specify the media-type to retrieve, as specified in [CRMA]. In the following example, we provide a media type even though it wasn't needed.

```
Paused: Cloud.service_templates.elements[0].uri->
["application/oracle.com.cloud.jaas.JavaPlatformTemplate"]

context_id: D2520A0CFFE348BCE040F20A4C1B2D8F
created: '2013-01-02 09:35:52.0'
default_instance_media_type: application/oracle.com.cloud.jaas.JavaPlatformInstance+json
...
```

Similarly, you can use the traversal qualifier syntax to specify request parameters as defined in [CRMA], e.g.

```
Paused: Cloud.service_templates.elements[0].uri->[created, resource_state]
created: '2013-01-02 09:35:52.0'
resource_state:
  state: READY
```

'Eval' (or 'Blueprint') expressions

An 'eval' (aka 'blueprint') expression is any expression you can include in your blueprint.

Example: Simple Intrinsic Function Evaluation

Eval expressions can be evaluated in the debugger via the 'e' or 'eval' command. After entering the command, you enter the lines that comprise the expression followed by an empty line. Suppose you want to experiment with the f_concat intrinsic:

```
Paused: e
  Eval: f_concat:
  Eval:  - xxx
  Eval:  - yyy
  Eval:
xxxyyy
Paused:
```

You enter the call to f_concat (in 3 lines) and the value is printed after you terminate the expression with an empty line.

This time you nest another call:

```
Paused: e
  Eval: f_concat:
  Eval:  - xxx
  Eval:  - f_path:
  Eval:    - 'Inputs.DbPassword.Value'
  Eval:  - yyy
  Eval:
xxxmySecretyyy
Paused:
```

As you can see, this provides a way to experiment with snippets of blueprint.

Example: Lookup Intrinsic Function

In this example, suppose your blueprint includes the use of f_getTemplateURI to look up a template URI:

```
Resources:
  MyDB:
    Container:
      f_getTemplateURI:
        - Small DB
        - dbaas
```

If you think the wrong URI is being returned, you can check like this…

```
Paused: e
  Eval:       f_getTemplateURI:
  Eval:         - template1
  Eval:         - jaas
  Eval:
/em/cloud/jaas/javaplatformtemplate/BFAB458D36BDA87EE040E50A038F6D45
Paused:
```

This shows the URI value returned by f_getTemplateURI with the arguments you entered.

Intrinsic Functions

Intrinsic functions are functions that blueprints can use to compute/return desired information. For instance, the *f_concat* function returns the concatenation of its string arguments and the *f_getZoneURI* function looks up and returns the URI of a zone. The normal usage of intrinsic functions is to provide values needed as part of the Resources and Outputs sections, but an intrinsic function can be placed wherever a literal value is allowed. For instance, one can use an intrinsic to derive the DefaultValue used for an Input parameter.

This section describes the currently available intrinsic functions.

Evaluation Intrinsics

These two intrinsics are used to evaluate expressions (of different types) and return a single value.

f_path(pathExpr)

Apply the pathExpr string to the blueprint document, returning the specified value.

Parameters

- **derefString**: e.g. "member.subMember…"
  A path expression that describes how to traverse and extract information from the document. See "'Path' Expressions".

Example – Value from Data section

To specify a resource's property value using a literal value defined in the Data section of your blueprint …

```
...
  params:
    MasterUser:
      f_deref:
        - "Data.QADBCreds.user"
...
```

Example 2 – Value from Inputs section

To do the same as above, only using an input parameter value …

```
...
  params:
    MasterUser:
      f_deref:
        - "Inputs.my_param.value"
...
```

Example 3 – Default Input Parameter Value via Cloud Lookup

Intrinsic functions can appear in sections other than the Resources section. For instance, this blueprint shows the use of f_path in the Inputs section.

```
Inputs:
  JavaSvcZone:
    DefaultValue:
      f_path:
        - 'Cloud.zones.elements[0].name'
    Type: String
    Prompt: Enter the name of a jaas zone
...
```

The blueprint author here wants to provide an arbitrary default zone name, so the path expression selects the first zone that appears in the 'zones' attribute of the cloud.

The default value (shown in the prompt) is computed at runtime:

```
...

Input Parameter Value Entry
---------------------------
   Enter the name of a jaas zone (east_coast_zone):
```

f_eval(blueprintExpr)

Evaluate *blueprintExpr*, returning the specified value. See "'Eval' (or 'Blueprint') expressions" for information on blueprint expressions.

Parameters

- **blueprintExpr**: YAML text to be evaluated as if it appeared in a blueprint.

