Polymer User Guide

connectifex

v1.00 - 2020 08 28

Table of Contents

Revision History

Revision	Author	Changes
v1.00 2020 08 28	Peter Strong	Initial creation

1 Introduction

This guide is intended for use by anyone who wants a simplified approach to take advantage of the OpenDaylight Plastic model-to-model translation mechanisms.

The polymer tool provides a Mapping Definition Language (MDL) and a variety of utilities that:

- allow for creation of new mapping projects
- allow for the specification of plastic folder structures
- generate new mappings from example JSON payloads
- allow for the definition and execution of tests to exercise your mappings
- generate plastic assets such as input/output schemas and morphers
- allow for the specification of reusable pattern matching components

The generated morphers provide no-code support for:

- optional variables
- URL encoding of variables
- validation of variables based on defined sets of values
- validation of variables based on full regular expression (regex) matching
- splitting of variables into multiple variables based on regex matching
- normalization of variables that match a regex

1.1 Prerequisites

You must install the Java 8 runtime environment.

Head to JDK download page and download the version appropriate to your platform.

Install the JDK.

1.2 Download polymer

Download the polymer jar.

Save the jar to a convenient location e.g. ~/bin

1.3 Setup to run polymer

If you're running in a UN*X environment, simply create an alias, such as:

```
alias polymer='java -jar ~/bin/[jar name]'
```

When you enter `polymer` at the command prompt, you should see:

```
Please specify one of the following utilities as the first argument:

gen     Runs the plastic file GENeration utility
new     Runs the new module creation utility
pattern Runs plastic PATTERN test utility
test    Runs plastic TEST utility
var     Runs the VARiablizer utility
guide   Displays the user guide
```

1.4 Editing MDL Files

The Mapping Definition Files (MDL) used by `polymer` are plain ASCII files; you may use whatever editor you like.

However, if you are using Eclipse, or wish to install it, you can then configure it to use the Java Editor to edit the `.mdl` files.

- Click Eclipse->Preferences
- Select General->Editors->File Associations
- For the File Types section click Add… and enter:
  `*.mdl`
- That file type will be selected in the list
- For the Associated Editors section, click Add…
  And scroll through the list and select Java Editor and click okay
- Click Apply and Close

Using Eclipse to edit `.mdl` files allows for commenting/uncommenting blocks of text using the "command / " shortcut. This will insert/remove Java comment markers "//" at the beginning of each line selected.

2 Getting Started

The following sections will guide you through the initial use of `polymer`.

We'll go through some basic concepts first and then a series of simple tutorials.

If you're already familiar with the basics, you can see more details in the Mapping Definition Language chapter.

2.1 Basic Concepts

2.1.1 Modules

What you will be creating are ASCII files that contain Mapping Definition Language (MDL) specifications. These files, called `Modules`, end with a `.mdl` file extension; this lets the `polymer` tool know that it can read those files.

2.1.2 Objects

A `Module` contains a series of `Object` specifications that express various concepts e.g. plastic mappings, patterns, tests etc. Each `Object` begins with a line that indicates its `Class` i.e. what type of `Object` it is. You can see a description of the available MDL Classes in the next chapter.

`Objects` are separated from one another by one or more blank lines.

We'll show some examples shortly.

2.1.3 Attributes

Each `Object` is composed of a series of `Attributes` (sometimes referred to as properties).

Generally, each `Attribute` exists on a single line of the file.

An `Attribute` can span multiple lines if the lines that follow the initial line start with a whitespace character e.g a space or tab.

Attributes come in a few different flavors, including Multi-valued Attributes and Complex Attributes. We'll discuss these as needed.

2.1.4 Example MDL Objects/Attributes

Every `Module` begins with an `MdlModule` object - here's an example:

```
MdlModule
name tutorials
description This module recreates the plastic tutorial by specifying, in the
 Mapping Definition Language (MDL).
```

MdlModule	This is the `Class` of object
name	This is an attribute that indicates a unique name for the object. NOTE: the name of a module must match the name of the file in which it is specified i.e. this module would be in a file named `tutorials.mdl` NOTE: within a module, every object MUST have a unique name. NOTE: an attribute name in green means that it is mandatory.
description	This is an attribute that provides a description of the module. You'll notice that it spans multiple lines; the second line begins with a space to indicate that it's a continuation of the line above. NOTE: an attribute name in blue means that it is optional.

