Integrating Rule Apps into your CI/CD Pipeline

Welcome back! For those of you just joining us, this is part 2 of a pair of blog posts discussing CI/CD with Rule Apps – you can find the previous post here (url of previous post).

Part 2: irCatalog Check-in triggered, Catalog-sourced Rule App, Promoting pipeline

Scenario:

For this sample, we're going to use a rule app that is stored in an instance of the InRule Repository tool (irCatalog). We have an accompanying GitHub repository containing all our test scenarios, and we want to run the new version of a rule application against all those tests whenever one is checked into the `Catalog. If all tests pass, we then want to promote the rule app from one instance of irCatalog to the next one down the line.

Ready? Let's take a meander through it.

Trigger from irCatalog Check-in

Remember that irCatalogExtension Helper folder we skipped over last time? This is where that project comes into play. While the irCatalog service does not have a formal integration hook, we are able to leverage the WCF Endpoint Behavior configuration to add a Behavior Extension that injects a Parameter Inspector into the request/response pipeline. This gives us visibility into requests as they're entering the `Catalog, and responses as they are returned. We can inspect those as they pass and, if desired, manually trigger our pipeline from there.

```
public class RuleApplicationCheckinBehavior : BehaviorExtensionElement, IEndpointBehavior
{
    0 references | Dan Gardiner, 7 days ago | 1 author, 1 change
    protected override object CreateBehavior()
    {
        return new RuleApplicationCheckinBehavior();
    }
    0 references | Dan Gardiner, 7 days ago | 1 author, 1 change
    public override Type BehaviorType => typeof(RuleApplicationCheckinBehavior);
    0 references | Dan Gardiner, 7 days ago | 1 author, 1 change
    public void ApplyDispatchBehavior(ServiceEndpoint endpoint, EndpointDispatcher endpointDispatcher)
    {
        foreach (var operation in endpointDispatcher.DispatchRuntime.Operations)
        {
            operation.ParameterInspectors.Add(new RuleApplicationParameterInspector());
        }
    }
    Unused Interface Methods
}
```

In the CheckinRequestListener project, the RuleApplicationParameterInspector is where the application logic lives; that structure can be used as a foundation for creating custom assemblies. The essence of the

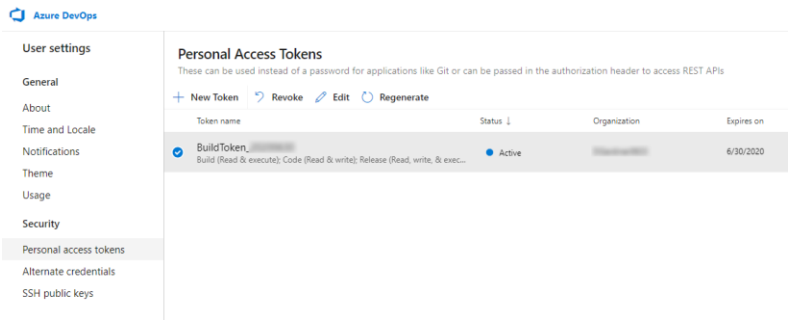
logic is that when a check-in or an apply label request completes, it may call out to Azure DevOps to trigger the pipeline to execute.

```
public class RuleApplicationParameterInspector : IParameterInspector
{
    0 references | Dan Gardiner, 17 hours ago | 1 author, 2 changes
    public object BeforeCall(string operationName, object[] inputs)...
    0 references | Dan Gardiner, 17 hours ago | 1 author, 2 changes
    public void AfterCall(string operationName, object[] outputs, object returnValue, object correlationState)
    {
        try
        {
            if (string.Equals(operationName, "CheckinRuleApp", StringComparison.OrdinalIgnoreCase))
            {
                var response = (InRule.Repository.Service.Data.Responses.CheckinRuleAppResponse)returnValue;
                var ruleApp = RuleAppDataCache.Add(response.RuleAppXml.Xml);

                // If you always want to trigger reguardless of label, then trigger upon checkin complete and not on ApplyLabel
                if(!IsLabelTriggerConfigured() && IsRuleAppConfiguredToTriggerPipeline(ruleApp.Name))
                    TriggerPipeline(ruleApp);
            }
            else if (string.Equals(operationName, "ApplyLabel", StringComparison.OrdinalIgnoreCase) && IsLabelTriggerConfigured())
            {
                string label = correlationState.ToString().Split('|')[0];
                if (IsLabelConfiguredToTriggerPipeline(label))
                {
                    string ruleAppGuid = correlationState.ToString().Split('|')[1];
                    var ruleApp = RuleAppDataCache.Get(ruleAppGuid);

                    if (IsRuleAppConfiguredToTriggerPipeline(ruleApp.Name))
                        TriggerPipeline(ruleApp);
                }
            }
        }
        catch(Exception ex)
        {
            Console.WriteLine("Error in RuleApplicationParameterInspector following checkin: " + ex.Message);
        }
    }
```