Example (contrived)

The f_eval intrinsic is used internally by the blueprint processor, and it's unlikely you'll need to use it. (You just use the blueprint expression directly.) One reason you might want to use this function is if you have a variable whose value is a blueprint expression in the form of a YAML string.

```
Data:
  demoOfYamlMultilineText: |
    This is a multi-
    line text string which is
    carefully indented. :-)
  myBlueprintExpressionText: |
    f_concat:
      - 'Mister '
      - 'Mxyzptlk'
Outputs:
  demoOfYamlMultilineText:
    Value:
      f_path:
        - 'Data.demoOfYamlMultilineText'
  myBlueprintExpressionText:
    Value:
      f_path:
        - 'Data.myBlueprintExpressionText'
  useOfEvalonExpressionText:
    Value:
      f_eval:
        - f_path:
          - 'Data.myBlueprintExpressionText'
```

Which results in this output:

```
18:49:58 INFO: Output Processing
18:49:58 INFO: -----------------
18:49:58 INFO:
18:49:58 INFO: Output values specified: 3
18:49:58 INFO:    Value of demoOfYamlMultilineText: This is a multi-
line text string which is
carefully indented. :-)
18:49:58 INFO:    Value of myBlueprintExpressionText: f_concat:
  - 'Mister '
  - 'Mxyzptlk'
18:49:58 INFO:    Value of useOfEvalonExpressionText: Mister Mxyzptlk
```

Resource Access Intrinsics

These intrinsics are used to access resource attributes. As part of their operation, unlike f_path, they assure that the resource is in the READY state, waiting if needed.

f_getResourceAttr(bpResName, derefString)

Get the value of a cloud resource attribute after it is READY.

Parameters

- **bpResName**: Resource name (specified in blueprint)
- **pathExpr**: Same semantics as used in f_deref, only against the document of the cloud resource identified by bpResName.
- **Returns**: result of evaluating *pathExpr* of the resource, once it is created and its resource_state is READY.

Example

To add an application to a MW platform, an Application resource can use this Container clause…

```
Container:
  f_getResourceAttr:
    - myJavaPlatform
    - uri
```

Example 2

To access the name of the zone in which your MW platform was created, you can write…

```
f_getResourceAttr:
  - myPlatform
  - zone.name
```

f_getResourceURI(bpResName)

Get the URI for a blueprint-defined resource. This is just a shorthand for using f_getResourceAttr with the specified attribute being 'uri'.

Parameters

- **bpResName**: Name used in the blueprint resource definition
- **Returns**: URI

Example

To define a Datasource resource that is to be contained in a JavaPlatformInstance resource created elsewhere in your blueprint…

```
  MyDatasource:
    Type: Datasource
    Container:
      f_getResourceURI:
        - MyJavaServer
```

Lookup Intrinsics

These intrinsics search for a template, zone, or application component, returning its URI.

f_getTemplateURI(name, type)

Get the URI for a template, based on its name and type.

Parameters

- **name**: Template name
- **type**: A service type name. The current list of allowed values is *iaas, jaas,* and *dbaas*
- **Returns**: URI

Example

A blueprint resource to create a database using template *simpleDb,* could be written…

```
Container:
  f_getTemplateURI:
    - simpleDb
    - dbaas
```

f_getZoneURI(name, type)

Get the URI for a zone, based on its name and type.

Parameters

- **name**: Zone name
- **type**: A service type name. The current list of allowed values is *iaas, jaas,* and *dbaas*
- **Returns**: URI

Example

To get the URI of zone EMEA_db_zone…

```
  f_getZoneURI:
    - EMEA_db_zone
    - dbaas
```

f_getAppCompURI(name, owner, version) …

Get the URI for an application component, based on its name, owner, and version.

Parameters

- **name**: Application component name
- **owner**: Owner of application component
- **version**: Version of application component. If blank, the latest version is used.
- **Returns**: URI

Example

To get the URI of the most recent version of the application component *jbcomponent*, owned by SSA_USER1 …

```
  f_getAppCompURI:
    - jbcomponent
    - SSA_USER1
    -
```

Note that the third argument is required.

Debugging Intrinsics

These intrinsics are used to establish breakpoints or printpoints.

f_break(expression, [breakpointMessage])

Pause evaluation of the blueprint, print optional message, and enter blueprint debugger.

Parameters

- **expression:** any blueprint expression
- **breakpointMessage**: Message to be printed when the intrinsic function is invoked, just prior to entering the debugger.
- **Returns**: Value of *expression.* This is computed when the 'continue' command is entered.

Notes

See "Simulation Mode

The blueprint processor simulation mode can be used to aid in developing and testing blueprints. In this mode, the requests normally sent to the cloud server are simulated as well as the results returned by the server. Otherwise the blueprint processing logic is the same. To run the blueprint processor in this mode, you simply don't specify a cloud URI, i.e. don't use the *-c* option on the command line.

One benefit of simulation mode is the speed with which you can run a blueprint and try variations. Normal running of blueprints involves cloud requests for which the processing may be quite time consuming. When in simulation mode, the default behavior is that requests to create each resource consume 2 seconds and then succeed.