Objects in a `Module` are separated from one another by one or more blank lines. If we added a `FolderStructure` object to the file it would look like this:

```
MdlModule
name        tutorials
description This module recreates the plastic tutorial by specifying, in the
 Mapping Definition Language (MDL).

FolderStructure
name         tutorial
inFolder1    api-in  note="Gathering input schemas in one folder"
outFolder1   api-out note="Gathering output schemas in another folder"
description  This defines the top level name for the generated plastic files and
 subfolders for the input and output schemas.
```

The `FolderStructure` object also has an example of complex attributes, in this case `inFolder1` and `outFolder1`.

3 Tutorials

The following tutorials lead you through using most of the functionality provided by polymer, starting from scratch

3.1 Create a new module.

Step 1	Create a new folder in which to run the tutorial and cd to that folder.
Step 2	In a terminal window run: `polymer new -outdir . -name tutorials` This will create a file called `tutorials.mdl`

Step 3	Open `tutorials.mdl` with your preferred editor
	
	You should see something like:
	
	`MdlModule`
	`name tutorials`
	`description Add a description of the module here. If your description`
	`spans multiple lines, start subsequent lines with whitespace.`
	
	`FolderStructure`
	`name tutorials-structure`
	`inFolder1  api-in`
	`outFolder1 api-out`
	`description Provide a description of your chosen folder structure.`
	
	`// Your PlasticMappings will go here - use "polymer var" to help generate them`

Throughout the various tutorials, you will continue to add content to this `tutorials.mdl` file.

3.2 Create a new PlasticMapping

In this section, we create a basic plastic mapping. All plastic mappings consist of an input schema that is mapped to a corresponding output schema.

Although plastic supports both JSON and (to a more limited extent) XML schemas, polymer currently supports only JSON schemas.

Step 1	In a terminal window run:
	`polymer var`
	
	You should see:
	
	`Enter i for input schema or o for output schema:`

Step 2	Enter:
	`i`
	
	And hit return and you'll see:
	
	`Paste in a JSON Yang configuration, end your input with a blank line:`

Step 3	Paste the following after the prompt:
	
	`{`
	`  "admin-status":  "UP",`
	`  "min-mtu":  "1500",`
	`  "max-delay-msec": "10"`
	`}`
	
	And hit return twice
	
	You'll see the following:
	
	`Here's an example PlasticMapping, minus the output information`
	
	`PlasticMapping`
	`name  insert-mapping-name`
	`folderStructure tutorials-structure`
	`variables  admin-status  example="UP" default="UP" note="Add notes here"`
	`variables  max-delay-msec example="10`
	`variables  min-mtu  example="1500`
	`inputSchema {`
	`  "admin-status":  "${admin-status}",`
	`  "max-delay-msec": "${max-delay-msec}",`
	`  "min-mtu":  "${min-mtu}"`

	<pre> }
	outputSchema {
	     "add-your-schema": "here"
	 }
	description Document your mapping here...
	
	// And here's an example PlasticTest that uses your specified input example:
	
	PlasticTest
	name insert-mapping-name-test1
	mapping insert-mapping-name
	inputPayload {
	    "admin-status": "UP",
	    "max-delay-msec": "10",
	    "min-mtu": "1500"
	  }
	
	Enter i for input schema or o for output schema:</pre>

Step 4	Copy the section highlighted in orange and paste it at the end of the tutorials.mdl file.
Step 5	Replace `insert-mapping-name` with `tutorial01` in: • the PlasticMapping `name` attribute • the PlasticTest `name` attribute • the PlasticTest `mapping` attribute Save the file.

We are going to pause here for a moment and look a bit more closely at the generated `PlasticMapping` and discuss the concept of variables.

3.3 The concept of variables

The concept of a variable is central to the entire plastic approach of schema-to-schema (or model-to-model) mapping.

