Oracle® Cloud

Developing Custom Policies for Oracle API Platform Cloud Service - Classic

E96998-02
January 2019

Oracle Cloud Developing Custom Policies for Oracle API Platform Cloud Service - Classic,

E96998-02

Primary Author: Oracle Corporation

Contents

1

Understand Custom Policy Development

Review the following topics for an overview of policy SDK, components of policy and why develop a custom policy.

Topics:

About Custom Policies

When built-in policies that are available do not cover your use case, Policy SDK helps you build your own custom policies.

API execution is based on a policy-based execution flow, which is also referred to as policy pipeline. Policies are applied to an API definition to secure, throttle, route, or log requests sent to your API. Depending on the policies applied, requests can be rejected when they do not meet criteria you specify when configuring each policy. Oracle provides predefined built-in policies for Security, Traffic Management, Interface Management, Routing, and so on.

However, the set of built-in policies that are available might not cover all your use cases. Use Policy SDK to build your own custom policies to address the use cases that built-in policies do not address.

For more information, see Understanding Policies in *Using Oracle API Platform Cloud Service* guide.

Understand Policy SDK

Policy SDK is used by customers, partners, and Oracle Development to deliver policies and also to create custom policies for specific use cases.

Policy runtime classes depend on policy SDK jar to interact with requests and response. Use Policy SDK when you want to abstract the technical details of a use case, or when you want to capture a repeatable task.

Policy SDK contains,

- ApiRuntimeContext layer which manipulates the API request and response.
- Base classes for policy life cycle management.
- ApipAction which is the shell for Apip custom actions.
- Util classes

Policy SDK jar is deployed on both management tier and gateway.

Contents of a Policy

A policy contains design-time components and runtime features that control the policy usage semantics.

Metadata JSON document contains the design-time components and runtime execution is covered by respective policy files.

Design-time components:

- **Descriptive Attributes:** Provides generic metadata to drive the main UI
- **UI:** Contains web resources to render the provider-specific UI.
- **Configuration:** Contains provider-specific validation feature and a few other design-time features.

Runtime execution occurs when a request for an API that is deployed to the gateway reaches the point in the execution flow.

2

Understand Structure and Execution of a Policy

The topics help you develop design-time and runtime components, and author a policy.

Topics:

Policy Metadata

Policy metadata contains both descriptive attributes and also dynamic UI and other configuration components.

The metadata:

- Provides a default configuration.
- Provides a fixed mandatory configuration.
- Provides cloning information.
- Indicates whether the policy is required for the given tenant.

General Information

Metadata contains general information that is descriptive in nature and that is manually modified.

Metadata contains the following general information (descriptive attributes) and the fields always start with lowercase:

- **Type:** unique type. OOTB policies start with the prefix ***o:***
- **Name:** can be the actual name or a key within the l10n file (l10n string)
- **Version:** version of the policy
- **Revision:** revision number of the policy

> **Note:**
>
> For information on version and revision number, see Change the Version and Revision Number in Metadata

- **schemaVersion**: The schema of the JSON for policy configuration can change resulting in multiple versions over a period of time. **schemaVersion**, helps you know which version of the JSON the policy is configured with.
- **sdkVersion**: version of sdk
- **category:** category name for the policy. The category can either be an ID to an OOTB category, or an l10n string. To have pointer to the OOTB category IDs, prefix the ID with an @ symbol. To use a category by name, provide the string without prefixing the @ symbol. The following are the IDs and their values:
  - @implementations.policyCategory.endpoints = “EndPoints”
  - @implementations.policyCategory.routing = “Routing”
  - @implementations.policyCategory.trafficMgmt = “Traffic Management”
  - @implementations.policyCategory.interfaceMgmt = “Interface Management”
  - @implementations.policyCategory.other = “Other”
  - "@implementations.policyCategory.security='Security'". Custom policy cannot be added to this category.
- **description:** can be the actual description or a key within the l10n file (l10n string)
- **constraints:** Policies can be inserted at different points in the execution flow, with some possible restrictions. The metadata also captures constraints for the policy:
  - ***direction:*** represents where a policy can be inserted. One of the four possible values for direction are: REQUEST, RESPONSE, REQUEST_OR_RESPONSE, REQUEST_AND_RESPONSE
  - **requestZone / responseZone:** If the policy direction is "REQUEST", provide requestZone and for "RESPONSE" provide responseZone. Provide both if the direction is either REQUEST_OR_RESPONSE / REQUEST_AND_RESPONSE
  - **singleton:** represents whether the policy is allowed only once, true by default.
  - **required:** the policy is required to be present
  - **requires:** a policy might require other policies to be inserted before or in the other pipeline
  - **before:** a policy might need to be inserted before other policies. For example, the *API Request* policy must be *before* the *Service Request* policy
  - **after:** a policy might need to be inserted after other policies. For example, the *Service Request* policy must be *after* the *API Request* policy.