Another benefit is the ability to test various possibilities. For each resource, you can specify the simulated processing time as well as whether the request succeeds or fails. To do this for a given resource, use the Simulation attribute when defining a resource, e.g.

```
MyJavaServer1:
  Container:
    f_getTemplateURI:
      - Small WLS
      - jaas
  Properties:
    destination_zone:
      f_getZoneURI:
      - MyZone
      - jaas
    params:
      user: app_user
      password: pw_you_should_change
  Simulation:
     delay: 3
     result: f
```

In the above example, creation of MyJavaServer1 will fail after 3 seconds.

Debugging with the Blueprint Processor" for examples and other information on how to use breakpoints to help debug your blueprints.

f_print(expression, [printpointMessage])

Print a line that displays the value of *expression.* Pause evaluation of the blueprint, print optional message, and enter blueprint debugger.

Parameters

- **expression:** any blueprint expression
- **printpointMessage**: Message to be printed when the intrinsic function is invoked, just prior to entering the debugger.
- **Returns**: Value of *expression*

Notes

The line printed looks like this:

```
>>> Printpoint [<printpoint message>]:
    Value = <expression>
```

See "Simulation Mode

The blueprint processor simulation mode can be used to aid in developing and testing blueprints. In this mode, the requests normally sent to the cloud server are simulated as well as the results returned by the server. Otherwise the blueprint processing logic is the same. To run the blueprint processor in this mode, you simply don't specify a cloud URI, i.e. don't use the *-c* option on the command line.

One benefit of simulation mode is the speed with which you can run a blueprint and try variations. Normal running of blueprints involves cloud requests for which the processing may be quite time consuming. When in simulation mode, the default behavior is that requests to create each resource consume 2 seconds and then succeed.

Another benefit is the ability to test various possibilities. For each resource, you can specify the simulated processing time as well as whether the request succeeds or fails. To do this for a given resource, use the Simulation attribute when defining a resource, e.g.

```
  MyJavaServer1:
    Container:
      f_getTemplateURI:
        - Small WLS
        - jaas
    Properties:
      destination_zone:
        f_getZoneURI:
        - MyZone
        - jaas
      params:
        user: app_user
        password: pw_you_should_change
    Simulation:
```

```
      delay: 3
      result: f
```

In the above example, creation of MyJavaServer1 will fail after 3 seconds.
Debugging with the Blueprint Processor" for examples and other information on how to use printpoints to help debug your blueprints.

Other Intrinsics

f_concat(string1, … stringN)

Return the concatenation of the string arguments..

Parameters

- **string***: A string to be concatenated with the other string arguments

Example

To set the description of a JavaPlatformInstance to "Created by blueprint FOO on <current date>" …

```
Resources:
  MyJavaServer:
    ...
    Properties:
      description:
        f_concat:
          - "Created by blueprint FOO on "
          - f_path:
            - "Info.date"
    ...
```

Dealing with Errors

This section illustrates various types of errors you may encounter and, by example, how to interpret/resolve the issues.

YAML syntax errors

Any syntax errors encountered by the YAML parser are diagnosed by the parser. Consider this blueprint snippet…

```
# Example of YAML syntax error
Data:
   userId: Lex
  password: changeMe
...
```

In the above example, the YAML parser would detect an indentation error and diagnose it like this:

```
18:55:59 ERROR: Error loading blueprint YAML:
```

```
while parsing a block mapping
  in "<string>", line 2, column 1:
    Data:
    ^
expected <block end>, but found '<block mapping start>'
  in "<string>", line 4, column 3:
      password: changeMe
      ^
```

The second half of the diagnostic, the "expected" part, is usually the most helpful. In this case, it tells you the error was detected at the token 'password', what it was expecting, and what it found.

One common error to avoid is the use of tabs in the YAML file. YAML does not allow tab characters.

Protocol Version Mismatch

```
A diagnostic like this…
Blueprint Processor - Invocation Summary
----------------------------------------
   Cloud URI:              http://hostname.us.oracle.com:4473/em
   User:                   sysman
   Blueprint file:         examples/evalintrinsic.yml
   Timeout:                90 minutes, 0 seconds
   Refresh frequency:      15 seconds
   Inputs:
   Pause points:           Inputs, Termination
   Debug logging:          False
   Instance name:          default_instance_name
   Versions:
      Blueprint processor: 12.1.0.4 May 25
      Cloud protocol:      10001

19:05:06 INFO: Connecting to cloud: http://adc2100705.us.oracle.com:4473/em
19:05:07 ERROR: Cloud protocol version mismatch.  Expected 10001.  Found None.
```

The diagnostic in bold, indicates that the blueprint processor was able to connect to the site but did not get the expected response. In particular, the site returned no x-specification-version value as part of the HTTP response.

This can happen if you specified an incorrect cloud URI. For instance, in the above example, the URI was not of the form

```
https://host:port/em/cloud
```

A common mistake is to omit the "/cloud". Also, don't forget to use *https*, not *http*.

If the diagnostic indicates it found a protocol version that is lower than the one expected, that may indicate that you are using a version of the blueprint processor that requires a more recent version of EM.

