DSL Training

SKO 2016

Topics Covered

- Domain Specific Languages and ElectricFlow DSL
- Difference between ec-perl and DSL usage
- Running ElectricFlow DSL
- Groovy - Key features and syntax style
- Mapping ElectricFlow objects and APIs to DSL
- DSL samples, tips and tricks

DSL Basics

What is a Domain Specific Language?

- A computer language that is targeted to a particular problem space
  - In contrast to a general purpose language that is aimed at any kind of software problem
- Some common examples of a Domain Specific Language (DSL)

DSL	Domain
SQL	Database management
HTML	Web pages
Chef recipes	Infrastructure automation

What is ElectricFlow DSL?

Intuitive, easy to use domain specific language that allows us to model **continuous delivery** and **application release automation**

Key features of ElectricFlow DSL

- Intuitive, easy to use and understand
- Self documenting
- Process-as-code
- Allows modeling applications, deployment pipelines and releases using simple, logical constructs
- Based on Groovy language
- Supports various means of code reuse
- Complete access to ElectricFlow API

```
pipeline 'ProductABCPipeline', {
   description = 'pipe line from dev to
prod'
   stage 'dev', {
       description = 'dev stage'
       task 'run build', {
           ...
       }
   }
   stage 'test', {
       description = 'test stage'
       task 'manual regression', {
           ...
       }
       task 'QE automation', {
           ...
       }
   }
}
```

ec-perl vs. DSL

```
use strict;
use ElectricCommander;
$| = 1;
my $ec = new ElectricCommander
({'format'=>'json'});
$ec->createProject("Hello Project");
$ec->createProcedure("Hello Project",
       "Hello Procedure");
$ec->createStep("Hello Project",
       "Hello Procedure",
       "echo ‘Hello World from ec-perl’");
```

```
ec-perl helloProcedure.pl
```

```
project "Hello Project", {
  procedure "Hello Procedure", {
       step "Hello World",
       command: "echo ‘Hello World from EF DSL!’"
  }
}
```

```
ectool --format json evalDsl -dslFile
helloProcedure.groovy
```

What are some of the differences you can see?

ec-perl vs. DSL : Differences

ec-perl	DSL
Script-based, imperative (how)	Declarative (what)
Relatively verbose	Concise
Perl language understanding required	Groovy understanding not required to read DSL
Client-server chattiness - each API invocation is a separate call to the server	Entire DSL script is sent once to the server for evaluation

Running DSL

1. From the Command-line
2. Using REST
3. From a Step Command

Running DSL - Command-line

ectool evalDsl <dsl>

ectool evalDsl “project ‘My Project’”

ectool evalDsl --dslFile <path to dsl file>

ectool evalDsl --dslFile c:/dslScripts/myProject.dsl

Running DSL - Using REST

URL: http://<server>:8000/rest/v1.0/server/dsl

DSL Example using REST

Request	Response

URL: http://localhost:8000/rest/v1.0/server/dsl
HTTP Method: POST
Content-Type: application/json
Payload:
{
"dsl":
"project 'Default', {
 	
application 'TestApp', {
artifactLocation = 'my artifact location'
}
}"
}
	{
	"project": {
	"projectId": "7e238199-bef1-11e5-ae75-34e6d71279c8"
	"projectName": "Default"
	...
	"workspaceName": ""
	}
	}

Running DSL - From a Step Command

Step Command: <dsl script>

Shell: ectool evalDsl --dslFile {0}

Running DSL - evalDsl arguments

Argument	Description
dsl	The DSL text. E.g., `ectool evalDsl "project 'myProject'"`
dslFile	DSL script file. Applicable only when evalDsl is invoked through ectool or ec-perl, not through REST API. Either dsl or dslFile needs to be provided.
debug	Optional. If this argument is set to `true` or 1, ElectricFlow generates debug output as the DSL script is evaluated.
parameters	Optional. Parameters to be passed to the DSL script during evaluation in JSON text format.
describe	Optional. If this argument is set to `true` or 1,ElectricFlow prints a description of the specified DSL method. E.g., `ectool evalDsl pipeline --describe 1`
serverLibraryPath	Optional. Path on the server directory that contains external library jars and classes that should be added to the classpath when evaluating the DSL script.
applicationName, .., zoneName	Optional. All arguments starting from application to zone to allow relative property lookup. E.g., `ectool evalDsl "getProperty 'artifactLocation'" --applicationName TestApp --projectName Default`

