VMware vRealize Orchestrator クライアントの使用

2022 年 2 月

vRealize Orchestrator 8.7

最新の技術ドキュメントは、 VMware の Web サイト （https://docs.vmware.com/jp/）

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

ヴイエムウェア株式会社
〒 108-0023 東京都港区芝浦 3-1-1
田町ステーションタワー N 18 階
www.vmware.com/jp

目次

VMware vRealize Orchestrator クライアントの使用

1

VMware vRealize Orchestrator クライアントを使用すると、ワークフローの自動化機能および vRealize Orchestrator Client の機能に関する情報を取得できます。

対象者

この情報は、vRealize Orchestrator ワークフローの実行と管理に役立つツールをお探しの経験豊富なシステム管理者を対象としています。

vRealize Orchestrator クライアントの概要

2

vRealize Orchestrator サービスおよびオブジェクトを管理するには、vRealize Orchestrator Client を使用します。

vRealize Orchestrator Client は `https://your_orchestrator_FQDN/orchestration-ui` からアクセスできます。

UI 要素	説明
[ダッシュボード]	vRealize Orchestrator Client ダッシュボードおよびプロファイル機能を使用して、vRealize Orchestrator 環境およびワークフローに関する有用なメトリック データを収集します。
[ワークフロー]	ワークフローを作成、編集、スケジュール設定、実行、および削除します。
[アクション]	アクションを作成、編集、および削除します。アクション エディタにより、vRealize Orchestrator API Explorer に含まれる共通のスクリプト要素の自動補完がサポートされています。
[ポリシー]	ポリシーを作成、編集、実行、および削除します。
[パッケージ]	vRealize Orchestrator オブジェクトを含むパッケージを作成、削除、エクスポート、およびインポートします。
[構成]	構成要素を作成、実行、および削除します。
[リソース]	リソース要素をエクスポート、インポート、および更新します。
[グループ]	管理者権限を持つユーザーは、vRealize Orchestrator Client のユーザーにロールを割り当て、グループに追加できます。
[監査ログ]	オブジェクトがいつ作成されたかなど、vRealize Orchestrator Client に記録されているさまざまなイベントを表示します。
[Git リポジトリ]	Git リポジトリとの統合を作成し、統合を使用して複数の環境にまたがるワークフローとその他の vRealize Orchestrator オブジェクトの開発を管理します。 Git ブランチを使用して vRealize Orchestrator オブジェクト インベントリを管理する方法を参照してください。
[削除済みアイテム]	ワークフロー、アクション、ポリシー、構成要素、リソース要素などの削除済みの vRealize Orchestrator Client オブジェクトをリストアします。
[API Explorer]	vRealize Orchestrator Client で使用できる API コマンドを検索できます。 **注：** vRealize Orchestrator Client は、REST プロキシを介して vRealize Orchestrator REST API と通信します。

この章には、次のトピックが含まれています。

vRealize Orchestrator クライアントの使用状況ダッシュボード

vRealize Orchestrator Client ダッシュボードは、vRealize Orchestrator Client ワークフローの監視、管理、およびトラブルシューティングに役立つツールを提供します。

vRealize Orchestrator Client ダッシュボードの情報は、5 つのパネルに表示されます。

ウィンドウ	説明
ワークフローの実行	実行中、待機中、および失敗したワークフローの実行の数に関する視覚的なデータを提供します。
お気に入りワークフロー	お気に入りに追加されたワークフローを表示します。
入力を待機中	追加のユーザー操作が必要な保留中のワークフローの実行が表示されます。これらのワークフローは、ユーザー インターフェイスの右上隅の通知メニューにも表示されます。
最近のワークフローの実行	最近のワークフローの実行を管理します。ワークフローの実行の名前、状態、開始日、および終了日を示します。
注意	失敗したワークフローの実行と、ワークフロー実行のパフォーマンス メトリックを表示します。

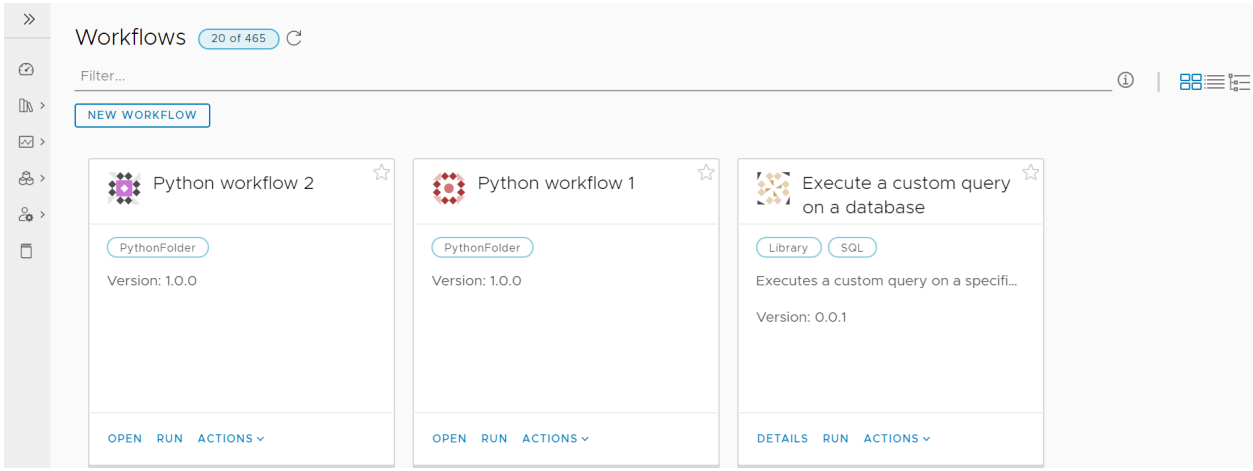
vRealize Orchestrator Client でのコンテンツの編成

vRealize Orchestrator オブジェクト インベントリの vRealize Orchestrator Client での表示方法を管理することができます。

vRealize Orchestrator Client では、ワークフロー、アクション、ポリシー、リソース、設定などのオブジェクトに対して、カード ビュー、リスト ビュー、ツリー ビューといった 3 つのビュー タイプをサポートしています。現在のビュー タイプは、ページの右上隅から変更できます。

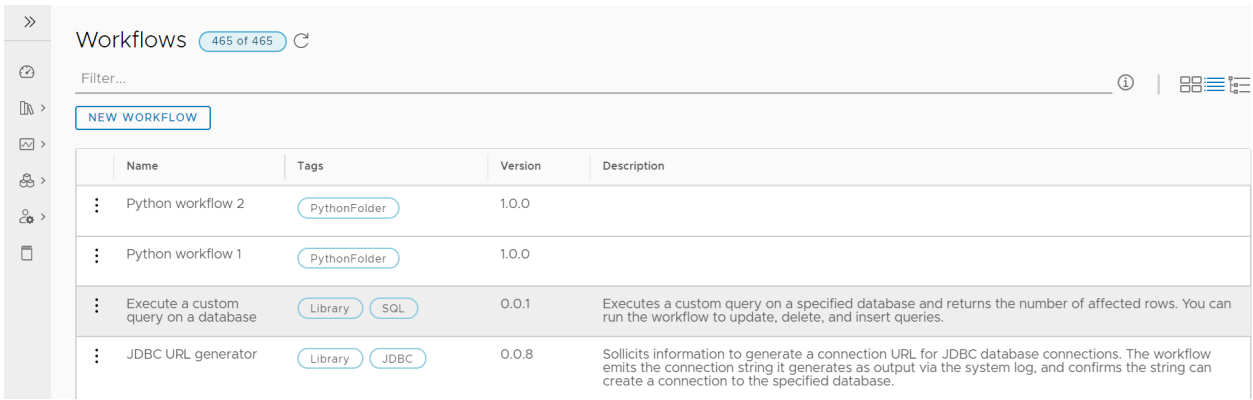
カード ビュー

カード ビューは、vRealize Orchestrator Client で使用されるデフォルトのビュー タイプです。ワークフローなど、個別のインベントリ オブジェクトに関する情報は、個々のカード要素に表示されます。



リスト ビュー

リスト ビューには、リストとして編成された vRealize Orchestrator オブジェクトに関する情報が表示されます。オブジェクトに対して実行できるアクションの詳細については、オブジェクトの左にある垂直方向の省略記号アイコンをクリックしてください。