Let's look at our first `PlasticMapping`:

```
PlasticMapping
name            insert-mapping-name
folderStructure tutorials-structure
variables    admin-status   example="UP" default="UP" note="Add notes here"
variables    max-delay-msec example="10"
variables    min-mtu        example="1500"
inputSchema  {
    "admin-status":   "${admin-status}",
    "max-delay-msec": "${max-delay-msec}",
    "min-mtu":        "${min-mtu}"
  }
outputSchema {
     "add-your-schema": "here"
 }
```

The parts of the `inputSchema` highlighted in **blue** are plastic variables.

In raw plastic schema specifications (that we'll be generating shortly) these variables indicate where plastic is going to find values that it will subsequently insert into the `outputSchema` (although we haven't gotten to that part yet).

Plastic lets you indicate a few other things when you specify variables, but we'll discuss those aspects later.

The thing that's different with polymer (as opposed to raw plastic) is that these variables are also listed in the multi-valued `variables` attribute. This is a complex attribute with multiple fields.

The sections highlighted in **green** are the names of the polymer variables; these are mandatory fields of the `variables` attribute.

The other fields are optional:

example	a documentation field meant to provide examples of what values might be in this variable
default	a default value for when a value isn't provided (much more on this later)
note	a place to document further things that might be useful for someone maintaining the project

NOTES: The names specified in the `variables` attribute and the `inputSchema` must align - polymer will generate an error if there's a mismatch.

The names of variables in the `inputSchema` MUST be unique.

The names of variables **HAVE NOTHING TO DO WITH** the property names in the JSON!

In this example, because we've generated the `PlasticMapping`, the `polymer var` tool has simply used the property names directly as variable names. However, if we had nested JSON structures, you would see much more complex variable names that don't happen to align to property names.

You could change the plastic variables to look like:

```
inputSchema  {
    "admin-status":   "${status}",
    "max-delay-msec": "${delay}",
    "min-mtu":        "${schrodingers-cat}"
  }
```

Now, let's continue with trying to generate some plastic assets.

3.4 Generate plastic assets

Step 1	In a terminal window run:
	`polymer gen -modules tutorials`
	
	You should see something like:
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`
	
	`Error: Input variable not used in outputSchema: admin-status`
	`File: /Users/peter/polymer-tutorials/tutorials.mdl`
	`Line: 13`
	
	polymer will always try to provide a clear indication of what has gone wrong.
	
	The line number indicates the first line of the `Object` in which the error occurred.
	
	In this case it's telling us that we're not using admin-status in the `outputSchema`, which is not surprising since we haven't defined it yet.
	
	So, let's do that.

Step 2	Insert the following for the outputSchema:
	
	`outputSchema {`
	`"admin": {`
	`"overall-status": "${admin-status}"`
	`},`
	`"network-element": {`
	`"mtu": "${min-mtu}",`
	`"max-delay": "${max-delay-msec}"`
	`}`
	`}`
	
	Save tutorials.mdl

Step 3	In a terminal window run:
	`polymer gen -modules tutorials -trace`
	
	You should see something like:
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`
	
	`Input: /Users/peter/polymer-tutorials/tutorials-structure/schemas/api-in/R1.0/api-in-tutorial01-input-1.0.json`
	`Output: /Users/peter/polymer-tutorials/tutorials-structure/schemas/api-out/R1.0/api-out-tutorial01-output-1.0.json`
	
	`Morpher fn:`
	`/Users/peter/polymer-tutorials/tutorials-structure/morphers/api-out/R1.0/api-out-tutorial01-output-1.0.groovy`

	`Class: Api_Out_Tutorial01_Output_1_0` This indicates that we've generated the input and output schemas as well as a Morpher.

There are some details here that we're going to skip for now. These include:

- file naming conventions and versioning which will be discussed in the Project Structure chapter
- generated Morphers which will be discussed in the Morphers chapter

3.5 Executing plastic tests

Now that we've got a valid plastic mapping generated, it's time to actually execute plastic and see if our mapping works.