Groovy features

Groovy key features and syntax style

- **return** keyword is optional - Use it only if it adds to code readability

```
project 'Hello Project', {
  procedure 'Hello Procedure', {
      step 'Hello World',
      command: 'echo Hello World from EF DSL!'
  }
}

project 'Hello World'
```

Returns the last project created/updated

- semi-colons optional - Don't use it - keeps the DSL looking more natural
- Method parenthesis can be removed

```
zone 'Data center A'
```

Groovy key features and syntax style - contd.

- **with** for repeated operations on the same object

```
ec_content_details.with {
   source = artifactPath
   artifact = 'mycompany.war'
   version = '1'
   directory = ''
   pluginProcedure = 'Retrieve File Artifact'
   pluginProjectName = 'EC-FileSysRepo'
}
```

Adds nested properties to ec_content_details property sheet

Groovy key features and syntax style - contd.

- Interpolated strings
  - Single-quoted strings are regular (Java) strings - Use these by default

```
‘no $magic here’ // will result in exactly ‘no $magic here’
```

  - Any groovy expression can be interpolated in double-quoted strings

```
def magic=’Groovy magic’
“$magic here” // will result in “Groovy magic here”
```

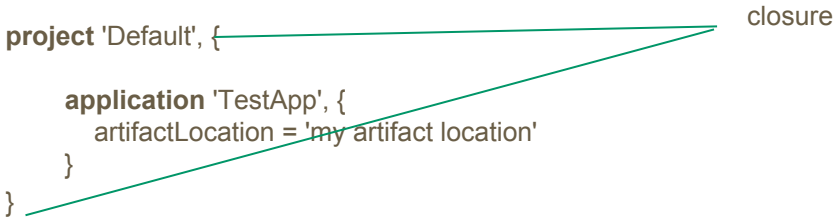
- Multi-line strings
  - Triple single-quoted strings are multiline - Use these where needed

```
command: ‘’’
set -x
cp mycompany.war mycompany_bk.war'''
’’’
```

  - Triple double-quoted strings are multiline and allow interpolation

Groovy key features and syntax style - contd.

- Closures
  - A closure in Groovy is an open, anonymous, block of code that can take arguments, return a value and be assigned to a variable.
  - Used in DSL for defining nested objects



  - A lot of Groovy default methods for collections use closures

Groovy key features and syntax style - contd.

- Groovy truth, Elvis operator and more here.

ElectricFlow objects to DSL

Mapping ElectricFlow objects to DSL

- ElectricFlow objects model the applications, pipelines, and all other continuous delivery, and application release processes
- DSL provides a one-to-one mapping to the ElectricFlow objects

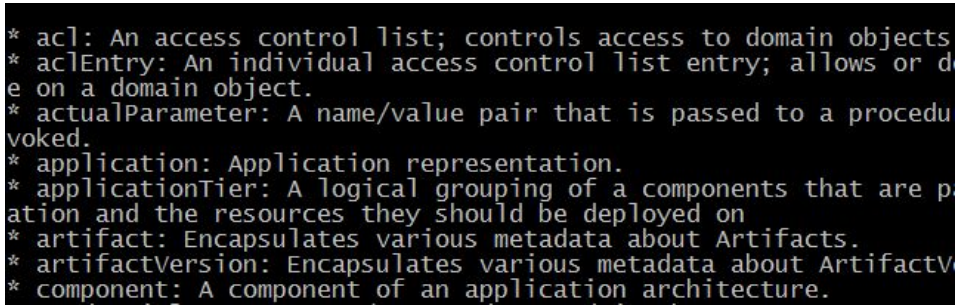
```
pipeline ‘lightSaberiOSApp’, {
      stage ‘development’, {
           task ‘build’, {..}
           task ‘tests’, {..}
       }
       stage ‘beta’, {
           task ‘distribute’, {..}
       }
}
```

Mapping ElectricFlow objects to DSL - contd.