Expression Evaluation Error

As part of blueprint processing, an attempt to evaluate an expression may result in an error. Examples of errors include:

- Passing the wrong number of parameters to an intrinsic function
- Referring to a non-existent intrinsic function
- Referring to a non-existent cloud resource such as a zone or template

When an expression evaluation error occurs, the issue is diagnosed and the expression is displayed. For instance, consider this contrived blueprint:

```
Outputs:
  ExampleValue:
    Value:
      f_concat:
        - MyApp
```

To keep the example short, the blueprint only has an Outputs section and one expression, which is a call to f_concat. Notice that f_concat requires at least two parameters, but only one is provided. When the blueprint processor is run for this blueprint, this is displayed:

```
15:26:25  INFO: Output Processing
15:26:25  INFO: -----------------
15:26:25  INFO:
15:26:25  INFO: Output values specified: 1
15:26:25 ERROR:    Value of ExampleValue: Expression could not be evaluated.  Error is...
15:26:25 ERROR: Function/macro concat requires parameter count between 2 and 99, not 1
15:26:25 ERROR: Expression being evaluated at the time:
15:26:25 ERROR:    {'f_concat': ['MyApp']}
15:26:25  INFO:
```

The blueprint processor attempts to print the value of 'ExampleError' when the error occurs. After displaying the diagnostic, the expression being evaluated at the time of the error is displayed (in JSON notation).

Expressions are generally nested, and the error may occur within a subexpression. In that case, the diagnostic includes an expression stack, so that you can see the specific expression in error as well as the outer context. For instance, consider this contrived blueprint:

```
Outputs:
  ExampleValue:
    Value:
      f_concat:
        - aaa
        - bbb
        - f_concat:
          - ccc
```

Notice that the expression involves two uses of f_concat. The outer use is correct, but there is an error with the inner use. When the blueprint processor is run for this blueprint, this is displayed:

```
15:42:50  INFO: Output Processing
15:42:50  INFO: -----------------
15:42:50  INFO:
```

```
15:42:50  INFO: Output values specified: 1
15:42:50 ERROR:    Value of ExampleValue: Expression could not be evaluated.  Error is...
15:42:50 ERROR: Function/macro concat requires parameter count between 2 and 99, not 1
15:42:50 ERROR: Expression evaluation stack follows (with failed expression at bottom)
...
15:42:50 ERROR: -----
15:42:50 ERROR: |   Expr: {'f_concat': ['aaa', 'bbb', {'f_concat': ['ccc']}]}
15:42:50 ERROR: |   Expr: {'f_concat': ['ccc']}
15:42:50 ERROR: -----
15:42:50  INFO:
```

The expression evaluation stack shows the outer expression at the top and the expression with the error at the bottom. This stack only has two levels but in general there are many levels and each level of evaluation is shown. (The next example illustrates a multi-level expression evaluation stack.)

When a macro is evaluated, its definition is expanded and this expansion is shown in the expression evaluation stack. For instance, consider this contrived blueprint:

```
Outputs:
  ExampleValue:
    Value:
      f_getZoneURI:
        - myZone
        - jaas
```

In this case, the expression is a call to f_getZoneURI and the error is that the zone 'myZone' does not exist. When the blueprint processor is run for this blueprint, this is displayed:

```
15:57:17  INFO: Output Processing
15:57:17  INFO: -----------------
15:57:17  INFO:
15:57:17  INFO: Output values specified: 1
15:57:18 ERROR:    Value of ExampleValue: Expression could not be evaluated.  Error is...
15:57:18 ERROR: Name not found: myZone
15:57:18 ERROR: find_one for predicate {'f_EQ': [{'f_pathc': ['name']}, 'myZone']} failed
15:57:18 ERROR: Expression evaluation stack follows (with failed expression at bottom)
...
15:57:18 ERROR: -----
15:57:18 ERROR: |   Expr: {'f_getZoneURI': ['myZone', 'jaas']}
15:57:18 ERROR: |   Expr: {'f_pathc': [{'f_findByName': [{'f_pathc': [{'f_findByName':
[{'f_path': ['Cloud.service_family_types.elements']}, 'jaas']}, '.uri-.zones.elements']},
'myZone']}, '.uri']}
15:57:18 ERROR: |   Expr: {'f_findByName': [{'f_pathc': [{'f_findByName': [{'f_path':
['Cloud.service_family_types.elements']}, 'jaas']}, '.uri->.zones.elements']}, 'myZone']}
15:57:18 ERROR: |   Expr: {'f_findOne': [{'f_pathc': [{'f_findByName': [{'f_path':
['Cloud.service_family_types.elements']}, 'jaas']}, '.uri->.zones.elements']}, {'f_EQ':
[{'f_pathc': ['name']}, 'myZone']}, {'f_concat': ['Name not found: ', 'myZone']}]}
15:57:18 ERROR: -----
15:57:18  INFO:
```

When the error is detected, a diagnostic is displayed: "Name not found: myZone". In this case, the error should be clear and you need not bother reading further.