As part of the `polymer var` run, we also generated a `PlasticTest` instance:

```
PlasticTest
name            tutorial01-test1
mapping         tutorial01
inputPayload    {
    "admin-status":   "UP",
    "max-delay-msec": "10",
    "min-mtu":        "1500"
  }
```

You renamed the test previously to ensure that its name started with the name of the mapping that it's meant to test. This is actually sanity checked by `polymer` so that you don't inadvertently create a test for the wrong mapping if you're doing a cut/paste job.

It can also be a useful practice to define tests for failure situations.

Let's do that now.

Step 1	Open your tutorials.mdl file for editing.

Step 2	Add another test that exercises a failure scenario:
	
	`PlasticTest`
	`name tutorial01-test2`
	`mapping tutorial01`
	`inputPayload {`
	`"admin-status": "UP",`
	`"min-mtu": "1500"`
	`}`
	`description This should cause a failure because we're missing max-delay-mtu.`
	
	Make sure you have a blank line between your two `PlasticTest` objects!

Step 3	Save the file.
Step 4	In a terminal window run: `polymer test -modules tutorials`

You should see something like:

```
Reading: /Users/peter/polymer-tutorials/tutorials.mdl

----------------------------------------

Execute tutorial01-test1

Payload:
{
  "admin-status":   "UP",
  "max-delay-msec": "10",
  "min-mtu":        "1500"
}

Plastic result:
{
    "admin": {
        "overall-status": "UP"
    },
    "network-element": {
        "max-delay": "10",
        "mtu": "1500"
    }
}


----------------------------------------

Execute tutorial01-test2

Payload:
{
  "admin-status": "UP",
  "min-mtu":      "1500"
}

Error: PLASTIC-MISSING-IN: For (in-> [api-in-tutorial01-input/1.0/json]) (out->
[api-out-tutorial01-output/1.0/json]), the following input variables were not found on
the incoming payload: [max-delay-msec]
```

As you can see, the first test was successful and you'll see the values for the variables provided in the Payload, injected into your outputSchema.

However, the second test resulted in an error because of the missing `max-delay-msec` value.

We've just executed all tests, in the next tutorial, we'll see how you can be more selective.

3.6 Run selected PlasticTests

As the number of `PlasticMappings` increases, simply running all tests as you're working on new mappings would get very tedious.

If you want to run a single `PlasticTest` you can specify the name of the test via the -run option as follows:

```
polymer test -modules tutorials -run tutorial01-test1
```

By default, any test can be run by specifying its name to the -run option.

The `-run` option allows multiple values, so you can execute several specific tests if you like.

3.6.1 Tagging your tests

You can also specify tags on your `PlasticTests` so that you can create groups of tests to be executed.

Here's an example of adding some tags - you can add as many as you need:

```
PlasticTest
name            tutorial01-test1
mapping         tutorial01
inputPayload    {
    "admin-status":   "UP",
    "max-delay-msec": "10",
    "min-mtu":        "1500"
  }
tags tutorial

PlasticTest
name            tutorial01-test2
mapping         tutorial01
inputPayload    {
    "admin-status":   "UP",
    "min-mtu":        "1500"
  }
tags fail-test
tags tutorial
description This should cause a failure because we're missing max-delay-mtu.
```

To execute only those tests with a given tag, you would simply run:

```
polymer test -modules tutorials -run fail-test
```

You may specify multiple tags for the run option:

```
polymer test -modules tutorials -run tag1 tag2... tagN
```

3.6.2 Skipping tests

In addition to running particular tests, you can also exclude or skip tests based on their tags.

If you have tagged all of your failing tests with `fail-test`, you could run just the success tests by specifying:

```
polymer test -modules tutorials -skip fail-test
```

3.6.3 Summarizing the test run

If you're not interested in the details of your tests and just want a pass/fail summary, you can execute:

```
polymer test -modules tutorials -summary
```

And you'll see something like:

```
Reading: /Users/peter/polymer-tutorials/tutorials.mdl
tutorial01-test1 - passed
tutorial01-test2 - failed - PLASTIC-MISSING-IN: For (in->
[api-in-tutorial01-input/1.0/json]) (out-> [api-out-tutorial01-output/1.0/json]), the
following input variables were not found on the incoming payload: [max-delay-msec]
tutorial02-test1 - passed
tutorial02-test2 - passed
```

3.7 Optional variables

Back when we executed `tutorial01-test2` we got an error because of the missing `max-delay-msec` value.