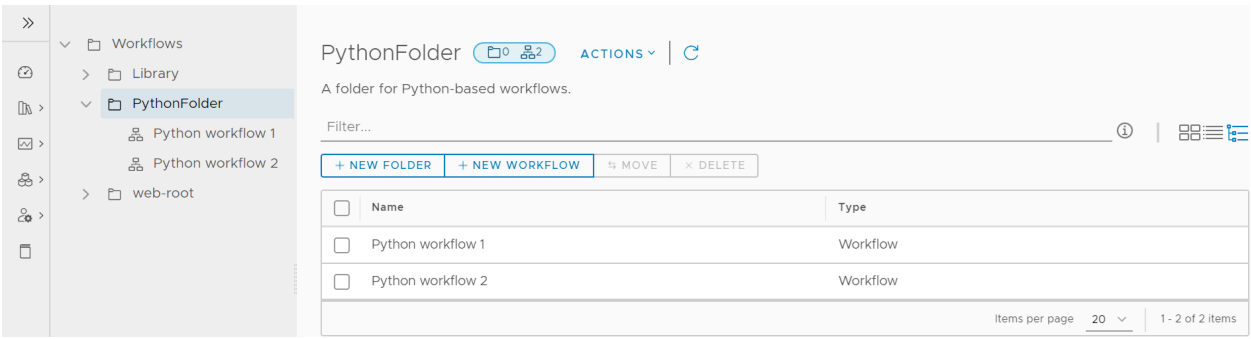
ツリー ビュー

ツリー ビューでは、オブジェクト インベントリを階層フォルダの下にまとめて整理することができます。各 vRealize Orchestrator オブジェクト タイプには、ルート レベルのフォルダがあります。ルート フォルダの下には、ワークフローなどの新しいオブジェクトを作成することはできません。ルート フォルダの下に編成されたフォルダを個別に作成する必要があります。各フォルダには、コンテンツ フィルタなどのコンテンツの管理に有用なツールが含まれています。

注： 各フォルダには個別のコンテンツ フィルタがあります。フォルダ間でコンテンツをフィルタリングすることはできません。

フォルダの詳細については、vRealize Orchestrator クライアントでのフォルダまたはサブフォルダの作成を参照してください。

注： ツリー ビューから選択したオブジェクトは、読み取り専用モードで開きます。ワークフロー変数、ワークフロー スキーマなどのオブジェクト コンテンツを編集するには、上部のオプション メニューで [編集] をクリックします。



vRealize Orchestrator クライアントでのフォルダまたはサブフォルダの作成

階層フォルダ構造を使用して、vRealize Orchestrator オブジェクトを編成します。

フォルダとサブフォルダを作成して、次のタイプの vRealize Orchestrator オブジェクトを編成できます。

- ワークフロー
- アクション
- ポリシー
- 構成要素
- リソース要素

手順

1 vRealize Orchestrator Client にログインします。

2 左側のナビゲーション ペインで、[ワークフロー] などのオブジェクト ページを選択します。

3 右上から、ツリー ビュー アイコン ( ) を選択します。

4 （オプション） サブフォルダを作成するには、左側のツリー ビューから親フォルダを選択します。

5 [新規フォルダ] をクリックします。

6 名前と説明を入力し、[保存] をクリックします。

7 オブジェクトまたはサブフォルダを新規作成したフォルダに追加します。

8 （オプション） フォルダ名を編集するには、[アクション] - [編集] の順に選択します。

vRealize Orchestrator Client でのオブジェクトとフォルダの移動

vRealize Orchestrator コンテンツを別のフォルダに移動して再編成します。

アクション モジュール間でアクションを移動したり、オブジェクトをルート フォルダに移動することはできません。ルート フォルダには、主なオブジェクト フォルダとサブフォルダが含まれていますが、オブジェクトの格納に使用することはできません。

手順

1 vRealize Orchestrator Client にログインします。

2 左側のナビゲーション ペインで、[ワークフロー] などのオブジェクト ページを選択します。

3 右上から、ツリー ビュー アイコン () を選択します。

4 ツリー ビューを展開し、移動するオブジェクトまたはフォルダを選択します。

5 オブジェクトまたはフォルダを新しい親フォルダにドラッグします。

> **注：** オブジェクトをオブジェクト エディタから直接新しいフォルダに移動することもできます。[概要] タブで [フォルダの選択] をクリックし、オブジェクトの新しい親フォルダを選択します。もう 1 つの移動オプションとして、フォルダ ページのテーブルからオブジェクトを選択します。このオプションは、複数の vRealize Orchestrator オブジェクトを含む一括移動操作を実行する場合に有用です。

vRealize Orchestrator Client でのフォルダまたはサブフォルダの削除

古いフォルダまたはサブフォルダを vRealize Orchestrator Client から削除します。

各 vRealize Orchestrator オブジェクト タイプの対応するルートレベルのフォルダを削除することはできません。

手順

1 vRealize Orchestrator Client にログインします。

2 左側のナビゲーション ペインで、[ワークフロー] などのオブジェクト ページを選択します。

3 右上から、ツリー ビュー アイコン () を選択します。

4 削除するフォルダの横にあるチェック ボックスをオンにします。

> **注：** サブフォルダを削除するには、ツリー ビューから親フォルダを選択し、チェック ボックスをオンにします。

5 [削除] をクリックします。

6 選択したフォルダが空の場合は、以下の手順を実行します。

   a フォルダを削除することを確認します。

   b [削除] をクリックします。

7 選択したフォルダに vRealize Orchestrator Client オブジェクトまたはサブフォルダが含まれている場合は、以下の手順を実行します。

a フォルダを削除することを確認します。

b [削除] をクリックします。

「アイテム「your_folder_name」を削除できませんでした： フォルダ「your_folder_name」が空ではありません」というメッセージが表示されます。

c フォルダとそのすべての内容を削除するには、[強制的に削除] をクリックします。

d フォルダを削除することを確認し、[削除] をクリックします。

注： フォルダ メニューに含まれているテーブルから複数のオブジェクトを選択して、一括削除を行うこともできます。

vRealize Orchestrator Client の設定 3

vRealize Orchestrator Client の機能を最大限に活用するには、ユーザー権限を構成し、バージョン履歴を使用してオブジェクトを管理する方法を学習する必要があります。

この章には、次のトピックが含まれています。

- vRealize Orchestrator のロールとグループ
- vRealize Orchestrator オブジェクトのバージョン履歴

vRealize Orchestrator のロールとグループ

vRealize Orchestrator 管理者は、vRealize Orchestrator Client の機能とコンテンツへのアクセスを制御する権限を設定できます。アクセス権は、ユーザー ロールとグループ権限に分けられます。

ロールは、ユーザーが表示および使用できる vRealize Orchestrator Client 機能を制御します。ロール管理機能へのアクセスは、vRealize Orchestrator 環境のライセンス タイプに依存します。

表 3-1. vRealize Orchestrator のロール管理へのライセンスベースのアクセス

	認証	
ライセンス	**[vSphere]**	**[vRealize Automation]**
[vSphere]	ロール管理はサポートされていません。グループがサポートするのは実行の権限のみです。	
[vRealize Automation]	vRealize Orchestrator クライアントのロールを管理します。 vRealize Orchestrator Client でのロールの割り当てを参照してください。	vRealize Automation の [ID とアクセス管理] を使用してロールを管理します。 vRealize Automation での vRealize Orchestrator クライアントのロールの設定を参照してください。

グループ権限は、ワークフロー、アクション、ポリシー、構成要素、リソース要素など、どの vRealize Orchestrator Client コンテンツをユーザーが表示および使用できるかを制御します。標準のワークフローやアクションなど、事前構成された vRealize Orchestrator システム コンテンツへのアクセスは、グループ権限を使用して構成されていない限り、すべてのユーザー間で共有されます。

管理者ロールおよび閲覧者ロールを持つユーザーのアクセス権は、グループ権限によって制限されません。ロールが割り当てられていないユーザー、およびワークフロー デザイナ ロールを持つユーザーのアクセス権は、そのユーザーに割り当てられたグループに依存します。グループの権限を変更することによって、これらのユーザーのアクセス権を拡大できます。この方法で、ユーザーを共通のプロジェクトに編成できます。たとえば、カスタムの vRealize Orchestrator プラグインの開発を行うユーザーを含むグループを作成し、それらのユーザーにはグループに固有のコンテンツのみを変更できるようにすることができます。