> **Note:**
>
> *Zone* signifies the order of execution during runtime. You cannot place a policy with zone 30 after a policy with zone greater than 30. You can place policies only in specific positions in your API implementations. For more information on where and in what order polices can be placed in the request and response flows for your APIs, see Policy Placement in *Using Oracle API Platform Cloud Service* guide.

The general information in metadata looks like:

```
{
    "type" : "o:ApiRequest",
    "name" : "policy.name",
    "version" : "1.0",
    "revision" : "4",
    "schemaVersion" : "1",
    "sdkVersion" : "3.0",
    "category" : "@implementations.policyCategory.endpoints",
    "description" : "policy.description",
    "ocsgActionName" : "ApiRequest",
    "constraints" : {
        "direction" : "REQUEST",
        "requestZone" : "5",
        "singleton" : true,
        "required" : true,
        "requires" : [ "o:ApiResponse/1.0"],
        "before" : [ "o:ServiceRequest/1.0" ]
    }
}
```

> **Note:**
>
> The attributes `policy.name` and `policy.description` refer to a key in the file `L10n/header.js`

UI

The UI contains web resource files of different content types that the REST layer must set when returning the resource. API Platform supports a default mapping for HTML, JavaScript and JSON files.

The `ui` element in metadata.json file contains the file names that render the policy and a Javascript file feeds the UI. When creating a metadata.json file for your custom policy follow these guidelines covering all aspect of the policy.

The UI is dynamically loaded and it consists of the following components:

- **edit**: rendered for edit mode in the implementation tab.
- **view**: rendered for view-only mode, for example as seen in a deployment.

Metadata for each UI component contains three attributes *html*, *js* and *l10nbundle* which are relative paths for the root HTML, JavaScript and l10 bundles.

The UI part of the policy metadata looks like:

```
   "ui": {
      "edit": {
         "html": "custompolicy-edit.html",
         "js": "custompolicy-edit.js",
         "helpInfo": "#helpInfo",
         "helpUrl": "http://www.oracle.com",
         "helpTopicId": "policies.custompolicy"
      },
      "view": {
         "html": "custompolicy-view.html",
         "js": "custompolicy-view.js",
         "helpInfo": "#helpInfo",
         "helpUrl": "http://www.oracle.com",
         "helpTopicId": "policies.custompolicy"
      },
      "l10nbundle": "L10n/custompolicy.json"
   },
```

The attribute *helpInfo* refers to a string to be displayed as is. Presence of the HASH (#) sign in the beginning of the value of “helpInfo” as in `#apiReqHelpInfo` indicates that the string is to be referenced from the l10nbundle file `request.json`.

