15. Click **Publish New Version**. On the **Publish New Version** page, click **Save**. This publishes the content view with our module.

16. Scroll to the new version of our view and click **Promote**. Choose a life-cycle environment and click **Promote Version**. This makes the view a part of the chosen life-cycle environment.

Our content view is now published. As a part of the content view creation, Red Hat Satellite 6 creates a new Puppet environment for use in the provisioning process. This Puppet environment contains our module. You can view this new Puppet environment on the **Configure** > **Environments** page.

3.7. CONFIGURING SMART CLASS PARAMETERS FROM PUPPET CLASSES

Some module classes contain complex parameters, called Smart Class Parameters, that define a value for a key just as a simple parameter does, but allow conditional arguments, validation, and overrides for specific object types. Satellite 6 has the ability to import these parameterized classes and allow modification of such smart parameters. For more information on the types of parameters in Puppet, see the Parameters section of the Red Hat Satellite 6.2 Host Configuration Guide.

For example, **mymodule** contains a parameter for the HTTP port of our web server. This parameter, **`httpd_port`**, is set to a default of 8120. However, a situation might occur where we need to use a different port for a provisioned system. Satellite 6 can override the **`httpd_port`** parameter during configuration. This provides an easy way to change the HTTP port on our web server.

This procedure requires the **mymodule** module to be uploaded to a Product and added to a Content View. This is because we need to edit the classes in the resulting Puppet environment. Alternatively, you can download and install another module as described in Chapter 3, *Adding Puppet Modules to Red Hat Satellite 6*.

1. Navigate to **Configure** > **Smart class parameters**.

2. A table appears listing all Smart Class Parameters from the classes in your Puppet modules. Enter **`httpd_port`** in the search field. Click on the **`httpd_port`** parameter.

3. The options for the Smart Class Parameter appears. To allow overriding this parameter during provisioning, select the **Override** option.

4. Selecting the **Override** option allows us to change the **Key type** and **Default value**. This is useful if we aim to globally change this value for all future configurations.

   The following key types are available:

   String

   The value is interpreted as a plain text string. For example, if your Smart Class Parameter sets the host name, the value is interpreted as a string:

   ```
   myhost.example.com
   ```

   Boolean

   The value is interpreted and validated as a true or false value. Examples include:

```
True
true
1
```

Integer

The value is interpreted and validated as an integer value. Examples include:

```
8120
-8120
```

Real

The value is interpreted and validated as a real number value. Examples include:

```
8120
-8120
8.12
```

Array

The value is interpreted and validated as a JSON or YAML array. For example:

```
["Monday","Tuesday","Wednesday","Thursday","Friday"]
```

Hash

The value is interpreted and validated as a JSON or YAML hash map. For example:

```
{"Weekdays":
["Monday","Tuesday","Wednesday","Thursday","Friday"],
"Weekend": ["Saturday","Sunday"]}
```

YAML

The value is interpreted and validated as a YAML file. For example:

```
email:
  delivery_method: smtp
  smtp_settings:
    address: smtp.example.com
    port: 25
    domain: example.com
    authentication: none
```

JSON

The value is interpreted and validated as a JSON file. For example:

```
{
  "email":[
    {
      "delivery_method": "smtp"
      "smtp_settings": [
        {
          "address": "smtp.example.com",
            "port": 25,
            "domain": "example.com",
            "authentication": "none"
          }
        ]
      }
    ]
  }
```

For this example, leave the default as 8120.

5. Selecting the **Override** option also exposes **Optional Input Validator**, which provides validation for the overridden value. For example, we can include a regular expression to make sure **`httpd_port`** is a numerical value. For our example, leave this section blank.

6. Selecting the **Override** option also exposes **Prioritize attribute order**, which defines a hierarchical order of system facts, and **Specify matchers**. The matcher-value combinations determine the right parameter to use depending on an evaluation of the system facts. For our example, leave these sections with the default settings.

7. Click **Submit**.

We now have an override value for the Smart Class Parameter **`httpd_port`**.

3.8. USING THE SMART VARIABLE TOOL

Smart Variables are a tool to provide global parameters to the Puppet Master for use with a Puppet Classes that do not contain Smart Class Parameters. The same Smart Matcher rules are used for both Smart Variables and Smart Class Parameters.