表 3-2. vRealize Orchestrator のユーザー ロールとグループ権限

ロール	アクセス権		
[管理者]	管理者は、特定のグループによって作成されたコンテンツを含む、すべての vRealize Orchestrator クライアントの機能およびコンテンツにアクセスできます。ユーザー ロールの設定、グループの作成と削除、およびグループへのユーザーの追加を担当します。管理者がグループ権限によって制限されることはありません。 **注：** vRealize Orchestrator の認証に使用される vRealize Automation 環境のテナント管理者には、デフォルトで [管理者] 権限が付与されます。		
[閲覧者]	閲覧者は vRealize Orchestrator クライアントのすべてのコンテンツに対して読み取り専用のアクセス権を持っていますが、コンテンツの作成、編集、実行、エクスポートはできません。閲覧者は、すべてのグループおよびグループ コンテンツを見ることもできます。閲覧者がグループ権限によって制限されることはありません。		
	[グループ権限]		
	[割り当てられたグループなし]	[実行]	[実行および編集]
[ワークフロー デザイナ]	■ システム コンテンツを表示します。 ■ 独自の実行を表示および実行します。 ■ 独自のコンテンツを作成、実行、編集、および削除します。	■ システム コンテンツを表示します。 ■ 独自の実行を表示および実行します。 ■ 独自のコンテンツを作成、実行、編集、および削除します。 ■ 独自のコンテンツをグループに追加します。 ■ グループ コンテンツを実行します。ただし、編集することはできません。	■ システム コンテンツを表示します。 ■ 独自の実行を表示および実行します。 ■ 独自のコンテンツを作成、実行、編集、および削除します。 ■ 独自のコンテンツをグループに追加します。 ■ グループ コンテンツを実行および編集します。 **注：** vSphere で認証された vRealize Orchestrator インスタンスでは使用できません。
ロールが割り当てられていないユーザー	■ 独自の実行を表示します。	■ 独自の実行を表示および実行します。 ■ グループ コンテンツを表示および実行します。	■ 独自の実行を表示および実行します。 ■ グループ コンテンツを表示および実行します。 **注：** コンテンツの作成、編集、追加をできるようにするには、このグループのユーザーにワークフロー デザイナ ロールを割り当てる必要があります。 **注：** vSphere で認証された vRealize Orchestrator インスタンスでは使用できません。

vRealize Orchestrator Client でのロールの割り当て

管理者は、ユーザーを vRealize Orchestrator Client に追加して、ユーザーが表示および使用できる機能を設定できます。

ロール管理では、vRealize Orchestrator Client の機能へのユーザーのアクセスを vRealize Orchestrator の ID プロバイダから制御します。ロール管理は、vRealize Orchestrator Client ユーザー インターフェイスと API 機能の両方に適用されます。

注： クライアント側のロール管理を使用できるのは、vRealize Automation ライセンスを使用する vSphere で認証された vRealize Orchestrator インスタンスのみです。vRealize Automation で認証された vRealize Orchestrator にロールを割り当てる方法については、vRealize Automation での vRealize Orchestrator クライアントのロールの設定を参照してください。

手順

1 vRealize Orchestrator クライアントに管理者としてログインします。

2 [管理] - [ロール管理] の順に移動します。

3 [追加] をクリックします。

4 vRealize Orchestrator Client に追加するユーザーまたはグループを検索します。

5 ユーザーのロールを選択します。ロールの詳細については、vRealize Orchestrator のロールとグループを参照してください。

6 [保存] をクリックします。

vRealize Automation での vRealize Orchestrator クライアントのロールの設定

vRealize Automation の [ID とアクセス管理] ページで vRealize Orchestrator Client のサービス ロールを割り当てることができます。サービス ロールは、組み込みの vRealize Orchestrator Client および vRealize Automation で認証されたスタンドアローン vRealize Orchestrator インスタンスに割り当てることができます。

vRealize Orchestrator サービス ロールは、組み込み vRealize Orchestrator Client のユーザーがアクセスできる機能を管理します。vRealize Orchestrator のロールの詳細については、vRealize Orchestrator のロールとグループを参照してください。

注： vRealize Automation ライセンスを使用する vSphere で認証されたスタンドアローン vRealize Orchestrator インスタンスは、vRealize Orchestrator Client でロールを直接割り当てることができます。『vRealize Orchestrator Client でのロールの割り当て』を参照してください。

前提条件

- 有効な vIDM インスタンスから適切なユーザーとグループがインポートされていることを確認します。
- vRealize Orchestrator のサービス ロールをユーザーに割り当てる前に、vRealize Automation でユーザーに割り当てられた組織ロールがあることを確認します。詳細については、『vRealize Automation の管理』の「vRealize Automation でのユーザーとグループの管理」を参照してください。

手順

1 右上のヘッダー ドロップダウン メニューから、[ID とアクセス管理] オプションを選択します。

2 [アクティブ ユーザー] タブで、vRealize Orchestrator に割り当てるユーザーのメール アドレスを検索します。

3 ユーザーの横にあるチェック ボックスを選択して、[ロールの編集] をクリックします。

4 [サービス アクセスの追加] をクリックします。

5 左側のドロップダウン メニューから、[Orchestrator] を選択します。

6 右側のドロップダウン メニューから、ユーザーに割り当てるロールを選択します。

7 [保存] をクリックします。

vRealize Orchestrator Client でのグループの作成

管理者は、グループを使用して、vRealize Orchestrator コンテンツ ユーザーが vRealize Orchestrator Client で表示およびアクセスできる内容を設定できます。

vRealize Orchestrator Client を使用して、vRealize Orchestrator ワークフロー、アクション、ポリシー、構成要素、リソース要素、およびパッケージに対するグループ権限を設定できます。

注： vSphere で認証された vRealize Orchestrator インスタンスのユーザーには、実行グループ権限のみを設定できます。

手順

1 vRealize Orchestrator クライアントに管理者としてログインします。

2 [管理] - [グループ] に移動します。

3 [新規グループ] をクリックします。

4 [サマリ] タブでグループの名前と説明を追加します。

5 [ユーザー] タブで [追加] をクリックします。

   a グループに追加するユーザーを検索します。

   b ユーザーにグループ権限を割り当てます。

   c [追加] をクリックします。

6 [項目] タブで、vRealize Orchestrator オブジェクトをグループに追加します。

   注： vRealize Orchestrator Client でオブジェクトを作成する際に、そのオブジェクトを既存のグループに追加することもできます。オブジェクトを追加するには、オブジェクト エディタの [サマリ/全般] タブの [アクセス許可] ドロップダウン メニューからグループを選択します。

7 [保存] をクリックします。

vRealize Orchestrator オブジェクトのバージョン履歴

vRealize Orchestrator Client は、各 vRealize Orchestrator オブジェクトのバージョン履歴レコードを保持します。バージョン履歴を使用すると、さまざまな vRealize Orchestrator オブジェクトのバージョンを比較して、以前のバージョンに戻すことができます。

vRealize Orchestrator は、オブジェクトを保存するときに、vRealize Orchestrator オブジェクトごとにバージョン履歴レコードを作成します。その後、vRealize Orchestrator オブジェクトを変更すると、新しいバージョン履歴レコードが作成されます。以前のバージョンの履歴レコードは保持され、オブジェクトに対する変更を追跡してオブジェクトを以前のバージョンに戻す際に使用できます。オブジェクトを以前のバージョンに戻すと、新しいバージョン履歴レコードが作成されます。

vRealize Orchestrator Client は、次の vRealize Orchestrator オブジェクトのバージョン履歴を追跡します。

- ワークフロー
- アクション
- パッケージ
- ポリシー
- リソース要素
- 構成要素

注： 生成されたワークフローは、ワークフローのバージョン履歴には表示されません。たとえば、[テーブルの CRUD ワークフローの生成] ワークフローによって生成されたワークフローは、[バージョン履歴] タブに表示されず、構成された Git リポジトリにプッシュすることはできません。これらのワークフローを vRealize Orchestrator バージョン履歴に含めるには、生成されたワークフローを複製します。

オブジェクトのバージョン履歴には、オブジェクト エディタ ページの [バージョン履歴] タブからアクセスできます。別のユーザーと同時にオブジェクトを編集すると、マージの競合が発生する可能性があります。マージの競合を解決するには、エラー メッセージの右にある [解決] をクリックします。[競合の解決] ウィンドウには、次の 3 つのオプションがあります。

- [相手の変更を使用]：他のユーザーが行った変更を使用してマージの競合を解決します。
- [自分の変更を使用]：自分が行った変更を使用してマージの競合を解決します。
- [解決]：表示された変更モデルを編集してマージの競合を解決します。指定されたモデルが無効の場合、このオプションは使用できません。

ワークフローを前のバージョンにリストアする

ワークフローを以前保存したバージョンにリストアできます。

手順

1 vRealize Orchestrator Client にログインします。

2 [ライブラリ] - [ワークフロー] の順に移動し、ワークフローを選択します。

3 [バージョン履歴] タブを選択します。

4 バージョン間の比較を表示するには、ワークフロー バージョンを選択し、[比較対象] ドロップダウン メニューから別のバージョンを選択します。

現在のワークフロー バージョンと選択したワークフロー バージョンの違いがウィンドウに表示されます。

5 ワークフローを別のバージョンにリストアするには、[リストア] をクリックします。

ワークフローの状態は選択したバージョンの状態に戻ります。

注： ワークフロー バージョンは、グラフィック差分ツール ビューからリストアすることもできます。ワークフロー バージョン間の視覚的な比較を参照してください。

ワークフロー バージョン間の視覚的な比較

ワークフロー バージョン間の変更を、グラフィック差分ツールで比較します。

デフォルトでは、vRealize Orchestrator のバージョン履歴には、ワークフロー バージョン間の差分が YAML 形式で表示されます。また、さまざまなワークフロー バージョン間で視覚的な比較を行うこともできます。次の変更を表示できます。