Sample contents for {policy name}-edit.js

Use the sample files as a model while creating HTML and JS files for your policy.

```
function PolicyConfigurationModel(ko, $, oj, config, additionalParams) {
    self = this;
    self.config = config;
    self.conditions = ko.observableArray();
    self.l10n = ko.observable(additionalParams.l10nbundle);
    self.disableApplyPolicyButton =
additionalParams.disableApplyPolicyButton;

    self.opList   = ko.observableArray([
                        {"value" : "=",  "label" : "="},
                        {"value" : "!=", "label" : "!="},
                        {"value" : ">",  "label" : ">"},
                        {"value" : "<",  "label" : "<"},
                        {"value" : ">=", "label" : ">="},
                        {"value" : "<=", "label" : "<="},
                        {"value" : "null",  "label" : self.l10n()
['label.isnull']},
                        {"value" : "not null", "label" : self.l10n()
['label.isnotnull']}
                    ]);

    self.actions = ko.observableArray([
                            {"value" : "PASS", "label" : self.l10n()
['label.pass']},
```

```
                            {"value" : "REJECT", "label" : self.l10n()
['label.reject']}
                        ]);

    self.conjunctions = ko.observableArray([
                            {"value" : "ANY", "label" : self.l10n()
['label.any']},
                            {"value" : "ALL", "label" : self.l10n()
['label.all']}
                        ]);

    self.selectedAction = ko.observableArray(["REJECT"]);
    self.selectedConjunction = ko.observableArray(["ANY"]);

    self.initialize = function() {
        if (self.config) {
            for (var i = 0; i < self.config.conditions.length; ++i) {
                var condition = {};

                if (self.isUnary(self.config.conditions[i].operator)) {
                    condition = {
                        "headerName": self.config.conditions[i].headerName,
                        "operator":
ko.observableArray([self.config.conditions[i].operator]),
                        "showValueCol": ko.observable(false)
                    };
                } else {
                    condition = {
                        "headerName": self.config.conditions[i].headerName,
                        "operator":
ko.observableArray([self.config.conditions[i].operator]),
                        "headerValue":
self.config.conditions[i].headerValue,
                        "showValueCol": ko.observable(true)
                    };
                }
                self.conditions.push(condition);
            }
        }

        if (self.config && self.config.action) {
            self.selectedAction([self.config.action]);
        }

        if (self.config && self.config.conjunction) {
            self.selectedConjunction([self.config.conjunction]);
        }

        // we need at least one header to start with
        if (self.conditions().length === 0) {
            self.addCondition();
        } else {
            self.disableApplyButton(false);
        }
```

```
        self.populatePassRejectContainer();
    };

    self.addCondition = function() {
        self.conditions.push({"headerName" : "", "operator" :
ko.observableArray(["="]), "headerValue" : "", "showValueCol":
ko.observable(true)});
        self.disableApplyButton(false);
    };

    self.removeCondition = function(condition) {
        self.conditions.remove(condition);
        self.disableApplyButton(self.conditions().length === 0);
    };

    self.handleOperatorChange = function(index, event, data) {
        if (data.option !== 'value'){
            return;
        }

        if (self.isUnary(self.conditions()[index].operator()[0])) {
            self.conditions()[index].showValueCol(false);
        } else {
            self.conditions()[index].showValueCol(true);
        }
    };

    self.isUnary = function(operator) {
        return operator === "null" || operator === "not null";
    };

    self.populatePassRejectContainer = function() {
        var actionSelect = $('<select id=\'action\' data-
bind="ojComponent: { component: \'ojSelect\', options: actions, value:
selectedAction, rootAttributes: { style:\'max-width:50px;\'}}"></select>');
        var optionSelect = $('<select id=\'option\' data-
bind="ojComponent: { component: \'ojSelect\', options: conjunctions,
value: selectedConjunction, rootAttributes: { style:\'max-width:50px;
\'}}"></select>');
        var passRejectHtml = apiplatform.utils.substituteParams(self.l10n()
['label.passorreject'], [actionSelect[0].outerHTML,
optionSelect[0].outerHTML]);
        $("#pass-reject-container").append(passRejectHtml);
    };

    // Begin - Framework methods

    self.getPolicyConfiguration = function(){
        //self.config.operator = self.operator()[0];
        var config = {"action": "", "conjunction" : "", "conditions" : []};
        for (var i = 0; i < self.conditions().length; i++) {
            var condition = {};
            if (self.isUnary(self.conditions()[i].operator()[0])) {
                condition = {
                    "headerName": self.conditions()[i].headerName,
```

```
                    "operator"  : self.conditions()[i].operator()[0]
                };
            } else {
                condition = {
                    "headerName" : self.conditions()[i].headerName,
                    "operator"   : self.conditions()[i].operator()[0],
                    "headerValue": self.conditions()[i].headerValue
                };
            }
            config.conditions.push(condition);
        };
        config.action = self.selectedAction()[0];
        config.conjunction  = self.selectedConjunction()[0];
        return config;
    };

    self.disableApplyButton = function(flag) {
        self.disableApplyPolicyButton(flag);
    };

    // End - Framework methods

    self.initialize();

}
```