Let's fix that situation by making `max-delay-msec` optional.

Step 1	Open your tutorials.mdl file for editing.

Step 2	Add the following mapping:
	
	`PlasticMapping`
	`name tutorial02`
	`folderStructure tutorials-structure`
	`variables admin-status example="UP" default="UP" note="Add notes here"`
	`variables max-delay-msec example="10" optional=true`
	`variables min-mtu example="1500"`
	`inputSchema {`
	`"admin-status": "${admin-status}",`
	`"max-delay-msec": "${max-delay-msec}",`
	`"min-mtu": "${min-mtu}"`
	`}`
	`outputSchema {`
	`"admin": {`
	`"overall-status": "${admin-status}"`
	`},`
	`"network-element": {`
	`"mtu": "${min-mtu}",`
	`"max-delay": "${max-delay-msec}"`
	`}`
	`}`
	
	This is simply a copy of our first mapping, but we've added the `optional` flag to the `max-delay-msec` attribute.

Step 3	Add the following test:
	
	`PlasticTest`
	`name tutorial02-test1`
	`mapping tutorial02`
	`inputPayload {`
	`"admin-status": "UP",`
	`"min-mtu": "1500"`
	`}`
	`description No problem with this input, now that max-delay-msec is optional.`

Step 4	In a terminal window run:

	`polymer gen -modules tutorials` This will generate the Morpher that actually implements the code to handle optional variables.

Step 5	In a terminal window run:
	`polymer test -modules tutorials -run tutorial02-test1`
	
	You should see something like:
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`
	
	`----------------------------------------`
	
	`Execute tutorial02-test1`
	
	`Payload:`
	`{`
	`  "admin-status": "UP",`
	`  "min-mtu":      "1500"`
	`}`
	
	`Plastic result:`
	`{`
	`    "admin": {`
	`        "overall-status": "UP"`
	`    },`
	`    "network-element": {`
	`        "mtu": "1500"`
	`    }`
	`}`
	
	By making `max-delay-msec` optional, we've made the previous error disappear.
	
	This is actually facilitated by the Morpher that gets generated when you run `polymer gen`.
	
	We'll discuss this aspect in the Morphers chapter. For now though, you don't need to worry about it.

3.8 Variable defaults

Specifying defaults is another way to prevent problems when an incoming payload is missing a value.

NOTE: Although plastic allows you to specify defaults inline with your variable definitions, it is recommended that you specify them in the `variables` definitions.

Following this practice means that all information associated with the variable e.g. optionality, sanity checking (which we'll see shortly), splitting etc. is all specified in one location.

We already have an example of using a default in the `tutorial02` mapping:

```
PlasticMapping
name            tutorial02
folderStructure tutorials-structure
variables    admin-status   example="UP" default="UP" note="Add notes here"
variables    max-delay-msec example="10" optional=true
variables    min-mtu        example="1500"
. . .
```

Let's create another test for tutorial02 and leave out the admin-status value.

Step 1	Open your tutorials.mdl file for editing.

Step 2	Add the following test:
	
	`PlasticTest`
	`name tutorial02-test2`
	`mapping tutorial02`
	`inputPayload {`
	`"min-mtu": "1500"`
	`}`
	`description We should see admin-status get filled with its default value.`
	
	NOTE: since we're just adding a test, we don't need to regenerate; we only need to do that if we change a mapping.

Step 3	In a terminal window run:
	`polymer test -modules tutorials -run tutorial02-test2`
	
	You should see something like:
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`
	
	`----------`

```
Execute tutorial02-test2

Payload:
{
  "min-mtu": "1500"
}

Plastic result:
{
    "admin": {
        "overall-status": "UP"
    },
    "network-element": {
        "mtu": "1500"
    }
}
```

And we see that our default value has been inserted.

3.9 Variable validation with ValueSets

A common situation is to have variables that must conform to certain rules e.g. only certain values are allowed.