- バージョン番号、ワークフローの説明などの一般的なワークフロー情報。
- ワークフローで使用される変数。
- ワークフローの入力パラメータと出力パラメータ。
- ワークフロー スキーマ。

前提条件

ワークフローを作成します。

手順

1 vRealize Orchestrator Client にログインします。

2 [ライブラリ] - [ワークフロー] の順に移動し、ワークフローのいずれかを選択します。

3 ワークフローのコンテンツを編集します。

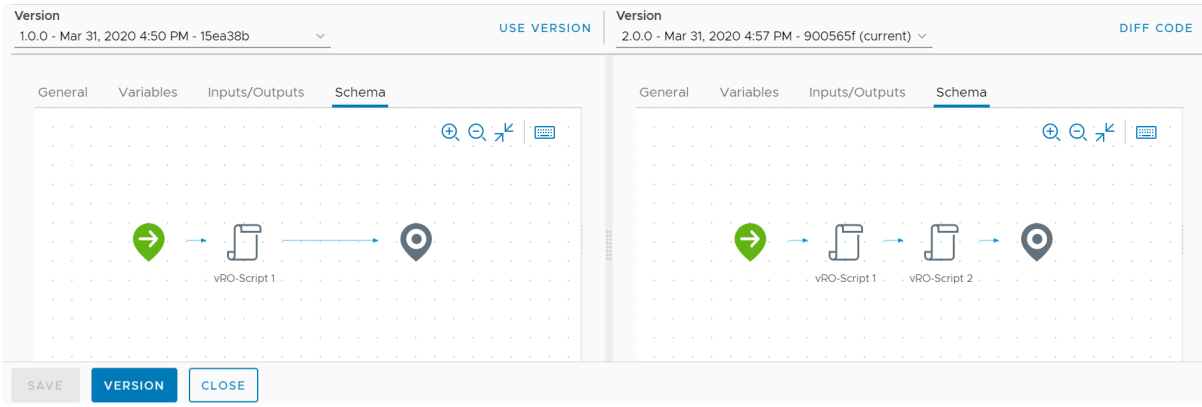
たとえば、[スキーマ] タブで追加の [スクリプト可能タスク] 要素を追加できます。

4 [保存] をクリックします。

5 [バージョン履歴] タブを選択します。

6 右上から、[視覚的な差分] を選択します。

選択した 2 つのワークフロー バージョン間で視覚的な比較を行うことができます。[バージョン] ドロップダウン メニューから比較するバージョンを選択できます。



7 （オプション） ワークフローを別のバージョンにリストアするには、[バージョンの使用] を選択します。

Git による vRealize Orchestrator コンテンツ インベントリの以前の状態へのリセット

過去の Git コミットを使用して、vRealize Orchestrator コンテンツを以前の状態にリセットすることができます。

特定のコミットを選択すると、vRealize Orchestrator コンテンツを以前の状態にリセットすることができます。

前提条件

- GitHub または GitLab リポジトリへの接続を設定します。『Git リポジトリとの接続の設定』を参照してください。
- ローカル変更セットを設定済みの Git リポジトリにプッシュします。

手順

1 vRealize Orchestrator Client にログインします。

2 [管理] - [Git 履歴] の順に移動します。

3 リセットする変更セットを選択し、[次にリセット] をクリックします。

4 この特定のコミットにリセットすることを確認し、[OK] をクリックします。

vRealize Orchestrator コンテンツ インベントリは、コミットで指定された状態にリセットされます。関連する vRealize Orchestrator コンテンツは以前のバージョンに戻されます。コミットがプッシュされたときになかったコンテンツは、インベントリから削除されます。

次のステップ

vRealize Orchestrator インベントリを Git リポジトリに保存されている最新の状態にリストアするには、[Git 履歴] ウィンドウで `Pull` コマンドを実行します。

vRealize Orchestrator の使用事例

4

これらの使用事例では、vRealize Orchestrator プラットフォームの機能の一部を示しています。

使用事例で示される値は、単なる例です。構造と命名規則はそれぞれの環境によって異なることがあります。

この章には、次のトピックが含まれています。

- Python を使用して vRealize Orchestrator で Amazon Web Services を統合する方法
- Git ブランチを使用して vRealize Orchestrator オブジェクト インベントリを管理する方法
- サードパーティ モジュールを使用して vRealize Automation プロジェクト API を呼び出す方法

Python を使用して vRealize Orchestrator で Amazon Web Services を統合する方法

この vRealize Orchestrator の使用事例では、Python を使用して vRealize Orchestrator 環境の機能を拡張する方法の例を示します。

アクションおよびワークフロー スクリプトでは、次のランタイムを使用できます。

- Python 3.7
- Node.js 14
- PowerCLI 11/Powershell 6.2
- PowerCLI 12.3.0/Powershell 7.1

> **注：** PowerCLI ランタイムには PowerShell と次のモジュールが含まれています：VMware.PowerCLI、PowerNSX、PowervRA。

重要： 新しいランタイムを使用できるのは、vRealize Orchestrator 環境で vRealize Automation ライセンスが使用されている場合のみです。

この使用事例では、Amazon Web Services (AWS) で EC2 インスタンスを呼び出す Python スクリプトを作成する方法を示します。

重要： カスタム スクリプトの開発を開始する前に、vRealize Orchestrator での Python、Node.js、および PowerShell スクリプトの使用の主要な概念について理解していることを確認します。Python、Node.js、PowerShell スクリプトの主要な概念 を参照してください。

手順

1 初期 Python スクリプトの作成

ローカル マシンで、Python スクリプトを作成し、スクリプトと boto3 ライブラリを ZIP フォルダとしてパッケージ化します。

2 Amazon Web Services アクションの作成

Python スクリプトを使用する vRealize Orchestrator アクションを作成します。

3 Amazon Web Services アクションのデバッグ

Python スクリプトの元のバージョンには、スクリプトのデバッグ方法を学習できるように、エラーが意図的に組み込まれています。

4 Amazon Web Services アクションの更新

更新された Python スクリプトをインポートし、アクションを再度実行します。

初期 Python スクリプトの作成

ローカル マシンで、Python スクリプトを作成し、スクリプトと boto3 ライブラリを ZIP フォルダとしてパッケージ化します。

前提条件

- Python 3 をダウンロードしてインストールします。Python のダウンロード ページを参照してください。
- Visual Studio Code をダウンロードしてインストールします。Visual Studio Code のダウンロード ページを参照してください。
- Visual Studio Code 用 Python 拡張機能をインストールしていることを確認します。「Visual Studio Marketplace」を参照してください。

手順

1 ローカル マシンで、 `vro-python-aws` フォルダを作成し、boto3 Python SDK をインストールします。

```
mkdir vro-python-aws
cd vro-python-aws
mkdir lib
pip install boto3 -t lib/
```

2 エディタを開き、main の Python スクリプトを作成します。この使用事例では、Visual Studio Code を使用しています。

```
import boto3

def handler(context, inputs):
    ec2 = boto3.resource('ec2')
    filters = [{
        'Name': 'instance-state-name',
        'Values': ['running']
    }]

    instances = ec2.instances.filter(Filters=filters)
    for instance in instances:
        print('Instance: ' + instance.id)
```

この Python スクリプトは、特定のリージョンで実行されているすべての EC2 インスタンスを一覧表示します。

3 作成したスクリプトを `main.py` ファイルとして `vro-python-aws` フォルダに保存します。

4 コマンドライン インターフェイスにログインします。

5 `vro-python-aws` フォルダに移動します。

```
cd vro-python-aws
```

6 Python スクリプトを含む ZIP パッケージを作成します。

```
zip -r --exclude=*.zip -X vro-python-aws.zip .
```

注： ZIP パッケージは、7-Zip などの ZIP ユーティリティ ツールを使用して作成することもできます。

結果

これで、ベースとなる Python スクリプトが作成され、vRealize Orchestrator 環境にインポートするための準備ができました。

Amazon Web Services アクションの作成

Python スクリプトを使用する vRealize Orchestrator アクションを作成します。

手順

1 vRealize Orchestrator クライアントにログインします。

2 [ライブラリ] - [アクション] の順に移動します。

3 [新規アクション] をクリックします。

4 [全般] タブでアクションの名前、モジュール、およびバージョン番号を入力します。

5 [スクリプト] タブで、ランタイムとして [Python 3.7] を選択し、スクリプト タイプとして [Zip] を選択します。

6 [インポート] をクリックします。

7 `vro-python-aws` フォルダを参照し、Python スクリプトが含まれている ZIP パッケージを選択します。

8 [エントリ ハンドラ] テキスト ボックスに **`main.handler`** と入力します。

注： アクションのエントリ ハンドラは、インポートされた ZIP パッケージの main スクリプトに基づいています。main スクリプトは `main.py` と呼ばれるファイルにあり、**`handler`** と呼ばれる関数であるため、エントリ ハンドラは **`main.handler`** である必要があります。main スクリプト ファイルの名前が異なる場合は、エントリ ハンドラの値を適宜変更します。