Sample Contents of {policy name}-edit.html

Top level must have the id `policy-ui-content`because the Policy UI code uses this div to bind the contents dynamically.

```
<div id="policy-ui-content" class="oj-form" >
    <div id="pass-reject-container">
    </div>

    <div class="conditions-container-div">
        <table
               data-bind="visible: conditions().length > 0 ">
                <col style="width:35%">
                <col style="width:15%">
                <col style="width:35%">
                <col style="width:15%">
            <tr>
                <th style="text-align: left;"><span data-bind="text: l10n()
['label.headername']"
                                                  class="condition-field-label
condition-first-or-last-column"></span></th>
                <th style="text-align: left;"><span data-bind="text: l10n()
['label.headeroperator']"
                                                  class="condition-field-label
condition-middle-column"></span></th>
                <th style="text-align: left;"><span data-bind="text: l10n()
['label.headervalue']"
                                                   class="condition-field-label
condition-first-or-last-column"></span></th>
```

```
                <th></th>
            </tr>
            <tbody data-bind="foreach: {data: conditions, as:
'condition'}">
                <tr class="condition-container-row">
                    <td class="condition-first-or-last-column">
                        <input type="text" required
                               data-bind="attr: {id: 'headerName_'
+ $index()}, ojComponent: {
                                    component: 'ojInputText', value:
condition.headerName,
                                    placeholder: $parent.l10n()
['placeholder.headername']}"/>
                    </td>
                    <td class="condition-middle-column">
                        <select data-bind="attr: {id: 'operator_'
+ $index()}, ojComponent: {
                                   component: 'ojSelect',
                                   options: $parent.opList,
                                   optionChange: function(data, event) {
                                             $parent.handleOperatorChange($i
ndex(), data, event);
                                        },
                                   value: condition.operator}">
                        </select>

                    </td>
                    <td class="condition-middle-column">
                        <div data-bind="visible: condition.showValueCol">
                            <input type="text" required
                                   data-bind="attr: {id: 'headerValue_'
+ $index()}, ojComponent: {
                                        component: 'ojInputText', value:
condition.headerValue,
                                        placeholder: $parent.l10n()
['placeholder.headervalue']}"/>
                        </div>
                        <div data-bind="visible: !
condition.showValueCol()">
                            <span data-bind="text: $parent.l10n()
['label.notApplicable']"></span>
                        </div>
                    </td>
                    <td class="condition-last-column">
                        <div>
                            <div style="float:left; vertical-align: top;
display: none;"
                                 class="policy-remove-item-icon-black-
withpadding"
                                 data-bind="attr: {id: 'btn_remove_'
+ $index()}, click: $parent.removeCondition,

visible: $parent.conditions().length > 1,
                                            attr: {title: $parent.l10n()
['tooltip.delcondition']}">
```

```
                            </div>
                            <div style="float:left; vertical-align: top;
display: none;"
                                 class="policy-add-item-icon-black"
                                 data-bind="attr: {id: 'btn_add_'
+ $index()}, click: $parent.addCondition,
                                                    visible: $index()
=== $parent.conditions().length - 1,
                                                    attr:
{title: $parent.l10n()['tooltip.addcondition']}">
                            </div>
                        </div>
                    </td>
                </tr>
                <tr>
                    <td colspan="3" class="policy-condition-separator"></
td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

Sample contents for {policy name}-view.js

```
function PolicyConfigurationModel(ko, $, oj, config, additionalParams){
    self = this;
    self.config = config;
    self.l10n = ko.observable(additionalParams.l10nbundle);
    self.conditions = ko.observableArray();
    self.selectedAction = ko.observable();
    self.selectedConjunction = ko.observable();

    self.operatorsMap   = [];

    self.initialize = function() {

        self.operatorsMap["="] = "=";
        self.operatorsMap["!="] = "!=";
        self.operatorsMap[">"] = ">";
        self.operatorsMap["<"] = "<";
        self.operatorsMap[">="] = ">=";
        self.operatorsMap["<="] = "<=";
        self.operatorsMap["null"] = self.l10n()['label.isnull'];
        self.operatorsMap["not null"] = self.l10n()['label.isnotnull'];

        if (self.config) {
            for (var i = 0; i < self.config.conditions.length; ++i) {
                var condition = {};

                if (self.isUnary(self.config.conditions[i].operator)) {
                    condition = {
                        "headerName": self.config.conditions[i].headerName,
                        "operator":
```

```
self.operatorsMap[self.config.conditions[i].operator],
                        "showValueCol": ko.observable(false)
                    };
                } else {
                    condition = {
                        "headerName": self.config.conditions[i].headerName,
                        "operator":
self.operatorsMap[self.config.conditions[i].operator],
                        "headerValue":
self.config.conditions[i].headerValue,
                        "showValueCol": ko.observable(true)
                    };
                }
                self.conditions.push(condition);
            }
        }

        if (self.config && self.config.action) {
            self.selectedAction(self.config.action);
        }

        if (self.config && self.config.conjunction) {
            self.selectedConjunction(self.config.conjunction);
        }

        self.populatePassRejectContainer();
    };

    self.isUnary = function(operator) {
        return operator === "null" || operator === "not null";
    };

    self.populatePassRejectContainer = function() {
        var actionHtml = $("<span id='action' data-bind='text:
selectedAction'
                        style='padding:0px 35px 0px 0px; color: #959595'></
span>");
        var conjunctionHtml = $("<span id='action' data-bind='text:
selectedConjunction'
                        style='padding:0px 35px 0px 35px; color:
#959595'></span>");
        var passRejectHtml = apiplatform.utils.substituteParams(self.l10n()
['label.passorreject'],
                         [actionHtml[0].outerHTML,
conjunctionHtml[0].outerHTML]);
        $("#pass-reject-container").append(passRejectHtml);
    };

    self.initialize();
}
```

Sample contents for {policy name}-view.html

```
<div id="policy-ui-content" class="oj-form" >
    <div id="pass-reject-container" class="condition-field-label-2">
    </div>