- Same arguments as perl for create/modify<*EFObject*> are available in DSL
- Use the following options with **evalDsl** to get information on the available ElectricFlow objects

1. **ectool evalDsl dsl --describe 1**

   Provides a complete listing of DSL methods for ElectricFlow objects




Pick one method and use it to get started with the DSL

Mapping ElectricFlow objects to DSL - contd.

2. **ectool evalDsl** <DSL method> **--describe 1**

   Provides details for a given DSL method

   ectool evalDsl pipeline **--describe 1**

3. **ectool evalDsl** <ElectricFlow API> **--describe 1**

   Provides details for a given ElectricFlow API

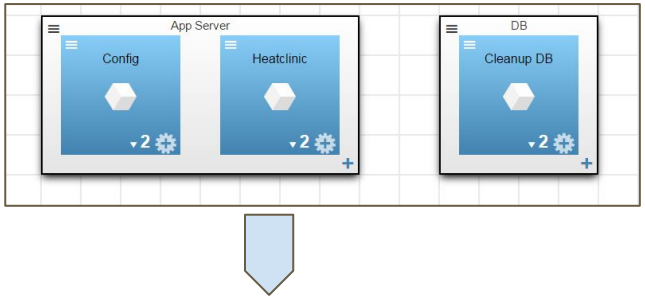
   ectool evalDsl getApplications **--describe 1**

```
List of supported attributes:

* pipelineName: (Required) The name of the pipeline
* projectName: (Required) Name for the project; must be unique among all project
s.
* description: Comment text describing this object; not interpreted at all by El
ectricFlow.
* enabled: True to enable the pipeline.
* newName: New name for an existing object that is being renamed.
* type: Type of pipeline


List of DSL methods for ElectricFlow objects that can be nested within:

* formalParameter
* property
* stage


-- 'pipeline' Syntax --
pipeline (
    pipelineName: pipelineNameValue,
    projectName: projectNameValue
) {
    // Add DSL methods here for ElectricFlow objects that may be nested within '
pipeline'
}

Or

pipeline (pipelineNameValue) {
    // Add DSL methods here for ElectricFlow objects that may be nested within '
pipeline'
}

-- Example using 'pipeline' --

pipeline (
    //pipeline attributes
    pipelineName: pipelineNameValue,
    projectName: projectNameValue) {

    stage (
        //stage attributes
        stageName: stageNameValue) {

        task (
            //task attributes
            taskName: taskNameValue)
    }
}
```

Copy one of the example snippets and try to use it to build your pipeline

Using generateDsl

- generateDsl allows you to generate DSL for an existing object
- Capture your existing processes in DSL
- A great way to get started with DSL!



ectool generateDsl /projects/Default/applications/HeatClinic

```
application 'HeatClinic', {
  projectName = 'Default'

  // Custom properties

  property 'ec_deploy', {

    // Custom properties
    ec_notifierStatus = '0'
  }
  jobCounter = '98'

  applicationTier 'App Server', {
    applicationName = 'HeatClinic'
    projectName = 'Default'

    component 'Config', pluginName: 'EC-FileSysRepo-0.0.4.81789', {
      applicationName = 'HeatClinic'
      pluginKey = 'EC-FileSysRepo'
      projectName = 'Default'
      reference = '0'
      sourceComponentName = null
      sourceProjectName = null

      // Custom properties

      property 'ec_content_details', {

        // Custom properties
        artifact = 'env.sh'
        directory = null
        pluginProcedure = 'Retrieve File Artifact'
        pluginProjectName = 'EC-FileSysRepo'
        source = '/net/f2home/dkarpenko/heatclinic/qe'
        version = '1'
      }

      process 'Start server', {
        applicationName = null
        processType = 'DEPLOY'
        projectName = 'Default'
        timeLimitUnits = null
        workspaceName = null

        processStep 'Retrieve', {
          applicationTierName = null
```

Tips and Tricks and Samples

Strings, references and expansions

Run the procedure after creating it through evalDsl from the command line. What does the procedure print?

```
project 'Training', {

    procedure 'Training Procedure', {
        step 'sayHello', {
            command = 'echo "Hello World from $[/myProcedure/procedureName]!"'
        }

    }
}
```

Strings, references and expansions

- Now, try the same after creating the procedure through a step command. (Copy-paste the dsl script in the **Command** section and set **Shell** to ectool evalDsl --dslFile {0}). What does the procedure print this time?