9 アクションを保存し、[実行] をクリックします。

アクションの実行でエラーが発生します。

10 [ログ] タブを選択します。

アクション実行のログに「`botocore.exceptions.NoRegionError: You must specify a region.`」というエラー メッセージが記録されます。初期 Python スクリプトではリージョンが定義されていないため、これは予期される動作です。

次のステップ

Python スクリプトをデバッグします。Amazon Web Services アクションのデバッグを参照してください。

Amazon Web Services アクションのデバッグ

Python スクリプトの元のバージョンには、スクリプトのデバッグ方法を学習できるように、エラーが意図的に組み込まれています。

前提条件

Amazon Web Services (AWS) アカウントにログインし、この使用事例シナリオ専用の IAM ユーザーを作成します。AWS アカウントでの IAM ユーザーの作成を参照してください。IAM ユーザーには、次の権限が付与されている必要があります。

```
"Effect": "Allow",
"Action": "ec2:DescribeInstances",
"Resource": "*"
```

手順

1 vRealize Orchestrator Appliance を準備します。

注意： vRealize Orchestrator 本番環境ではスクリプトをデバッグしないでください。開発およびテストに使用する単一ノード vRealize Orchestrator 環境からデバッグします。

a SSH を使用して、vRealize Orchestrator Appliance のコマンド ラインに root としてログインします。

b `vracli dev tools` コマンドを実行します。

c 続行を確認するプロンプトが表示されます。続行するには **yes**、キャンセルするには **no** を入力します。

重要： `vracli dev tools` コマンドを実行すると、Python スクリプトのデバッグに必要なポートが開きます。デバッグ中は、現在の SSH セッションを開いたままにする必要があります。

2 デバッグ構成を開始します。

a vRealize Orchestrator クライアントにログインします。

b AWS アクションを開き、[デバッグ] をクリックします。

デバッグ プロセスが開始され、アクションの実行が中断されます。

c [デバッグ構成] タブを選択します。

このタブには、リモートで IDE に接続して Python スクリプトをデバッグできる `.json` 構成が含まれています。

d 構成内容を手動でコピーするか、[クリップボードにコピー] をクリックします。

3 Python スクリプトをデバッグします。

a Visual Studio Code を開きます。

b `vro-python-aws` フォルダを開きます。

c 上部のナビゲーション ペインで、[実行] - [構成を開く] の順に選択します。

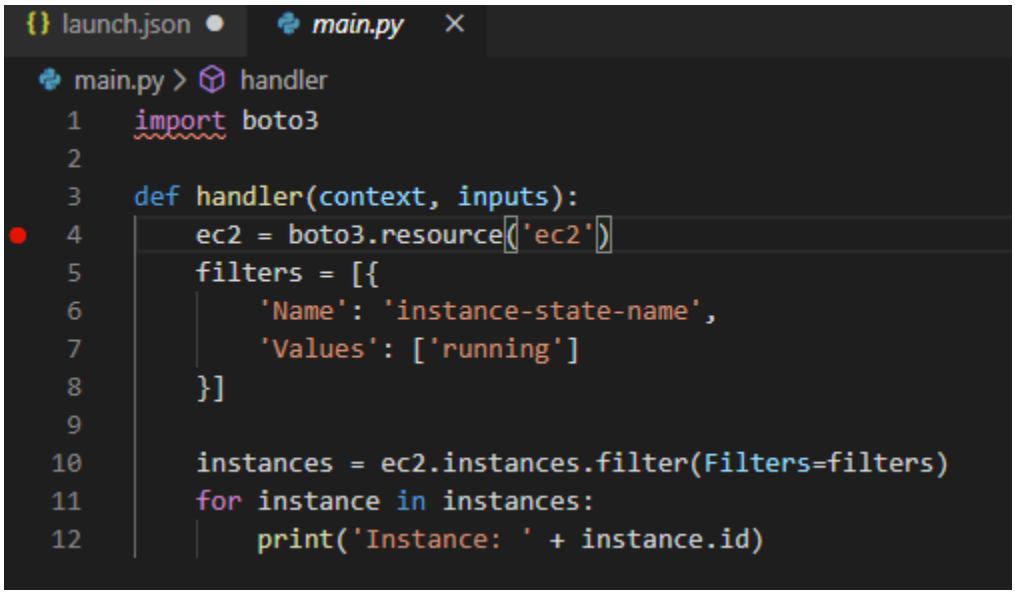
d [Python ファイル] を選択します。

e "version" 属性と "configuration" 属性を現在の位置のままにし、vRealize Orchestrator クライアントからコピーした .json 構成の内容を貼り付けます。生成された launch.json ファイルは、次のようになる必要があります。

```
{

    "version": "0.2.0",
    "configurations": [
        {
            "request": "attach",
            "port": 18281,
            "name": "vRO Python debug 8302f4c7-5beb-40da-848a-5003c0296f7b",
            "host": "es-sof-vc-vm-225-190.sof-mbu.eng.vmware.com",
            "type": "python",
            "pathMappings": [
              {
                "localRoot": "${workspaceFolder}",
                "remoteRoot": "/var/run/vco-polyglot/function"
            }
        ]
      }
    ]
  }
```

f main.py スクリプト ファイルを選択し、ブレークポイントを ec2 = boto3.resource('ec2') 行に追加します。



g 上部のナビゲーション ペインで、[実行] - [デバッグの開始] の順に選択します。

h デバッガがブレークポイントに到達すると、ステップ オーバー処理が実行されます。

デバッグの実行では、Python スクリプトに指定されたリージョンと AWS アクセス キーがないことが示されます。

i 開いている vRealize Orchestrator Appliance セッションに戻り、**Enter** キーを押して、このデバッグ セッション用に開いたポートを閉じます。

4 不足している情報を Python スクリプトに追加します。

a Visual Studio Code で `awsconfig` というファイルを作成します。これには、IAM ユーザーの AWS アクセス キーと、Python スクリプトを使用して ping を実行する AWS リージョンが含まれます。

```
[default]
aws_access_key_id=your key ID
aws_secret_access_key=your secret access key
region=your-region
```

b `awsconfig` を構成 (.cfg) ファイルとして `vro-python-aws` フォルダに保存します。

c `main.py` ファイルを開き、boto3 ライブラリで `awsconfig.cfg` ファイルを使用できるように変更します。

```
import boto3

import os
os.environ['AWS_CONFIG_FILE'] = os.getcwd() + '/awsconfig.cfg'

def handler(context, inputs):
    ec2 = boto3.resource('ec2')
    filters = [{
        'Name': 'instance-state-name',
        'Values': ['running']
    }]

    instances = ec2.instances.filter(Filters=filters)
    for instance in instances:
        print('Instance: ' + instance.id)
```

d `main.py` ファイル、`awsconfig.cfg` ファイルおよび boto3 ライブラリを含む新しい ZIP パッケージを作成します。

```
zip -r --exclude=*.zip -X vro-python-aws.zip .
```

注： ZIP パッケージは、7-Zip などの ZIP ユーティリティ ツールを使用して作成することもできます。

Amazon Web Services アクションの更新

更新された Python スクリプトをインポートし、アクションを再度実行します。

手順

1 vRealize Orchestrator クライアントにログインします。

2 [ライブラリ] - [アクション] の順に移動し、元の Amazon Web Services (AWS) アクションを選択します。

3 （オプション） [全般] タブで、バージョン番号を変更します。

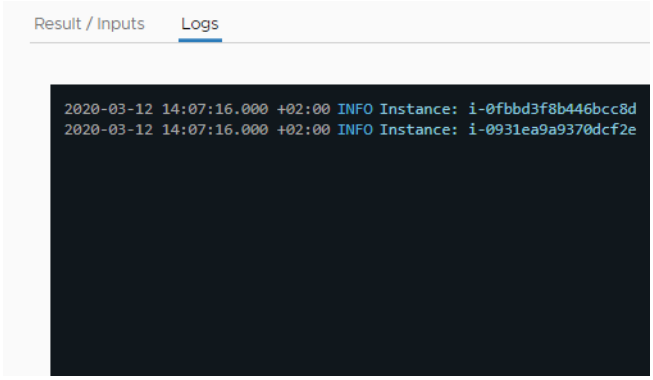
4 古い ZIP パッケージを削除し、[インポート] をクリックします。

5 更新された ZIP パッケージを選択します。

6 アクションを保存し、[実行] をクリックします。

7 アクションの実行が完了したら、[ログ] タブを選択します。

ログには、アクションによってクエリされた EC2 インスタンスが表示されます。



次のステップ

更新された AWS アクションを [アクション要素] として使用する vRealize Orchestrator ワークフローを作成します。

Git ブランチを使用して vRealize Orchestrator オブジェクト インベントリを管理する方法

ブランチを使用して、vRealize Orchestrator コンテンツを Git リポジトリで管理する方法を編成します。

Git を使用すると、リポジトリを一元化できるため、vRealize Orchestrator の開発における柔軟性を高めることができます。たとえば、Git を使用すると、複数の vRealize Orchestrator 環境にまたがってワークフロー開発を管理できます。