    <div class="conditions-container-div">
        <table style="width:100%; padding:0px;"
               data-bind="visible: conditions().length > 0 ">
                <col style="width:40%">
                <col style="width:20%">
                <col style="width:40%">

            <tbody data-bind="foreach: {data: conditions, as:
'condition'}">
                <tr data-bind="visible: $index() === 0">
                    <td class="condition-first-or-last-column">
                        <label data-bind="text: $parent.l10n()
['label.headername']" class="condition-field-label"></label>
                    </td>
                    <td class=" condition-middle-column">
                        <label data-bind="text: $parent.l10n()
['label.headeroperator']" class="condition-field-label"></label>
                    </td>
                    <td class="condition-first-or-last-column">
                        <label data-bind="text: $parent.l10n()
['label.headervalue']" class="condition-field-label"></label>
                    </td>
                </tr>

                <tr class="condition-container-row">
                    <td class="condition-first-or-last-column">

                        <label data-bind="text:condition.headerName"
class="policy-readonly-text"></label>
                    </td>
                    <td class="condition-middle-column">

                        <label data-bind="text:condition.operator"
class="policy-readonly-text"></label>

                    </td>
                    <td class="condition-first-or-last-column">
                        <div data-bind="visible: condition.showValueCol">

                            <label data-bind="text: condition.headerValue"
class="policy-readonly-text"></label>
                        </div>
                        <div data-bind="visible: !
condition.showValueCol()">
                            <span data-bind="text: $parent.l10n()
['label.notApplicable']" class="policy-readonly-text"></span>
                        </div>
                    </td>
```

```
                </tr>
                <tr>
                    <td colspan="3" class="policy-condition-separator"></
td>
                </tr>
            </tbody>
        </table>
    </div>

</div>
```

Configuration

Configuration component contains the design time semantics and validation feature for handling policies.

The SDK defines Java APIs for each feature that policy plugins must implement. Each API can cover multiple features. The configuration component in metadata is used to declare the implementation classes.

The configuration component of a policy metadata looks like:

```
   "configuration": {
      "services": [
         {
            "type":
"oracle.apiplatform.policies.sdk.validation.PolicyValidator",
            "service":
"oracle.apiplatform.policies.custompolicy.CUSTOMPOLICYValidator"
         },
         {
            "type":
"oracle.apiplatform.policies.sdk.runtime.PolicyRuntimeFactory",
            "service":
"oracle.apiplatform.policies.custompolicy.CUSTOMPOLICYRuntimeFactory"
         }
      ]
   }
```

The following services are supported:

- `oracle.apiplatform.policies.sdk.validation.PolicyValidator`: This service is used to validate the policy configuration that is submitted. It returns error messages if it finds syntax or semantic errors.
- `oracle.apiplatform.policies.sdk.runtime.PolicyRuntimeFactory`: This service is used to register a factory class to create policy runtime class. The policy runtime class executes the logic that is defined in the policy configuration.
- `oracle.apiplatform.policies.sdk.references.PolicyReferenceHandler`: This service enforces the policies that refer other resources declare those references to the system. This helps us track those references. For example, we can track and avoid deletion of a referenced resource.

- `oracle.apiplatform.policies.sdk.security.PolicySecurityHandler`: This service is used to encode or decode the sensitive information in the policy configuration. This ensures that we do not have sensitive information saved in clear text as part of the API.

Localization

You can modify a few parameters to localize your custom policy.

Make the localization related changes in the `ui/resources/L10n/{policyname}.json` file.

Example of `l10n` bundle file in English

```
{
    "l10nBundle" : {
        "policy.name"             : "CUSTOMPOLICY",
        "policy.description"      : "Used to test a custom policy.",
        "#helpInfo"               : "Complete all fields. Comments are
optional.",
        "label.field.header"      : "CUSTOMPOLICY",
        "label.condition.phrase"  : "Pass the request/response if code
runs to completion.