But to illustrate how macro expansion and multi-level nested expressions are shown in the expression evaluation stack, we continue walking through the example. The top line of the stack shows the outer expression, which is what was specified in the blueprint as the expression for

'Value:'. Because the f_getZoneURI intrinsic is implemented as a macro, line 2 of the expression evaluation stack shows the expression after macro expansion. (It is long, so line wrapping is needed.)

Line 3 shows the subset of line 2 that was being evaluated when the error occurred and line 4 shows the same expression after the f_findByName macro was expanded.

Note: Intrinsic macros use some internal functions, which is why you see names like f_pathc.

Hint: Use '-T' Option

While developing and testing your blueprint, it's a good idea to use the -T command line option. This tells the blueprint processor to drop you into the debugger prior to terminating for any reason. Should any results be unexpected, whether an outright error or just unexpected output, you can use the debugger to investigate.

Cloud Resource Creation Error

In the above example, the failure to create a resource was detected some time after the request was accepted. In some cases, the request to create may fail immediately.

If an error occurs while attempting to create a resource, you will see a diagnostic that identifies the resource, the error code, and some diagnostic text. In the following example, the attempt to create a resource name MyJavaPf failed with an HTTP code of 500. Reading further, you can see diagnostic text like, "cannot process request for …", "Unable to start the Instance deployment" , and "stack_trace_cause" : "java.lang.IllegalArgumentException: Unable to service executable from service template…"

```
14:43:37 INFO:          MyJavaPf
14:43:37 INFO:         /
14:43:37 INFO:        /    MyDatasource
14:43:37 INFO:       /    /
14:43:37 INFO:   -----------
14:43:37 INFO:   | e  |    |
14:43:38 INFO:   | es |    |
14:43:42 ERROR: Failure creating resource MyJavaPf: 500
{
  "messages" :
  [
      {
        "date" : "2012-05-22T18:43:42+0000" ,
        "text" : "cannot process request for
oracle.sysman.emInternalSDK.ssa.cloudapi.ResourceInteraction@767d6a37 on /em/cloud/jaas/
javaplatformtemplate/C086733BCCF2A4F3E040F10A716049A8" ,
        "hint" : " Unable to start the Instance deployment" ,
        "stack_trace_cause" : "java.lang.IllegalArgumentException: Unable to service
executable from service template : C086733BCCF2A
4F3E040F10A716049A8\n\tat
oracle.sysman.ssa.mwaas.model.util.remoteop.DPSubmissionHelper._createRequestMWaasSetup(D
PSubmissionHelper.
java:359)\n\tat
oracle.sysman.ssa.mwaas.model.util.remoteop.DPSubmissionHelper._submitMWaasSetupServiceRe
```

```
quest(DPSubmissionHelper.jav
a:616)\n\tat
oracle.sysman.ssa.mwaas.model.util.remoteop.DPSubmissionHelper.submitMWaasSetupServiceReq
uest(DPSubmissionHelper.java:71
2)\n\tat
oracle.sysman.ssa.cloudapi.jaas.JavaPlatformInstance.GenerateJavaPlatformInstance(JavaPla
tformInstance.java:369)\n\tat oracl
e.sysman.ssa.cloudapi.jaas.JavaPlatformTemplate.processRequest(JavaPlatformTemplate.java:
128)\n\tat oracle.sysman.ssa.cloudapi.jaas.J
aasServiceProvider.processRequest(JaasServiceProvider.java:520)\n\tat
oracle.sysman.emInternalSDK.ssa.cloudapi.EMCloudServlet.perform
(EMCloudServlet.java:226)\n\tat
oracle.sysman.emInternalSDK.ssa.cloudapi.EMCloudServlet.performPost(EMCloudServlet.j" ,
        "stack_trace" :
"oracle.sysman.emInternalSDK.ssa.cloudapi.CloudServiceException:  Unable to start the
Instance deployment\n\t
at
oracle.sysman.ssa.cloudapi.jaas.JavaPlatformInstance.GenerateJavaPlatformInstance(JavaPla
tformInstance.java:373)\n\tat oracle.sysm
an.ssa.cloudapi.jaas.JavaPlatformTemplate.processRequest(JavaPlatformTemplate.java:128)\n
\tat oracle.sysman.ssa.cloudapi.jaas.JaasSer
viceProvider.processRequest(JaasServiceProvider.java:520)\n\tat
oracle.sysman.emInternalSDK.ssa.cloudapi.EMCloudServlet.perform(EMClo
udServlet.java:226)\n\tat
oracle.sysman.emInternalSDK.ssa.cloudapi.EMCloudServlet.performPost(EMCloudServlet.java:3
63)\n\tat oracle.s
ysman.emInternalSDK.ssa.cloudapi.rest.AbstractRestServlet.doPost(AbstractRestServlet.java
:134)\n\tat javax.servlet.http.HttpServlet.s
ervice(HttpServlet.java:727)\n\tat
javax.servlet.http.HttpServlet.service(HttpServlet.java:820)\n\tat
weblogic.servlet.internal.StubS
ecurityHelper$ServletServiceAction.run(StubSecurityHelper.java:227)\n\tat
weblogic.servlet.internal.StubSecurityHe"
      }
  ]
}


14:43:42 INFO:   | CF |    |
14:43:42 INFO:   -----------
14:43:42 INFO:
14:43:42 ERROR: Create of resource MyJavaPf failed
```

In this example, the HTTP code is 500. Any code that begins with a 5 indicates that the cloud server encountered an unexpected exception. This could be due to an environment issue or even a bug in the server software. Since a 5xx code reflects a server error, you should contact the SSA administrator.