注： Git を使用してオブジェクト インベントリを管理するには、vRealize Orchestrator 環境で vRealize Automation ライセンスを使用する必要があります。詳細については、『vRealize Orchestrator のインストールと構成』の「ライセンスによる vRealize Orchestrator 機能の有効化」を参照してください。

これで、ブランチ間でオブジェクトのプッシュとプルができるようになりました。ブランチを使用することで、メイン ブランチにマージする前の vRealize Orchestrator オブジェクトの特定のグループの開発を管理することができます。

この使用事例では、GitLab プロジェクトを使用して、Python ランタイムを使用する vRealize Orchestrator オブジェクトを管理しています。この使用事例は vRealize Orchestrator の Git 機能の例を示しており、機能範囲の制限を示しているわけではありません。

注： GitHub の使用に慣れている場合は、この使用事例で GitHub リポジトリを使用することも可能です。

手順

1 GitLab 環境の準備

vRealize Orchestrator Python オブジェクト用 Git ブランチを作成します。

2 Git リポジトリとの接続の設定

管理者 として、vRealize Orchestrator 展開環境と Git リポジトリまたはプロジェクト間の接続を設定することができます。

3 Git リポジトリへの変更のプッシュ

ローカル vRealize Orchestrator オブジェクトへの変更を統合 Git リポジトリにプッシュします。この使用事例では、Python ベースの vRealize Orchestrator アクションへの変更を特定の Git ブランチにプッシュします。

GitLab 環境の準備

vRealize Orchestrator Python オブジェクト用 Git ブランチを作成します。

前提条件

vRealize Orchestrator 環境用 GitLab プロジェクトを作成します。プロジェクトの作成を参照してください。

手順

1 GitLab アカウントにログインします。

2 GitLab プロジェクトに移動します。

3 左側のナビゲーション ペインで、[リポジトリ] - [ブランチ] の順に選択します。

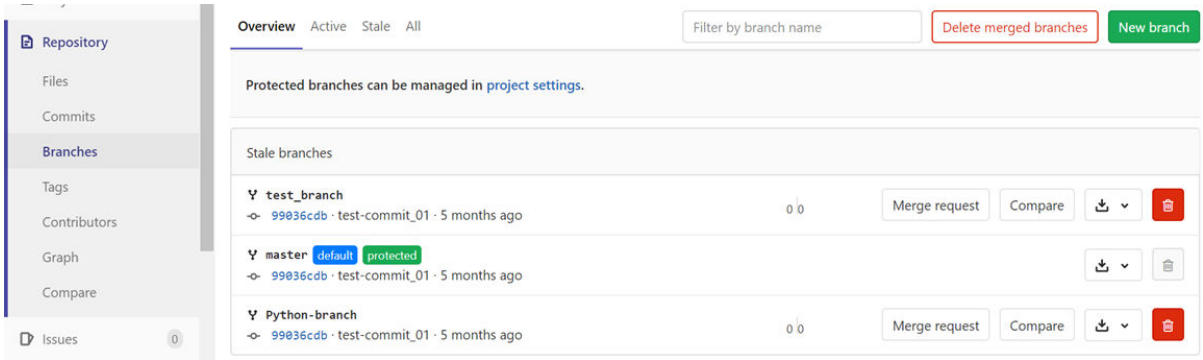
4 [概要] タブで、[新規ブランチ] をクリックします。

5 [ブランチ名] で、**`Python-branch`** と入力します。

6 [作成元] オプションは **`master`** のままにします。

7 [ブランチの作成] をクリックします。

Python ベースの vRealize Orchestrator オブジェクト用のブランチが作成されます。



Git リポジトリとの接続の設定

管理者 として、vRealize Orchestrator 展開環境と Git リポジトリまたはプロジェクト間の接続を設定することができます。

vRealize Orchestrator オブジェクト インベントリの管理に Git を使用するには、vRealize Orchestrator Client を使用して Git リポジトリへの接続を設定する必要があります。

注： vRealize Orchestrator がインスタンスごとに 1 つの SSH キーを作成するため、SSH を介して異なるアカウントから複数の Git リポジトリを追加することはできません。複数の Git リポジトリを追加するには、このドキュメントの説明に従って HTTP を介して追加します。

前提条件

- 現在の vRealize Orchestrator 環境で vRealize Automation ライセンスが使用されていることを確認します。
- GitLab プロジェクトのアクセス トークンを生成し、設定処理で使用するためにクリップボードにコピーします。パーソナル アクセス トークンの作成を参照してください。

  注： この使用事例では、GitLab プロジェクトを使用しています。GitHub に熟知している場合は、GitHub リポジトリを使用できます。GitHub トークン生成の詳細については、コマンド ライン用パーソナル アクセス トークンの作成を参照してください。

手順

1. 管理者として vRealize Orchestrator Client にログインします。
2. [管理] - [Git リポジトリ] の順に移動します。
3. [リポジトリの追加] をクリックします。
4. Git リポジトリの URL アドレスを入力します。

   たとえば、https://gitlab.com/*myusername*/*my-vro-repo* と入力します。

   注： SSH プロトコルで接続を確立することもできます。
5. Git プロファイルのユーザー名を入力します。
6. Git リポジトリのアクセス トークンを入力します。
7. Git リポジトリへの接続を検証するには、[検証] をクリックします。
8. （オプション） vRealize Orchestrator Client 内でリポジトリを識別するために使用する名前を変更します。
9. （オプション） 接続された Git リポジトリの簡単な説明を追加します。
10. 接続された Git リポジトリを有効にするには、[アクティブなリポジトリの作成] をクリックします。

    注： 同時にアクティブにできる Git リポジトリは 1 つだけです。アクティブな Git リポジトリの変更は、[Git リポジトリ] ページから行えます。
11. この変更をプッシュするブランチを選択します。この使用事例では、[Python-branch] を使用しています。GitLab 環境の準備を参照してください。

    注： 選択した Git ブランチは、最初の Git 設定が完了した後はいつでも変更できます。
12. 設定処理を終了するには、[保存] をクリックします。

次のステップ

[Git リポジトリ] メニューに戻り、リポジトリのステータスが [アクティブ] であることを確認します。

Git リポジトリへの変更のプッシュ

ローカル vRealize Orchestrator オブジェクトへの変更を統合 Git リポジトリにプッシュします。この使用事例では、Python ベースの vRealize Orchestrator アクションへの変更を特定の Git ブランチにプッシュします。

ローカルでの変更のセットを Git リポジトリにプッシュできます。各変更セットは、1 つ以上の変更された vRealize Orchestrator オブジェクトで構成されます。

注： 変更セットを Git リポジトリにプッシュおよび破棄するプロセスは、グループの権限によって制限されません。そのため、あるグループのワークフロー開発者が別の開発者の加えたローカル変更をプッシュまたは破棄できます。

前提条件

- Git ブランチを作成していることを確認します。GitLab 環境の準備を参照してください。
- Git リポジトリとの接続を設定していることを確認します。Git リポジトリとの接続の設定を参照してください。
- Git 統合が変更を [Python-branch] Git ブランチにプッシュするように設定されていることを確認します。
- Python ベースの vRealize Orchestrator オブジェクトを作成します。たとえば、Python を使用して vRealize Orchestrator で Amazon Web Services を統合する方法を参照してください。

手順

1. vRealize Orchestrator Client にログインします。
2. Python アクションを編集します。
   a. [ライブラリ] - [アクション] の順に移動し、Python アクションを選択します。
   b. 説明の変更など、アクションにマイナー変更を行います。
   c. アクションを保存します。