             Errors and rejections are handled as exceptions.",
        "label.headername"        : "Name",
        "label.headeroperator"    : "Operator",
        "label.headervalue"       : "Value",
        "label.passorreject"      : "{{0}} request if {{1}} of the
following header conditions are met.",
        "label.any"               : "Any",
        "label.all"               : "All",
        "label.pass"              : "Pass",
        "label.reject"            : "Reject",
        "label.header.conditions" : "Custom Policy Conditions",
        "label.notApplicable"     : "Not Applicable",
        "label.isnull"            : "Is Null",
        "label.isnotnull"         : "Is Not Null",
        "label.addcondition"      : "Add Condition",
        "tooltip.addcondition"    : "Add a new condition",
        "tooltip.delcondition"    : "Delete this condition",
        "placeholder.headername"  : "Enter a header name.",
        "placeholder.headervalue" : "Enter a header value.",
        "msg.noconditions"        : "No conditions to display"
    }
}
```

When you want to localize `l10n` bundle file for your locale, navigate to respective locale folderJapanese

1. Navigate to the `ui/resources/L10n/{locale folder}`.
2. Open the `{policyname}.json` file.
3. Localize the values for parameters on the right.

> **Note:**
>
> Do not modify the parameters on the left. For example, `policy.name`, `policy.description` and so on.

Policy Validation

Policy validation is split into two calls, one to perform syntax validation and one to perform semantics validation.

Syntax validation ensures that the data that is provided is syntactically correct. If the data is not syntactically valid, API creation or update to API fails

Semantic validation ensures that the data is semantically valid and runtime layer can execute it. You can create or update an API with semantically invalid configuration, but semantic validation failure results in deployment or execution errors.

The following is an example of syntax and semantic validation. validateSyntax() is used to catch JSON syntax errors and values of wrong types. validateSemantics() validates if the values are the ones that are allowed.

```
package oracle.apiplatform.policies.%%policyname%%;

import java.util.List;

import org.json.JSONObject;

import oracle.apiplatform.common.l10n.NonLocalizedText;
import oracle.apiplatform.common.l10n.Text;
import oracle.apiplatform.policies.sdk.validation.AbstractPolicyValidator;
import oracle.apiplatform.policies.sdk.validation.Diagnostic;
import oracle.apiplatform.policies.sdk.validation.Diagnostic.Severity;

public class %%POLICYNAME%%Validator extends AbstractPolicyValidator
implements %%POLICYNAME%%Constants {

    @Override
    public void validateSyntax(JSONObject config, Context context) {
    }

    @Override
    public void validateSemantics(JSONObject policyConfig, Context context,
            List<Diagnostic> diagnostics) {
        /**
        String script = policyConfig.getString(SCRIPT);

        if (script == null || script.length() == 0) {

            Text validationMsg = new NonLocalizedText("The script field
cannot be empty.");
            Diagnostic diagnostic = new Diagnostic(Severity.Error,
validationMsg.id());
            diagnostics.add(diagnostic);
```

```
        }
        */
    }

}
```

Runtime Execution

Each policy provides implementation of the policy execution APIs.

Runtime.jar deals with runtime execution and the jar contains,

- Runtime class `public boolean execute`that implements the logic (ApiRuntimeContext context).
- RuntimeFactory class `public PolicyRuntime getRuntime`that is used to instantiate the runtime (PolicyRuntimeInitContext context, JSONObject policyConfig).

For information on runtime, refer to Oracle API Platform Cloud Service Runtime Context Java Doc.

Policy Package

The policy package is used to distribute policies, and register new policies through REST services.

A policy package is an archive that contains the following components:

Runtime.ear

- META-INF/
- META-INF/MANIFEST.MF
- APP-INF
- APP-INF/lib/
- APP-INF/lib/config.jar
- APP-INF/lib/runtime.jar

Policy.jar

- META-INF/
- META-INF/MANIFEST.MF
- ui.zip
- config.jar
- runtime.ear
- metadata.json

Policy_generated.ear

- META-INF/MANIFEST.MF
- APP-INF/lib/config.jar

- APP-INF/lib/runtime.jar
- **metadata.json**: A JSON document that contains either one metadata object or an array of metadata objects.
- **ui.zip**: Contains web resources and localization files
- **config.jar**: Contains configuration management binary classes
- **runtime.jar**: Contains runtime binary classes
- A manifest file is used to identify the nested files in the archive. An example of a MANIFEST file:

```
Manifest-Version: 1.0
Policy-Metadata: metadata.json
Policy-UI: ui.zip
Policy-Config: config.jar
Policy-Runtime: runtime.ear
```

Persistence

Policies are registered and managed individually in the database.

The policy definitions are stored in database. The database is split into two levels, a system level and a tenant-specific level. The system level is used to register policies delivered by Oracle, and policies in this level are available to all tenants. The tenant level is used to register tenant-specific policies or policies that need to be delivered for a tenant.

3

Develop and Deploy Custom Policies

The topics here guide you with the tasks that you must do for developing and deploying a custom policy.

Topics:

Typical Workflow for Developing Custom Policy

To understand the typical workflow for developing custom policies, review the topics.

Task	Description	More Information
Set up the Environment	Configure the environment for custom policy development.	Set up the Environment
Generate a Policy Stub	Download Python script and generate a Policy Stub using the script.	Generate a Policy Stub
Import the Policy into Eclipse	Import the project into eclipse after the policy stub is created	Import the Policy into Eclipse
Change the Version / Revision Number in Metadata	Increase the revision number of the policy each time you customize it.	Change the Version and Revision Number in Metadata
Build and Deploy the Policy	Build using the ant script and deploy policy JAR using the REST API available in APIP Management Tier.	Build and Deploy the Policy
Automate Build and Deploy	Automate revision, building, and deployment of a policy, if needed.	Automate Build and Deploy
Validate Build Structure	Validate the directory structure at {policy Name}/src	Validate Build Structure

Set up the Environment

Before you develop custom policies, ensure that environment for custom policy development is configured.

To set up the custom policy development environment:

1. Install / update to the latest version of Oracle JDK.
2. Install the latest version of Java IDE. For example, Eclipse or IntelliJ.
3. Sign into the Management Portal with a Gateway Manager or Administrator user.
4. Download the Gateway Node Installer.
5. Install a Gateway Node.
6. Copy and place the following jars in plugins directory.
   - `groovy-all.jar` from `<gateway_install_dir>/GATEWAY_HOME/ocsg/httpclient/`.
   - `oracle.apiplatform.policies.sdk.jar`, `oracle.apiplatform.shared.jar`, `oracle.apiplatform.utils.jar`, and `json-20160212.jar` from `<gateway_install_dir>/domain/gateway1/servers/managedServer1/tmp/_WL_user/SharedSDK/{custom folder}/APP-INF/lib`

Generate a Policy Stub

Generate a policy stub with a python script.

To generate a policy stub:

1. Navigate to `<ApicsGatewayInstaller.zip extracted folder>/policyTemplate/PolicyTemplate.zip`
2. Extract the contents of the `PolicyTemplate.zip` into a local folder.
3. Run the following Python script.