In order to trigger the pipeline, the request needs to contain an authentication token from Azure; the process to retrieve that is outlined here (https://docs.microsoft.com/en-us/azure/devops/integrate/get-started/authentication/pats?view=azure-devops). There are several other pieces of information required, including the Pipeline Organization, Project, and Pipeline ID (hint: the Pipeline ID can be found in the URL of the Azure DevOps Portal when the pipeline is open).



In the Catalog Extension project, the DevOps pipeline queue request is abstracted into the AxureDevOpsApiHelper class. More information about what else that API supports is available here (https://docs.microsoft.com/en-us/rest/api/azure/devops/build/builds/queue?view=azure-devops-rest-5.0#definitionreference). The readme in the project repo (https://github.com/InRule/DemoRuleCICDPipeline/tree/master/Helpers/CheckinRequestListener%20IrCatalogExtension/CheckinRequestListener) contains instructions about how to modify the web.config to

enable the custom extension to WCF behavior, and also for the extension to have the configuration information it needs to initiate the DevOps pipeline.

Pipeline Structure

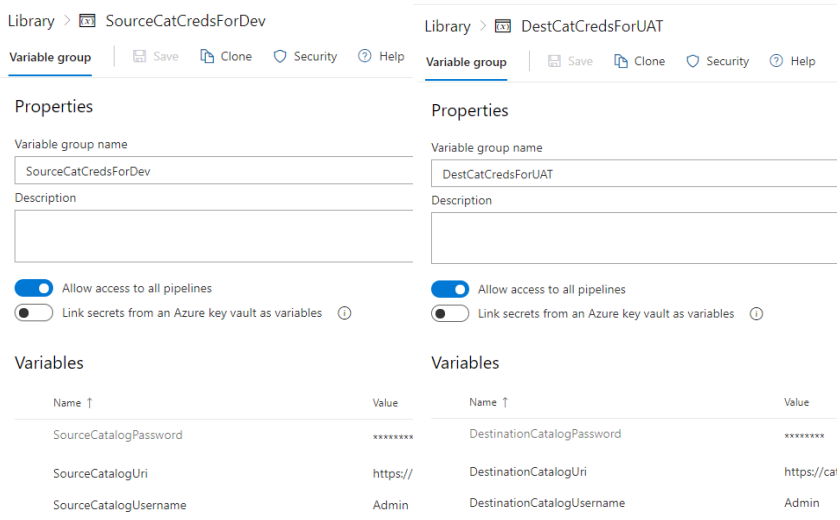
Like the pipeline discussed in the previous post in this series (Insert link here), this pipeline is structured with a template to make it reusable. We're also passing parameters into the template indicating which rule app we're working with, as well as which source and destination `Catalogs we want to use. Since we're manually triggering it with the irCatalog extension, we do not have any automatic triggers configured on this one.

```yaml
trigger: none

jobs:
- template: catalogSourced-TestAndPromote.yml
  parameters:
    RuleAppName: MultiplicationApp
    SourceCatalogName: Dev
    DestinationCatalogName: UAT
```

Catalog Credentials

As before, because we want to avoid storing credentials somewhere unsafe, we're keeping them in Library Variable Groups. Since we're going to be pulling in two Variable Groups at the same time in the pipeline (one for source and one for destination), we need to create "Source" and "Dest" prefixed variables and groups to avoid name collisions when importing multiple groups into the same job.