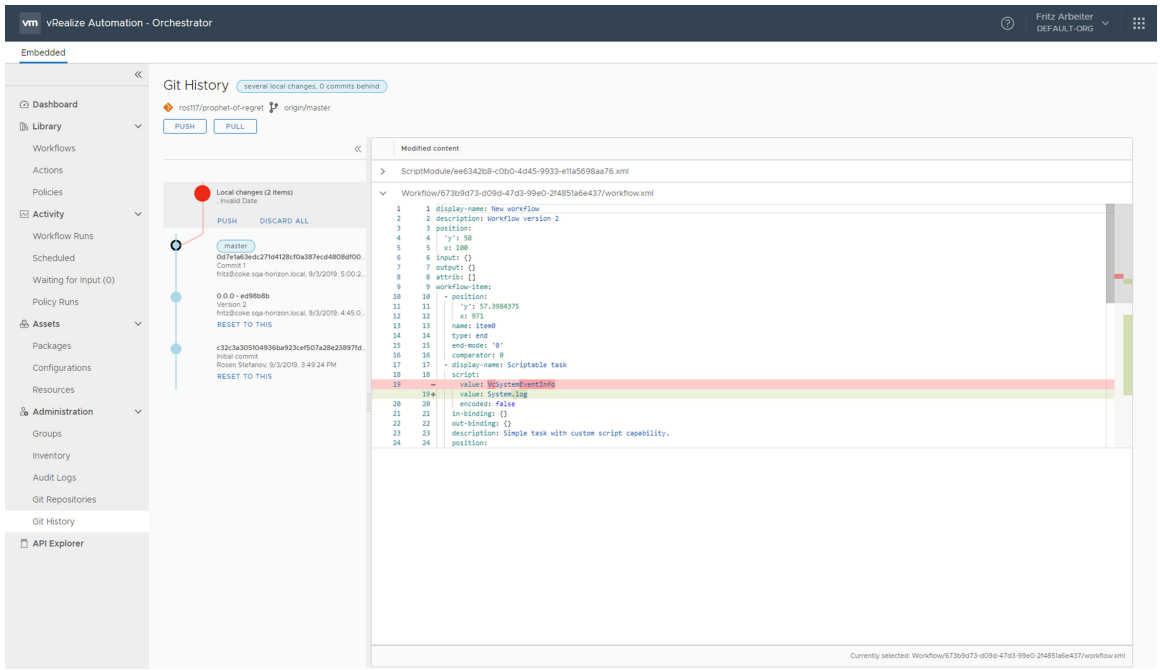
3 変更を Git リポジトリにプッシュします。

注： オブジェクト エディタの下部に表示される [バージョン] オプションをクリックして、オブジェクト レベルでローカルの変更をプッシュすることもできます。

a [管理] - [Git 履歴] の順に移動します。

[Git 履歴] 画面には、ローカル バージョンのブランチと選択した Git リポジトリのブランチとの現在の相違点が表示されます。変更された vRealize Orchestrator オブジェクトのエントリを展開して、バージョンの違いを表示できます。

注： ローカルでの変更のセットを破棄するには、[すべて破棄] を選択します。



b [プッシュ] をクリックします。

c コミットのタイトルを入力します。

d （オプション） コミットに関する短い説明を入力します。

e Git リポジトリにプッシュする Python アクションの変更を選択します。

4 ローカルでの変更のセットの Git リポジトリへのプッシュを完了するには、[プッシュ] をクリックします。

次のステップ

Git ブランチでの開発が完了したら、メイン ブランチとマージします。マージ要求の作成方法を参照してください。

サードパーティ モジュールを使用して vRealize Automation プロジェクト API を呼び出す方法

この vRealize Orchestrator の使用事例では、サードパーティ モジュールを使用して vRealize Automation プロジェクト API を呼び出す方法について説明しています。

アクションおよびワークフロー スクリプトでは、次のランタイムを使用できます。

- Python 3.7
- Node.js 14
- PowerCLI 11/Powershell 6.2
- PowerCLI 12.3.0/Powershell 7.1

> **注：** PowerCLI ランタイムには PowerShell と次のモジュールが含まれています：VMware.PowerCLI、PowerNSX、PowervRA。

この使用事例では、サードパーティの依存モジュールを使用して vRealize Automation プロジェクト API に接続する vRealize Orchestrator アクションの作成方法について説明しています。

重要： カスタム スクリプトの開発を開始する前に、vRealize Orchestrator での Python、Node.js、および PowerShell スクリプトの使用の主要な概念について理解していることを確認します。Python、Node.js、PowerShell スクリプトの主要な概念 を参照してください。

vRealize Automation プロジェクト API を呼び出す Python スクリプトの作成

Python を使用して vRealize Automation プロジェクト API を呼び出すサンプル スクリプトを作成します。

前提条件

Python 3 がインストールされていることと、PIP パッケージ インストーラがあることを確認します。Python のダウンロード ページおよび Python パッケージのインデックスを参照してください。

手順

1 ローカル マシンで、コマンドライン シェルを開きます。

2 `vro-python-vra` フォルダを作成します。

```
mkdir vro-python-vra
```

3 `vro-python-vra` フォルダに移動します。

```
cd vro-python-vra
```

4 `handler.py` という Python スクリプトを作成します。

```
touch handler.py
```

handler.py スクリプトでは、2 つの引数を受け入れる 1 つの関数と vRealize Orchestrator ワークフローの実行のコンテキスト、およびバインドされた vRealize Orchestrator の入力を定義する必要があります。

```
def handler(context, inputs):
    print('Hello, your inputs were ' + inputs)
    return None
```

注： 標準のログ ライブラリを使用すると、このスクリプトを使用するアクションで記録されたすべてがワークフロー ログにも表示されます。スクリプトの入力値および戻り値は、vRealize Orchestrator Client で構成された入力パラメータと戻り値のタイプに対応している必要があります。たとえば、スクリプトの vRAUrl 入力に対しては、vRealize Orchestrator Client に vRAUrl という対応する入力パラメータが必要になります。同様に、スクリプトによって文字列の値を返す場合は、vRealize Orchestrator Client で構成される戻り値のタイプも文字列タイプにする必要があります。アクションによって複雑なオブジェクトを返す場合は、Properties または Composite Type 戻り値のタイプを使用できます。

5 Python requests モジュールをインストールします。

重要： サードパーティの依存モジュールは、メインの vro-python-vra スクリプト フォルダのルート レベルのフォルダにインストールする必要があります。この使用事例では、requests モジュールに lib フォルダを作成します。

a lib フォルダを作成します。

```
mkdir lib
```

b requests モジュールをインストールします。

```
pip3 install requests -t lib/
```

6 requests モジュールを handler.js スクリプトに追加します。

```
import requests

def handler(context, inputs):
    print('Hello, your inputs were ' + inputs)
    return None
```

7 vRealize Automation プロジェクト API に対する GET 要求を作成します。

```
token = ''
vRAUrl = ''
r = requests.get(vRAUrl + '/iaas/api/projects', headers={'Authorization': 'Bearer ' +
token})


print('Got response ' + r.text)
```

8 `token` と `vRAUrl` の値を定義します。

a vRealize Automation ID サービス API を使用してアクセス トークンを取得します。vRealize Automation API のアクセス トークンを取得を参照してください。

b `vRAUrl` 値には、同じ名前の vRealize Orchestrator 入力パラメータを使用するようにスクリプトを定義します。

```
vRAUrl = inputs["vRAUrl"]
```

c `handler.py` ファイルに新しい値を追加します。

```
import requests

def handler(context, inputs):
    token = 'ACCESS_TOKEN'
    vRAUrl = inputs["vRAUrl"]

    r = requests.get(vRAUrl + '/iaas/api/projects', headers={'Authorization': 'Bearer
' + token})

    print('Got response ' + r.text)

    return r.json()
```

注： vRealize Automation プロジェクト API からの応答は JSON 形式で返されるため、vRealize Orchestrator アクションでは `Properties` または `Composite Type` の戻り値タイプを使用します。

9 requests モジュールの `handler.py` ファイルと、`lib` フォルダを含む ZIP パッケージを作成します。

```
zip -r --exclude=*.zip -X vro-python-vra.zip .
```

次のステップ

vRealize Orchestrator アクションに PowerShell スクリプトをインポートします。vRealize Orchestrator クライアントでのアクションの作成を参照してください。

vRealize Automation プロジェクト API を呼び出す Node.js スクリプトの作成

Node.js を使用して vRealize Automation プロジェクト API を呼び出すサンプル スクリプトを作成します。

前提条件

Node.js 14 をダウンロードしてインストールします。Node.js のダウンロードを参照してください。

手順

1 ローカル マシンで、コマンドライン シェルを開きます。

2 vro-node-vra フォルダを作成します。

```
mkdir vro-node-vra
```

3 vro-node-vra フォルダに移動します。

```
cd vro-node-vra
```

4 handler.js という Node.js スクリプトを作成します。

```
touch handler.js
```

handler.js スクリプトでは、2 つの引数を受け入れる 1 つの関数と vRealize Orchestrator ワークフローの実行のコンテキスト、およびバインドされた vRealize Orchestrator の入力を定義する必要があります。

```
exports.handler = (context, inputs) => {
    console.log('Hello, your inputs were ' + inputs);
    return null;
}
```