Note

The Smart Variables tool was introduced as an interim measure before Puppet modules supported Smart Class Parameters. If in doubt, use Smart Class Parameters as explained in the previous section.

Before Smart Class Parameters were introduced, users who wanted to override a parameter where asked to rewrite their Puppet code to use a global parameter. For example:

```
class example1 {
  file { '/tmp/foo': content => $global_var }
}
```

For the above example, **`$global_var`** is set in the Smart Variables section of the web UI and the value is associated with the "example1" class. Although it is recommend to precede global variables with **`::`** to restrict Puppet to search the global scope, their absence does not mean a variable is not a global variable.

With the introduction of Smart Class Parameters, the follow form could be used:

```
class example2($var="default") {
  file { '/tmp/foo': content => $var }
}
```

For the above example, **`$var`** is set in the Smart Class Parameters section of the web UI and the value is associated with the "example2" class. If you see a variable defined in the class header, as in the above **`class example2($var="default")`**, then you can be sure that **`$var`** is a class parameter and you should use the Smart Class Parameter function to override the variable.

As Smart Variables require custom-designed modules using global-namespace parameters, rather than standard modules from the Puppet community, and the result is the same, text placed in '/tmp/foo' in the examples above, there is no longer a reason to use Smart Variables except to support legacy modules.

Although Smart Variables are global variables, they are associated with a Puppet class and will only be sent to a host that has that specific Puppet Class assigned in Satellite. You can create a Smart Variable with any name, no validation is done in Satellite, but unless the associated Puppet module applied has a matching variable in its code, the Smart Variable will not be used.

Satellite adds the variable you create in Satellite to the Host YAML file. This file can be viewed in the web UI by navigating to **Hosts** > **All Hosts**, selecting the name of the host, and then click on the **YAML** button. The Satellite sends the Host YAML file to the *external node classifier* (ENC), a function included with the Puppet Master. When the Puppet Master queries the ENC about a host, the ENC returns a YAML document describing the state of the host. This YAML document is based on data taken from Puppet manifests, but is subject to Smart Class Parameter overrides and any Smart Variables.

Applying a Smart Variable to a Host

As Smart Variables should only be used to support your custom Puppet modules previously modified to include a global parameter, the following example uses a simple example called **anothermodule**. The **anothermodule** Puppet module manifest is as follows:

```
class anothermodule {
   file { '/tmp/motd':
     ensure  => file,
     content => $::content_for_motd,
  }
}
```

This example will supply a value for the **`$::content_for_motd`** parameter.

1. In the web UI, navigate to **Configure** > **Classes**
2. Select the name of the Puppet Class from the list.
3. Click the **Smart Variables** tab. This displays a new screen. The left section contains a list of previously created parameters, if any. The right section contains the configuration options. Click **Add Variable** to add a new parameter.
4. Enter the parameter in the **Key** field. In this example, **`content_for_motd`**.
5. Edit the **Description** text box, for example **`Testing /tmp motd Text`**.
6. Select the **Key type** of data to pass. Select **string**.
7. Type a **Default Value** for the parameter. For example, **`No Unauthorized Use`**.
8. Use the **Optional Input Validator** section to restrict the allowed values for the parameter. Choose a **Validator type** (either a **list** of comma separated values or a regular expression, **regexp**) and input the allowed values or regular expression code in the **Validator rule** field.

9. Use the **Prioritize attribute order** section to set the order of precedence in which the host attributes or Facts are to be evaluated against the matchers (configured below). You can rearrange the entries in the list and add to the default list. To create a logical AND condition between matchers, arrange the names of the matchers on one line as a comma separated list.

10. In the **Specify matchers** section, click **Add Matcher** to add a conditional argument. The attributes to match against should correspond to the entries in the **Order** list above. If no matcher is configured, then only the default value can be used.

    For example, if the desired value of the parameter is **`This is for Server1`** for any host with a fully qualified domain name of **`server1.example.com`**, then specify the **Match** as **`fqdn=server1.example.com`** and the **Value** as **`This is for Server1`**.

11. Click **Submit** to save your changes.