```
python PolicyTemplate.py -t {targetDirectory} -p {policyName} -s sdk [-h|--help]
```

When you run the script, ensure that:

- the target directory does not exist.
- the policy name could be used in a pathname / filename and also as part of a Java class / method name.
- the SDK parameter points to a directory that contains the JARs that the build and runtime relies on.

The script performs these actions:

- Creates the files that are required for creating a policy.
- Creates a directory structure that constitutes a valid Eclipse project that can be imported into the IDE.

- Checks files for tokens and replaces them appropriately.
  - Checks for tokens within files and filenames and renames.
  - Uses a template of files and directories partially made up of reserved names like `policyname` and `POLICYNAME`.
  - Generates .classpath file, which is specific to Eclipse, from scratch because its contents depend on what is in the SDK folder. As the number of SDK JARs can change, tokenization cannot generate this file.

  Tokens are case sensitive to meet the lowercase for Java package name and uppercase for class name requirements.

Import the Policy into Eclipse

Once the script is executed, policy stub is created. Import the project into eclipse after the policy stub is created.

To import the project into eclipse:

1. Open Eclipse IDE.
2. From the **File** menu, click **Open Project from File System or Archive.**.
3. Click **Directory** next to the **Import Source** field.
4. Specify the directory given as the target in the script.
5. The policy in the directory appears below. Select the policy.
6. Click **Finish**.

Build and Deploy the Customized Policy Manually

Learn how to manually build and deploy the customized policy.

Remember the following when you deploy the customized policy.

- Runtime behavior changes when a policy is published to the management tier with a modified revision.
- You can deploy the customized policy only if you are part of System Administrator group.
- The JAR file generated by the build process has a very unique packaging. The management tier validates the structure once you deploy the JAR file.

Topics:

Change the Version and Revision Number in Metadata

Increase the revision number of the policy each time you customize the policy and deploy it.

Management Tier checks whether the policy version and revision stored in the database matches with the metadata in the policy JAR file that you are deploying. If

that matches, the REST API returns an HTTP 200 code saying **OK** but no deployment happens.

To deploy the customized policy JAR, you must increase the revision number of the policy. You could also modify policy description to document the changes made to the policy.

To change the version / revision number in metadata:

1. Open the `policyXXX.json` file in a text editor.
2. Change the `policy.description` attribute.

   For example, change:`"policy.description" : ""Transforms a JSON payload into a XML payload”` to `"policy.description" : ""Transforms a JSON payload into a XML payload` **`after….”`**.
3. Save the changes and close the text editor.
4. Move to the folder that contains the `metadata.json` file.
5. Open the `metadata.json` file in a text editor.
6. Change the revision attribute.

   For example, change `"version": "1.0", "revision": "1"` to `"version": "1.0", "revision": "`**`2`**`"`.
7. Save the changes and close the text editor.

Build and Deploy the Policy

Build using the ant script and deploy policy JAR using the REST API available in APIP Management Tier.

You must run ant scripts to build and deploy from Eclipse. Ant script to build and deploy does not work outside Eclipse. To build and deploy the customized policy:

1. Navigate to the root folder the holds the policy implementation.
2. To build policy jars, execute the ant script `build.xml` from eclipse.
3. Once the script finishes, you can find the following new file in the root folder

   `{prefix to the policy jar}.{policyname}.jar` . Prefix to the policy jar is set in the build.properties file in the policy.
4. Deploy the policy JAR using the REST API available in APIP Management Tier.

   ```
   curl -X POST -u {username}:{password} http://{hostname}:{port number}/
   apiplatform/administration/v1/policies -H "Content-Type:application/
   octet-stream" --data-binary @{prefix to the policy jar}.{policyname}.jar
   ```

Automate Build and Deploy

You can also automate the revision, building, and deployment of a policy. You can configure for automating in build.properties file.

To automate revision and deployment of customized policies:

1. Open the build.properties file in a text editor.

2. Set the following values.

```
policyName={policy name}
jarPrefix= {prefix to the policy jar}
policy-sdk={location of the policy sdk}

autoRevision=false
autoDeploy=false

### if either 'autoRevision' or 'autoDeploy"
### properties are set to 'true', then the
### properties below must be set as well

mgmtServerHost=http:// {host name}
mgmtServerPort= {port number}
mgmtServerUser= {admin user name}
mgmtServerPass= {password}
```

   For example,

```
policyName=mypolicy
jarPrefix=apiplatform.policies
policy-sdk=/home/apics/labs/policy-develop/policy-sdk

autoRevision=true
autoDeploy=true

### if either 'autoRevision' or 'autoDeploy"
### properties are set to 'true', then the
### properties below must be set as well

mgmtServerHost= http://apics.xyz.com
mgmtServerPort= 7006
mgmtServerUser= admin
mgmtServerPass= pass123
```

3. Save the changes and close the text editor.
4. Execute the ant script `build.xml` from eclipse.
5. Once the script finishes, you can find the new file in the root folder and it is also automatically deployed to the management tier.
6. Go to the API implementation tab and check for the policy under the **Available Policies**, **Others** category.

   If the API Management Tier is open when policy is deployed, the new customized policy does not appear. Do a force browser refresh - Ctrl + F5 and APIP Management Tier reloads the policies.

Validate Build Structure

Once you run the script for building custom policy, you can validate the directory structure at `{policy Name}/src`.

Build structure:

```
`-- src
    |-- config
    |   `-- java
    |       `-- oracle
    |           `-- apiplatform
    |               `-- policies
    |                   `-- {policy name Folder}
    |                                   `-- {policy name}Constants.java
    |                                   `-- {policy name}Validator.java
    |-- metadata
    |   `-- metadata.json
    |-- runtime
    |   `-- java
    |       `-- oracle
    |           `-- apiplatform
    |               `-- policies
    |                   `-- {policy name Folder}
    |                           `-- {policy name}Runtime.java
    |                           `-- {policy name}RuntimeFactory.java
    |-- ui
        `-- resources
            |-- L10n
                    |-- Localization folders  -> folders for all
localizations supported
                    |-- {policy name}.json  -> JSON file containing token
            |    -- {policy name}-edit.html -> HTML markup of the policy
in edit mode
            |    -- {policy name}-edit.js   -> View model of policy in
edit mode
            |    -- {policy name}-view.html -> HTML markup of policy in
view mode
            |    -- {policy name}-view.js   -> View model of policy in
view mode
```

Glossary