This is facilitated by creating a `ValueSet` and specifying a `validate` option on the variable in question.

Step 1	Open your tutorials.mdl file for editing.

Step 2	Add the following at the end of your file:
	
	`ValueSet`
	`name  statusEnum`
	`values UP`
	`values DOWN note="The resource is not operative."`
	`description The allowed values for the status variable.`

Step 3	Indicate that you want to validate the admin-status variable in tutorial02:
	
	`PlasticMapping`
	`name  tutorial02`
	`folderStructure tutorials-structure`
	`variables  admin-status  example="UP" default="UP" validate=statusEnum`
	`variables  max-delay-msec example="10" optional=true`
	`variables  min-mtu  example="1500"`
	`. . .`
	
	NOTE: Now that you've added validation to the attribute, `polymer gen` will sanity check your default value as part of the generation process.

Step 4	Now, add a new test that lets you check that the validator is working.
	
	`PlasticTest`
	`name  tutorial02-test3`
	`mapping  tutorial02`
	`inputPayload  {`
	`  "admin-status":  "SIDEWAYS",`
	`  "min-mtu":  "1500"`
	`}`
	`description The admin-status validation should fail.`

Step 5	In a terminal window run: `polymer gen -modules tutorials`

	This will generate the Morpher that actually implements the code to handle the validation.

Step 6	In a terminal window run:
	`polymer test -modules tutorials -run tutorial02-test3`
	
	You should see something like:
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`
	
	`----------------------------------------`
	
	`Execute tutorial02-test3`
	
	`Payload:`
	`{`
	`  "admin-status": "SIDEWAYS",`
	`  "min-mtu":      "1500"`
	`}`
	
	`0    [main] ERROR ExtendedBasicMorpher  - Invalid value: SIDEWAYS for variable: admin-status - Allowed values: UP, DOWN`
	`Error: PLASTIC-ABORT-EX: A Plastic component (morpher or classifier) invoked abort: Invalid value: SIDEWAYS for variable: admin-status - Allowed values: UP, DOWN`
	
	The error will indicate the invalid input as well as the allowed values.

3.10 Create a PlasticPattern

In many situations, you will need to ensure that a variable conforms to some known pattern and take action based on whether or not the data matches.

In `polymer`, this mechanism is provided via the `PlasticPattern` specification that makes use of Java-based regular expressions. If you are not familiar with Java's regular expression syntax, there are many good tutorials that can lead you through the basics:

- Vogella - Regular expressions in Java - Tutorial
- The Java Tutorials - Regular Expressions
- W3 Schools - Java Regular Expressions

With all of these tutorials, you need only pay attention to the actual specification of the regular expression, not the Java code that implements the matching - that's taken care of by `PlasticPattern`.

We're going to create a pattern that emulates the behavior of the built-in plastic matching that's described in Chapter 8 - Pattern Matching of the plastic tutorial.

You might ask "Why bother to provide support for something that's already in plastic?"

The answer is that the `PlasticPattern` provides:

- a reusable definition of a pattern
- validation of values
- splitting of variables
- normalization of variables
- complete regex support (as opposed to very limited support for "wildcarding" in plastic)
- a mechanism to test your patterns separately from your mappings

In the plastic tutorial, the basic concept is that we have a variable value of the form:

```
one two/three four
```

and we want to split this into two new variables:

variable abc: `one two`
variable def: `three four`

Let's do that now.

Step 1	Open your tutorials.mdl file for editing.

Step 2	Add the following at the end of your file:
	
	`PlasticPattern`
	`name slashSeparated`
	`pattern ([^/]+)/([^/]+)$`
	`group 1 abc note="The part before the slash"`
	`group 2 def note="The part after the slash"`
	`description Implements the pattern matching used in Chapter 7 of the plastic tutorial.`
	
	The PlasticPattern makes use of the regex grouping concept i.e. the parts defined within the round brackets **( )** is a group. Groups are numbered sequentially and are given names that can be used as plastic variables (more on this in the Splitting variables with PlasticPatterns section).
	
	The highlighted **green** parts align to variable `abc` and the highlighted **blue** parts to variable `def`.