In other cases, you may see a 4xx error code, which is returned when the client seems to have erred. In such cases, you should check the 'hint' and 'message' information for clues as to what went wrong, because you may be able to correct an error you made.

The blueprint processor lists the diagnostic information it receives, but for security reasons, the cloud server may not provide sufficient information to diagnose the issue. If so, you'll want to contact the SSA Cloud administrator, who in turn can often diagnose the issue by reviewing the log files for the cloud request.

Simulation Mode

The blueprint processor simulation mode can be used to aid in developing and testing blueprints. In this mode, the requests normally sent to the cloud server are simulated as well as the results returned by the server. Otherwise the blueprint processing logic is the same. To run the blueprint processor in this mode, you simply don't specify a cloud URI, i.e. don't use the *-c* option on the command line.

One benefit of simulation mode is the speed with which you can run a blueprint and try variations. Normal running of blueprints involves cloud requests for which the processing may be quite time consuming. When in simulation mode, the default behavior is that requests to create each resource consume 2 seconds and then succeed.

Another benefit is the ability to test various possibilities. For each resource, you can specify the simulated processing time as well as whether the request succeeds or fails. To do this for a given resource, use the Simulation attribute when defining a resource, e.g.

```
MyJavaServer1:
  Container:
    f_getTemplateURI:
      - Small WLS
      - jaas
  Properties:
    destination_zone:
      f_getZoneURI:
      - MyZone
      - jaas
    params:
      user: app_user
      password: pw_you_should_change
  Simulation:
     delay: 3
     result: f
```

In the above example, creation of MyJavaServer1 will fail after 3 seconds.

Debugging with the Blueprint Processor

In addition to running the blueprint processor such that it deploys the blueprint and runs to completion, there are mechanisms you can use to debug blueprints. These are akin to mechanisms you may have used for debugging other applications, like print statements and the use of a debugger to interactively display values used by your application.

To enter the debugger at a particular point of execution, you can use either of two mechanisms. The simpler approach, which will usually be sufficient, is to use command line options that cause execution to pause between processing phases. These are called "pause points" and are described below. The alternative approach enables you to break at a more specific point, such as just prior to evaluating an expression for a specific resource's property. To do this, you edit the blueprint to include a breakpoint.

When either a pause point or breakpoint is reached, control is transferred to the "debugger". In the debugger, you enter various commands to display contents of the blueprint as well as that of the cloud to which you are connected.

Printing Intermediate Results

The essence of blueprint processing is to evaluate expressions and create resources once all required expressions have been evaluated. At any point during evaluation of an expression, you may wish to see some intermediate results to confirm the value is what you expected. To do so, you use the intrinsic function *f_print.*

Wherever an expression can appear in a blueprint, you simply nest it in a call to *print.* Optionally, you can include a second *text message* argument. When *print* is encountered, the text message and expression value are printed.

Examples

This (contrived) example shows how you plan to use a lookup table to access a template name for use in a call to f_getTemplateURI.

```
Data:
  MyTemplates:
  - {name: DbTemplate, type: dbaas}
  - {name: MWTemplate, type: jaas}
...
Resources:
  MyDB:
    Container:
      f_getTemplateURI:
        - f_path:
            - 'Data.MyTemplates[0].name'
        - dbaas
    Properties:
...
```

Suppose the code is not behaving as you intend, and you want to view the intermediate results before passing the name to f_ getTemplateURI. Wrap the expression in a call to f_print like this:

```
Data:
  MyTemplates:
  - {name: DbTemplate, type: dbaas}
  - {name: MWTemplate, type: jaas}
...
Resources:
  MyDB:
    Container:
      f_getTemplateURI:
        - f_print:
          - f_path:
              - 'Data.MyTemplates[0].name'
        - dbaas
    Properties:
...
```

At runtime, the value of the expression is printed:

```
16:52:05  INFO:
```

```
16:52:05  INFO: Resource State Timeline
16:52:05  INFO: ----------------------
...
16:52:05  INFO:
16:52:05  INFO:         MyDB
16:52:05  INFO:        /
16:52:05  INFO:    ------
16:52:05  INFO:    | e  |
>>> Print-point:
    Value = DbTemplate
...
```