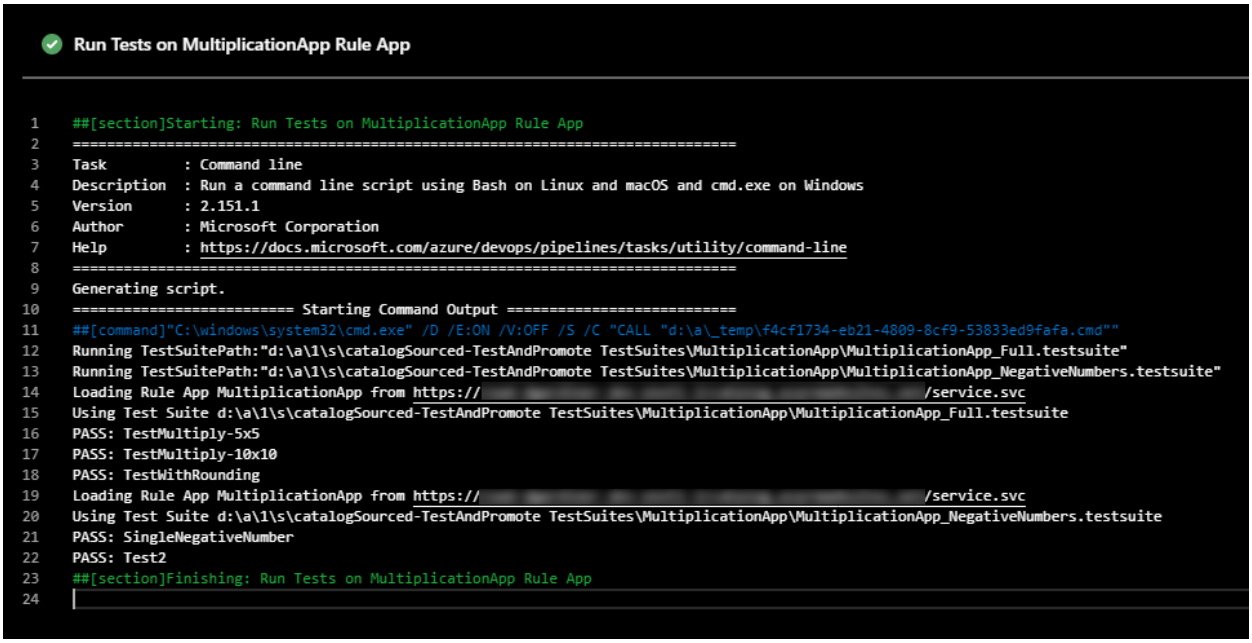


Test

Our job to perform the tests here is nearly identical to the previous post, but rather than providing a file path for the rule app to test, we're providing the `Catalog credentials that we retrieved from the

Credential Store. One unfortunate note here is that Azure DevOps has an open issue (https://github.com/MicrosoftDocs/vsts-docs/issues/3702) that Variable Groups cannot be imported using a name containing a variable substitution. In this case, the template only works with a fixed pair of catalog credential library variable group names.