- Why did the ‘Training Procedure’ print ‘Hello World from Training Procedure’ in the first case but not this time? Try to fix it.

- Change the DSL to print the procedure name with single quotes now.

Strings, references and expansions

- Here is a more practical example for avoiding eager expansions

```
component componentName: ‘comp name‘, pluginKey: ‘EC-Artifact‘, {
          ec_content_details.with {
               pluginProjectName = ‘EC-Artifact‘
               pluginProcedure = ‘Retrieve‘
               artifactName = ‘abc:def‘
               filterList = ‘‘
               overwrite = ‘update‘
               versionRange = artifactVersion
               artifactVersionLocationProperty =
                  ‘/myJob/retrievedArtifactVersions/$’ + ‘[assignedResourceName]’
          }
          …
}
```

Iterating over ElectricFlow object collections

- Use the Groovy default methods to iterate over collections returned by ElectricFlow APIs.

```
project 'Hello Project', {
  getProcedures.each {
    procedure it.procedureName, {
      step 'New step',
           command: 'echo Hello World from EF DSL!'
    }
  }
}
```

'it' is an implicit parameter to the closure for the current element.

- Groovy default methods that can be used with collections
  - each, eachWithIndex
  - find, findAll, any
  - collect
  - and more

Debugging DSL

- Use debug option to trace how the DSL is being evaluated by the ElectricFlow server

  ectool evalDsl --dslFile <path_to_file> --debug 1

- Use DSLIDE!
- Use return values for simple debugging

```
def debug = ‘’
…dsl…
debug += ‘ log this’
…dsl…
debug // put this at the end of the DSL script so it is returned by evalDsl
```

Modular Scripting approach using Files

- Refer to Application Onboarding on Github
  - Step commands DSL in different files on the server
  - Files can be tested independently and reused

Modular approach using classes and 3rd-party libraries

- Create utility Groovy classes that can be reused across scripts
- Fail-safe alternative to @Grab
- Steps required:
  - Copy required Groovy utility classes and jars to a known directory on the server
  - Run evalDsl with the serverLibraryPath option

  ectool evalDsl --dslFile <dsl file> --serverLibraryPath <path to directory on server>

- Example - Excel Sheet Reader

Transactions in DSL

- By default, the entire DSL script runs within a single transaction
- **transaction** DSL method creates a new transaction for the closure and evaluates the closure within it.

```
------------- TX1 : Begin -------------
project 'Hello Project', {
  procedure 'Hello Procedure', {
      step 'Hello World',
      command: 'echo Hello World from EF DSL!'
  }
}
------------- TX1 : Commit ------------
------------- TX2 : Begin -------------
transaction {
    runProcedure procedureName: 'Hello Procedure', projectName: 'Hello Project'
}
------------- TX2 : Commit ------------
```

And finally

Known Issues

- evalDsl
  - No API bindings for DSL to allow using through EC-Groovy
  - No over-write option available - equivalent of a ‘force’ import
  - Artifact publish/retrieval not available through DSL

- generateDsl
  - Has some issues in 6.0 - Current plan to fix in 6.0.4.
  - Public release 6.2 - was a hidden API prior to 6.2
  - Currently does not support application snapshots

References

ElectricFlow DSL

- Greg's DOES'15 session on ElectricFlow DSL
- Laurent's DSL Training presentation
- ElectricFlow DSL Online Documentation
- Sample DSLs on Github
- DSLIDE - Web-based ElectricFlow DSL IDE

Groovy

- Groovy Style Guide
- Groovy Language Documentation

That’s all folks!