Step 3	Now, we're going to add a new, very simple pattern that will make use of the pattern to validate the input being used to populate the mapping.
	
	Add the following mapping:
	
	`PlasticMapping`
	`name tutorial03`
	`folderStructure tutorials-structure`
	`variables var1 validate=slashSeparated`
	`inputSchema {`
	`"something": "${var1}"`
	`}`
	`outputSchema {`
	`"somethingelse": "${var1}"`
	`}`
	`description This will exercise the use of a PlasticPattern for validation of the content of var1.`

Step 4	Now, we'll add a couple of tests:
	
	`PlasticTest`
	`name tutorial03-test1`
	`mapping tutorial03`
	`tags t3`
	`inputPayload {`
	`"something": "one two/three four",`
	`}`
	`description This should be fine, we're following the pattern.`
	
	`PlasticTest`
	`name tutorial03-test2`
	`mapping tutorial03`
	`tags t3`

	`inputPayload {`
	`"something": "one two three four",`
	`}`
	`description This should fail, we're missing the slash.`
	
	Notice that we've tagged out tests with `t3` so that we run them together.

Step 5	In a terminal window run: `polymer gen -modules tutorials` This will generate the Morpher that actually implements the code to handle the validation using the pattern.

Step 6	Now, execute the tests:
	`polymer test -modules tutorials -run t3`
	
	You should see something like:
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`
	
	`----------`
	
	`Execute tutorial03-test1`
	
	`Payload:`
	`{`
	`"something": "one two/three four"`
	`}`
	
	`Plastic result:`
	`{`
	`"somethingelse": "one two/three four"`
	`}`
	
	`----------`
	
	`Execute tutorial03-test2`
	
	`Payload:`
	`{`
	`"something": "one two three four"`
	`}`
	
	`0 [main] ERROR ExtendedBasicMorpher - Value: "one two three four" does not match pattern: "([^/]+)/([^/]+)$" - for variable: var1`
	`Error: PLASTIC-ABORT-EX: A Plastic component (morpher or classifier) invoked abort: Value: "one two three four" does not match pattern: "([^/]+)/([^/]+)$" - for variable: var1`

3.11 Testing your PlasticPatterns

Regular expressions are somewhat arcane and often cause the most experienced coder to turn to Google for assistance.

There is often a sequence of:

- specify a regular expression
- write a little program to test it
- test it
- not get the match you were expecting
- curse
- stare at the expression
- curse again
- alter the expression
- test again… and repeat until you get it right

- move on with your life

A small portion of this cycle is taken care of by creating `PatternTests` and using `polymer pattern`.

Here's the approach:

Step 1	Open your tutorials.mdl file for editing.

Step 2	Add the following at the end of your file:
	
	`PatternTest`
	`name slashSeparated-test1`
	`usePattern slashSeparated`
	`input "one two/three four"`
	`expectThat part1 "one two"`
	`expectThat part2 "three four"`
	`description Because we've indicated expectThat, we expect the match to be successful.`
	
	`PatternTest`
	`name slashSeparated-test2`
	`usePattern slashSeparated`
	`input "one two three four"`
	`description Because we haven't indicated expectThat, we expect the match to fail.`
	
	We've defined two tests, one that should pass and another that should fail.

Step 3	In a terminal window run:
	`polymer pattern -modules tutorials`
	
	You should see something like:
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`
	
	`slashSeparated-test1 SUCCEEDED - MATCHED`
	`slashSeparated-test2 SUCCEEDED - NO MATCH EXPECTED`
	
	Just like with PlasticTests, you can run a single test by specifying its name with the -run option:
	`polymer pattern -modules tutorials -run slashSeparated-test1`
	
	So that, if you were developing a new pattern, you could quickly test it as you made changes in your .mdl file.
	
	I haven't added the -curse flag as yet, but I can if necessary.

3.12 Splitting variables with PlasticPatterns

The initial demonstration of using wildcarding in plastic was to facilitate the creation of multiple variables from a single piece of input.

The same functionality is supported in polymer, with a slight variation in approach.

Step 1	Open your tutorials.mdl file for editing.