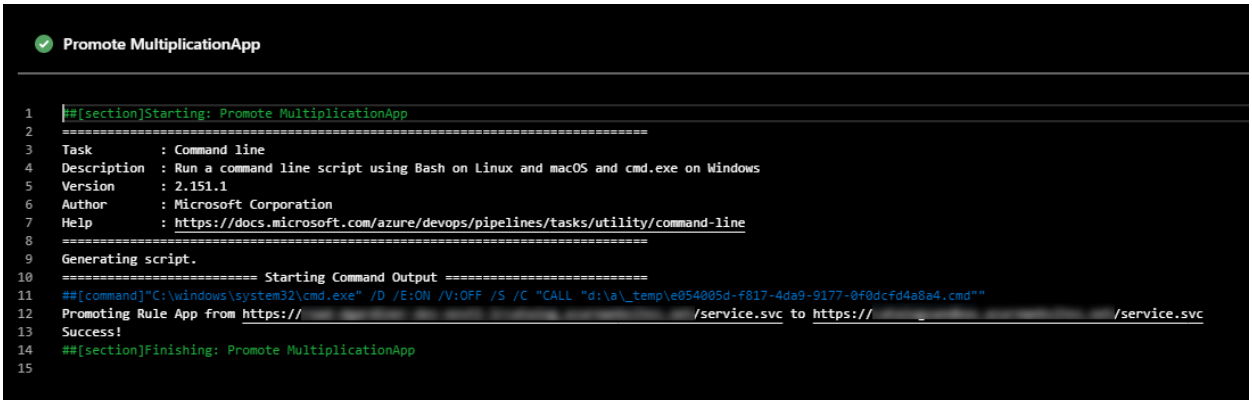
```
- job: Test_${{ parameters.RuleAppName }}
  pool:
    vmImage: 'windows-2019'
  variables:
  #- group: SourceCatCredsFor$(SourceCatalogName) This is not yet supported - https://github.com/MicrosoftDocs/vsts-docs/issues/3702
  - group: SourceCatCredsForDev
  - name: RuleAppName
    value: ${{ parameters.RuleAppName }}
  - name: RuleTestFolder
    value: $(System.DefaultWorkingDirectory)\catalogSourced-TestAndPromote TestSuites\$(RuleAppName)
  steps:
  - task: DownloadSecureFile@1
    inputs:
      secureFile: 'InRuleLicense.xml'
  - task: CopyFiles@2
    inputs:
      SourceFolder: '$(System.DefaultWorkingDirectory)/Helpers/ExecuteTests'
      Contents: '**'
      TargetFolder: '$(Agent.TempDirectory)'
      OverWrite: true
  - script: |
      for %%i in ("$(RuleTestFolder)\*.testsuite") do ( echo Running TestSuitePath:"%%i" )
      for %%i in ("$(RuleTestFolder)\*.testsuite") do ( .\ExecuteTests.exe -TestSuitePath:"%%i" -CatUri:"$(SourceCatalogUri)" -CatUsername:"$(SourceCatalogUsername)" -CatPassword:"$(SourceCatalogPassword)"
      -CatRuleAppName:"$(RuleAppName)")
    displayName: 'Run Tests on $(RuleAppName) Rule App'
    workingDirectory: $(Agent.TempDirectory)
```



Promote

Once we've tested and validated that all is well, we're now ready to promote between catalogs. This is performed in fundamentally the same way as the previous post's irDistribution request, but with a different command line tool and many more parameters to specify connection information for the two `Catalogs.

```yaml
- job: Promote_${{ parameters.RuleAppName }}
  dependsOn: Test_${{ parameters.RuleAppName }}
  pool:
    vmImage: 'windows-2019'
  variables:
  ##- group: SourceCatCredsFor$(SourceCatalogName) This is not yet supported - https://github.com/MicrosoftDocs/vsts-docs/issues/3702
  #- group: DestCatCredsFor$(DestinationCatalogName) This is not yet supported - https://github.com/MicrosoftDocs/vsts-docs/issues/3702
  - group: SourceCatCredsForDev
  - group: DestCatCredsForUAT
  - name: RuleAppName
    value: ${{ parameters.RuleAppName }}
  steps:
  - task: DownloadSecureFile@1
    inputs:
      secureFile: 'InRuleLicense.xml'
  - task: CopyFiles@2
    inputs:
      SourceFolder: '$(System.DefaultWorkingDirectory)/Helpers/PromoteRuleApp'
      Contents: '**'
      TargetFolder: '$(Agent.TempDirectory)'
      OverWrite: true
  - script: |
      echo Promoting Rule App from $(SourceCatalogUri) to $(DestinationCatalogUri)
      .\PromoteRuleApp.exe -RuleAppName:"$(RuleAppName)" -Comment:"Publish from command line tool" -SrcCatUri:"$(SourceCatalogUri)/core" -SrcCatUser:"$(SourceCatalogUsername)" -SrcCatPass:"$(SourceCatalogPassword)"
      -DestCatUri:"$(DestinationCatalogUri)/core" -DestCatUser:"$(DestinationCatalogUsername)" -DestCatPass:"$(DestinationCatalogPassword)"
    displayName: 'Promote $(RuleAppName)'
    workingDirectory: $(Agent.TempDirectory)
```



You Did It (Again)!

As before, you can view the full pipeline and GitHub structure of the catalogSourced-TestAndPromote-pipeline at https://github.com/InRule/DemoRuleCICDPipeline. This is just one example of the different ways these pieces can be put together. Feel free to adapt the elements of this exercise to whatever your needs are, and build out a CI/CD pipeline that give you exactly the functionality you need in your organization.

Was this helpful for you? What would you do differently? Let us know in the comments below!