A print point message can also be provided, which is useful when you have multiple print points in your blueprint:

```
Data:
  MyTemplates:
  - {name: DbTemplate, type: dbaas}
  - {name: MWTemplate, type: jaas}
...
Resources:
  MyDB:
    Container:
      f_getTemplateURI:
        - f_print:
          - f_path:
             - 'Data.MyTemplates[0].name'
          - My printpoint for template name
        - dbaas
    Properties:
...
```

At runtime, the value of the expression is printed:

```
16:52:05  INFO:
16:52:05  INFO: Resource State Timeline
16:52:05  INFO: ----------------------
...
16:52:05  INFO:
16:52:05  INFO:         MyDB
16:52:05  INFO:        /
16:52:05  INFO:    ------
16:52:05  INFO:    | e  |
>>> Print-point: My printpoint for template name
    Value = DbTemplate
...
```

Pause Points

The easiest way to specify points at which to enter the debugger is via are specified via command line options as described in “Optional Components for Graphical Summary Report”

The blueprint processor can generate a summary report that includes a graphical depiction of the blueprint. (To generate such reports, use the –g or –G option.) For these options to produce reports, additional third party software is required and you must install it separately.

Install GraphViz

To install GraphViz, see http://www.graphviz.org. Download the software for your platform and follow the instructions.

Install Pydot

To install pydot, see http://code.google.com/p/pydot/. Download the software (zip or tar file). The blueprint processor was tested using pydot version 1.0.28.

PyDot can be installed using setuptools, e.g.

```
sudo easy_install pydot
```

One can also use the setup.py script in the zip/tar file. From the directory into which you unzipped or untarred the file, run that script:

```
python setup.py install
```

Running the Blueprint Processor". You can specify that the blueprint processor pause and the debugger entered at any of these points:

- Prior to evaluating the Input section and prompting for Input parameters
- Prior to evaluating the Resources section
- Prior to evaluating the Output section
- Prior to termination, but after output processing or detecting an error that will terminate processing
- When an error is encountered, just prior to termination

Once the debugger is entered, you can use the commands described in "Debugger Commands" below. To continue blueprint execution, enter the "continue" command.

Examples

- Suppose you just want to browse the cloud resources at your server. Specify the –I option, which drops you into the debugger before attempting any blueprint processing.
- It's often useful to specify the '-E' (or '—error_debug') option, which drops you in the debugger if an error is encountered. (Otherwise, execution simply terminates.)

Breakpoints

Breakpoints are defined via the *f_breakpoint* intrinsic function as described in "Debugging Intrinsics ". Whenever evaluation encounters an *f_breakpoint* invocation, the optional text string parameter value is printed and the debugger entered.

Debugger Commands

When you enter the debugger, you'll see a prompt of "Paused:". At this prompt, there are several commands you can use including "help" or "h", e.g.

```
Paused: h
Commands are...
   p[ath] <path expression>: evaluate path expr
   e[val]:                   read & evaluate blueprint expression
   c[ontinue]:               continue blueprint instantiation
```

```
   x[it]:                    exit blueprint processor
   h[elp]:                   (this command)
If first token isn't a command, the line is treated as a path expression
Paused:
```

"Path" command

The "path" command is used to evaluate arbitrary path expressions as described earlier in "'Path' Expressions". (Unlike other debugger commands, the "p" or "path" keyword is not required.)

Example: Viewing values in blueprint

You can view values in your blueprint such as the value of an input parameter:

```
C:\bp> bp_processor.py -c https://...:15430/em/cloud -u sysman -p sysman -R xyzApp.yml
 ...

Blueprint Processor - Invocation Summary
----------------------------------------
   Cloud URI:                https://hostname.us.oracle.com:15430/em/cloud
   User:                     sysman
   Blueprint file:           xyzApp.yml
   Timeout:                  90 minutes, 0 seconds
   Refresh frequency:        15 seconds
   Inputs:
   Pause points:             Resources
   Debug logging:            False
   Instance name:            default_instance_name
   Graphical report dir:
   Versions:
      Blueprint processor:   12.1.0.5, 10-Oct-2012
      Cloud protocol:        10001

18:28:14  INFO: Connecting to cloud: https://hostname.us.oracle.com:15430/em/cloud

Input Parameter Value Entry
---------------------------
   Zone to use for db (Zone1):
   Password to use for db (welcome1):


...Pause point, prior to processing Resources section...
For command info, enter (h)elp

Paused: path Inputs
DbPassword:
  DefaultValue: welcome1
  Prompt: Password to use for db
  Sensitive: true
  Type: String
  Value: mySecret
DbZone:
  DefaultValue: Zone1
  Prompt: Zone to use for db
  Type: String
  Value: Zone1


Paused: Inputs.DbPassword.Value
```

```
mySecret

Paused:
```

In the above example, the simple test blueprint specifies two input parameters, DbZone and DbPassword.