Step 2	Now, we're going to add a new pattern (and test) that makes use of variable splitting..
	
	Add the following mapping:
	
	`PlasticMapping`
	`name tutorial04`
	`folderStructure tutorials-structure`
	`variables var1 split=slashSeparated`
	`inputSchema {`
	`"something": "${var1}"`
	`}`
	`outputSchema {`
	`"firstPart": "${abc}",`
	`"secondPart": "${def}"`
	`}`
	`description This will exercise the use of a PlasticPattern for validation of the content of var1.`
	
	`PlasticTest`
	`name tutorial04-test1`
	`mapping tutorial04`
	`inputPayload {`
	`"something": "one two/three four",`
	`}`
	`description This should be fine, we're following the pattern.`
	
	The **orange** highlight indicates that we want to split the `var1` variable with slashSeparated pattern.
	
	The **blue** highlights indicate the names of the groups we defined in the pattern.

Step 3	In a terminal window run:
	`polymer gen -modules tutorials`
	
	This will…. darn it!
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`

	Error: You must create a variable called: abc to represent the value created by splitting variable: var1 File: /Users/peter/polymer-tutorials/tutorials.mdl Line: 165 When you split a variable, you must add variables for each group into which it gets split.

Step 4	Add these variables to the `tutorial04` pattern:
	
	`variables abc note="Created by splitting"`
	`variables def note="Created by splitting"`
	
	Save the file.
	And run:
	`polymer gen -modules tutorials`

Step 5	In a terminal window run:
	`polymer test -modules tutorials -run tutorial04-test1`
	
	You should see something like:
	
	`Reading: /Users/peter/polymer-tutorials/tutorials.mdl`
	
	`----------------------------------------`
	
	`Execute tutorial04-test1`
	
	`Payload:`
	`{`
	`  "something": "one two/three four"`
	`}`
	
	`Plastic result:`
	`{`
	`    "firstPart": "one two",`
	`    "secondPart": "three four"`
	`}`
	
	We've successfully split our initial input and placed the resulting values in different locations.

3.13 Using PlasticPatterns for normalizing values

Coming soon!

3.14 Turning on Morpher tracing

Coming soon.

4 Project Structure

Coming soon!

5 Morphers

Coming soon!

6 Patterns

Coming soon!

7 Mapping Definition Language

7.1 Description Conventions

The names of classes of objects, their attributes or command line specifications are displayed in `Courier New` font.

Mandatory attributes/fields are displayed in green.

Optional attributes/fields are displayed in blue.

7.1.1 Multi-valued Attributes

Multi-valued attributes/fields are flagged with an asterisk * - multiple attribute values are represented by repeating the attribute name at the start of a line multiple times. For example:

```
attributex value1
attributex value2
```

7.1.2 Complex Attributes

Complex attributes are composed of multiple fields, generally separated by spaces e.g.
`attribute field-1 field-2 field-3`

Complex attributes may also have optional fields, indicated by specifying the field name followed by '=' and the value - e.g.
`attribute field-1 optional-1=value optional-2=value`

Sometimes, field values needed to be quoted, in that case, the attribute name will be documented inside quotes.

The `.mdl` configuration files allow for lines to be commented out using the Java convention of starting the line with double slashes - `//`

7.2 MDL Classes

7.2.1 MdlModule

Coming soon!

7.2.2 FolderStructure

Coming soon!

7.2.3 PlasticMapping

Coming soon!

7.2.4 PlasticTest

Coming soon!

7.2.5 ValueSet

Coming soon!

7.2.6 PlasticPattern

Coming soon!

7.2.7 PatternTest

Coming soon!

8 The polymer tool

8.1 Utility - guide

8.2 Utility - new

8.3 Utility - var

8.4 Utility - gen

8.5 Utility - test

8.6 Utility - pattern

8.7 Utility - yang

RFC7950 YANG 1.1

The yang utility provides a variety of useful functionality associated with using Yang as the basis for mappings:

- generation of example JSON structures from Yang
- generation of variabilized input schemas
- generation of variable definitions that include descriptions (notes), default values, type information and unit information.