1. When prompted, you accepted the default, for the first parameter by pressing “enter”. For the second, you entered your password.
2. Notice that your command line options included “-R”, which tells the blueprint processor to pause just prior to evaluating the Resources section of your blueprint. The “Paused” prompt appears, and you enter the “path” command with the path expression “Inputs”. The value of the Inputs section of the blueprint is then printed, namely the 2 input parameters and their values. The values include both those provided by the blueprint and the current runtime values, in this case ‘Zone1’ and ‘mySecret’.
3. You then simply entered a path expression. The “path” command is assumed if no explicit command is entered.

“Continue” command

The *continue* command is used to resume blueprint processing.

Example

Continuing the previous example…

```
Paused: Inputs.MyNum
{Sensitive: true, Type: Number, Value: '123'}

Paused: continue
```

“Exit” command

The *exit* command terminates the blueprint processor.

“Eval” command

The eval command is used to evaluate any expression you can include in your blueprint.

Example

Suppose you are debugging your blueprint and it appears to be failing when looking up a template by name. You can use the eval command to evaluate expressions that appear in your blueprint.

First you try executing the expression of interest as it appears in your blueprint:

```
Paused: e
  Eval: f_getTemplateURI:
  Eval:   - JaaS Template
  Eval:   - jaas
  Eval:
18:05:01 ERROR: Name not found: JaaS Template
Expression evaluation stack follows (with failed expression & diagnostic at bottom) ...
-----
|   Expr: {'f_getTemplateURI': ['JaaS Template', 'jaas']}
|   Expr: {'f_path': [{'f_findByName': [{'f_path': [{'f_findByName': [{'f_path':
['Cloud.service_types.elements']}, 'jaas']}, '.uri->
```

```
.service_templates.elements']}, 'JaaS Template']}, '.uri']}
|   Expr: {'f_findByName': [{'f_path': [{'f_findByName': [{'f_path':
['Cloud.service_types.elements']}, 'jaas']}, '.uri->.service_tem
plates.elements']}, 'JaaS Template']}
|   Expr: {'f_findOne': [{'f_path': [{'f_findByName': [{'f_path':
['Cloud.service_types.elements']}, 'jaas']}, '.uri->.service_templa
tes.elements']}, {'f_EQ': [{'f_path': ['name']}, 'JaaS Template']}, {'f_concat': ['Name
not found: ', 'JaaS Template']}]}
| End of stack for error message: Name not found: JaaS Template
-----
Paused:
```

In the above example, you first enter the 'eval' or 'e' command. Then you enter the expression. Note that indentation is significant, as it always is in YAML.

You see the same diagnostic you got when processing your blueprint, but now you can experiment with other values. Eventually, you realize that the template was created with two spaces in the name. You try with that name. It works, and the result of the expression evaluation, in this case a URI, is displayed.

```
Paused: e
  Eval: f_getTemplateURI:
  Eval:   - JaaS  Template
  Eval:   - jaas
  Eval:
/em/cloud/jaas/javaplatformtemplate/C086733BCCF2A4F3E040F10A716049A8
Paused:
```

Appendix A: Hints, Tips, and Frequently Asked Questions

Editing YAML – Notepad++ Example

YAML documents use indentation to denote containment semantics. This may affect your choice of editor or editing options. For instance, YAML doesn't allow tabs, so you should disable any editor options that cause automatic tab insertion.

As an example, suppose you use Notepad++ (http://notepad-plus-plus.org) . You would set the "replace by spaces" setting in Preferences -> Language Menu/Tab Settings. Better still, if your file has a suffix of ".yml", Notepad++ sets options to be suitable for YAML. For instance, it colorizes the text based on YAML syntax. If you don't use ".yml", you can manually set the language to YAML. (Settings -> Preferences -> Language Menu -> …)

YAML and Duplicate Name/Value Pairs

YAML requires that name/value pairs at the same level use unique names. Any duplicates override earlier occurrences.

For instance, these two blueprints are equivalent:

```
Data:
  Password: doNotChangeMe
  UserId: QA_user
```

```
  Password: changeMe
Resources:
  ...
```

```
Data:
  UserId: QA_user
  Password: changeMe
Resources:
  ...
```

Explicit Dependencies

Should you are blueprint defines two resources X and Y. If creation of X depends on the successful creation of Y, there will usually be a data-dependency between the two. However, if that's not the case, you can include anywhere in the definition of ResourceX an expression like …

```
f_getResourceAttr:
  - ResourceY
  - uri
```

English Only?

Due to schedule constraints, this version of the blueprint processor is not localized.

Help / Forums

Other questions? Post them at https://forums.oracle.com/forums/forum.jspa?forumID=220 .

References

[CRMA] *Oracle Cloud Resource Model API;* The Cloud Resource Model API is described in ***Oracle® Enterprise Manager Cloud Administration Guide, 12.1.0.5, Part VII, Using the Cloud APIs***

[Intro] ***Introduction to Cloud Blueprints, Describing a Set of Related SSA Instances***, included in the same directory in which this document was installed.

[JSON] ***Introducing JSON***, http://www.json.org/

[JSONpath] *JSONPath - XPath for JSON,* http://goessner.net/articles/JsonPath/

[YAML] http://yaml.org/