注： 標準のログ ライブラリを使用すると、このスクリプトを使用するアクションで記録されたすべてがワークフロー ログにも表示されます。スクリプトの入力値および戻り値は、vRealize Orchestrator Client で構成された入力パラメータと戻り値のタイプに対応している必要があります。たとえば、スクリプトの vRAUrl 入力に対しては、vRealize Orchestrator Client に vRAUrl という対応する入力パラメータが必要になります。同様に、スクリプトによって文字列の値を返す場合は、vRealize Orchestrator Client で構成される戻り値のタイプも文字列タイプにする必要があります。アクションによって複雑なオブジェクトを返す場合は、Properties または Composite Type 戻り値のタイプを使用できます。

5 Node.js request モジュールをインストールします。

```
npm install request
```

重要： サードパーティの依存モジュールは、メインの vro-node-vra スクリプト フォルダのルート レベルの node_modules フォルダにインストールする必要があります。このフォルダは、移動したり名前を変更したりしないでください。

6 request モジュールを handler.js スクリプトに追加します。

```
const request = require('request');

exports.handler = (context, inputs) => {
    console.log('Hello, your inputs were ' + inputs);
    return null;
}
```

7 vRealize Automation プロジェクト API に対する GET 要求を作成します。

```
const token = '';
const vRAUrl = '';
```

```
request.get(vRAUrl + '/iaas/api/projects', { 'auth': { 'bearer': token } }, function
(error, response, body) {
    console.log('Got response ' + body);
});
```

8 `token` と `vRAUrl` の値を定義します。

a vRealize Automation ID サービス API を使用してアクセス トークンを取得します。vRealize Automation API のアクセス トークンを取得を参照してください。

b `vRAUrl` 値には、同じ名前の vRealize Orchestrator 入力パラメータを使用するようにスクリプトを定義します。

```
const vRAUrl = inputs.vRAUrl;
```

c 新しい値を `handler.js` ファイルに追加します。

```
const request = require('request');
exports.handler = (context, inputs, callback) => {
    const vRAUrl = inputs.vRAUrl;
    const token = 'ACCESS_TOKEN';
    request.get(vRAUrl + '/iaas/api/projects', { 'auth': { 'bearer': token } },
function (error, response, body) {
        console.log('Got response ' + body);
        callback(null, JSON.parse(body));
    });
}
```

注： vRealize Automation プロジェクト API からの応答は JSON 形式で返されるため、vRealize Orchestrator アクションでは `Properties` または `Composite Type` の戻り値タイプを使用します。

9 request モジュールの `handler.js` ファイルと、`node_modules` フォルダを含む ZIP パッケージを作成します。

```
zip -r --exclude=*.zip -X vro-node-vra.zip .
```

次のステップ

vRealize Orchestrator アクションに Node.js スクリプトをインポートします。vRealize Orchestrator クライアントでのアクションの作成を参照してください。

vRealize Automation プロジェクト API を呼び出す PowerShell スクリプトの作成

PowerShell を使用して vRealize Automation プロジェクト API を呼び出すサンプル スクリプトを作成します。

手順

1 ローカル マシンで、コマンドライン シェルを開きます。

2 vro-powershell-vra フォルダを作成します。

```
mkdir vro-powershell-vra
```

3 vro-powershell-vra フォルダに移動します。

```
cd vro-powershell-vra
```

4 handler.ps1 という PowerShell スクリプトを作成します。

```
touch handler.ps1
```

handler.ps1 スクリプトでは、2 つの引数を受け入れる 1 つの関数と vRealize Orchestrator ワークフローの実行のコンテキスト、およびバインドされた vRealize Orchestrator の入力を定義する必要があります。

```
function Handler {
  Param($context, $inputs)


  $inputsString = $inputs | ConvertTo-Json -Compress
  Write-Host "Inputs were $inputsString"
}
```

注： 標準のログ ライブラリを使用すると、このスクリプトを使用するアクションで記録されたすべてがワークフロー ログにも表示されます。スクリプトの入力値および戻り値は、vRealize Orchestrator Client で構成された入力パラメータと戻り値のタイプに対応している必要があります。たとえば、スクリプトの vRAUrl 入力に対しては、vRealize Orchestrator Client に vRAUrl という対応する入力パラメータが必要になります。同様に、スクリプトによって文字列の値を返す場合は、vRealize Orchestrator Client で構成される戻り値のタイプも文字列タイプにする必要があります。アクションによって複雑なオブジェクトを返す場合は、Properties または Composite Type 戻り値のタイプを使用できます。

5 PowerShell の assert モジュールをインストールします。

重要： サードパーティの依存モジュールは、メインの vro-powershell-vra スクリプト フォルダのルートレベルのフォルダにインストールする必要があります。この使用事例では、assert モジュールに Modules フォルダを作成します。

a Modules フォルダを作成します。

```
mkdir Modules
```

b assert モジュールをインストールします。

```
pwsh -c "Save-Module -Name Assert -Path ./Modules/ -Repository PSGallery"
```

6 assert モジュールを handler.ps1 スクリプトに追加します。

```
Import-Module Assert

function Handler {
```

```
  Param($context, $inputs)


  $inputsString = $inputs | ConvertTo-Json -Compress
  Write-Host "Inputs were $inputsString"
}
```

7 `Invoke-RestMethod` コマンドレットを使用して vRealize Automation プロジェクト API に対する GET 要求を作成します。

```
$token = ''
$vRAUrl = ''
$projectsUrl = $vRAUrl + "/project-service/api/projects"
$response = Invoke-RestMethod $projectsUrl + '/iaas/api/projects' -Headers
@{'Authorization' = "Bearer $token"} -Method 'GET'

Write-Host "Got response: $response"
```

8 `token` と `vRAUrl` の値を定義します。

a vRealize Automation ID サービス API を使用してアクセス トークンを取得します。vRealize Automation API のアクセス トークンを取得を参照してください。

b `Assert-NotNull` および `Assert-Type` assert モジュール属性を追加します。

```
$token | Assert-NotNull
$token | Assert-Type String
```

c `vRAUrl` 値には、同じ名前の vRealize Orchestrator 入力パラメータを使用するようにスクリプトを定義します。

```
$vRAUrl = $inputs.vRAUrl
```

d 新しい値を `handler.ps1` ファイルに追加します。

```
Import-Module Assert
$ErrorActionPreference = "Stop"
function Handler {
  Param($context, $inputs)
  $token = "ACCESS_TOKEN"
  $token | Assert-NotNull
  $token | Assert-Type String
  $vRAUrl = $inputs.vRAUrl
  $projectsUrl = $vRAUrl + "/project-service/api/projects"
  $response = Invoke-RestMethod $projectsUrl -Headers @{'Authorization' = "Bearer
$token"} -Method 'GET'

  Write-Host "Got response: $response"

  return $response
}
```

注： vRealize Automation プロジェクト API からの応答は JSON 形式で返されるため、vRealize Orchestrator アクションでは `Properties` または `Composite Type` の戻り値タイプを使用します。

9 assert モジュールの `handler.ps1` ファイルと、`Modules` フォルダを含む ZIP パッケージを作成します。

```
zip -r --exclude=*.zip -X vro-powershell-vra.zip .
```

次のステップ

vRealize Orchestrator アクションに PowerShell スクリプトをインポートします。vRealize Orchestrator クライアントでのアクションの作成を参照してください。

ワークフローの管理

5

ワークフローとは、順次実行する一連のアクションおよび決定のことです。vRealize Orchestrator は、一般的な管理タスクを実行するワークフローのライブラリを提供します。また vRealize Orchestrator には、ワークフローが実行する個々のアクションのライブラリも用意されています。

ワークフローは、特定の順で実行されたときに、仮想環境において特定のタスクまたは特定のプロセスを完了するアクション、決定、および結果を組み合わせます。ワークフローでは、仮想マシンのプロビジョニング、バックアップ、通常のメンテナンスの実行、メール送信、SSH 操作、物理インフラストラクチャの管理、および他の汎用ユーティリティ操作などのタスクを実行します。ワークフローは、入力を機能に応じて受け付けます。定義されたスケジュールに従って実行するワークフローや、特定の予測イベントが発生した場合に実行するワークフローを作成できます。情報は、ユーザー、他のユーザー、別のワークフローまたはアクションによって提供するか、アプリケーションからの Web サービス呼び出しなどの外部プロセスによって提供できます。ワークフローでは、実行する前に情報の検証およびフィルタリングを行います。

ワークフローは他のワークフローを呼び出すことができます。たとえば、別のワークフローを呼び出して新しい仮想マシンを作成するワークフローを作成できます。

ワークフローは、vRealize Orchestrator Client インターフェイスの統合開発環境 (IDE) を使用して作成できます。この IDE では、ワークフロー ライブラリにアクセスでき、ワークフロー エンジンでワークフローを実行できる機能が用意されています。このワークフロー エンジンでは、vRealize Orchestrator にプラグインした外部ライブラリからのオブジェクトを使用することもできます。この機能を使用すると、プロセスのカスタマイズやサードパーティ アプリケーション提供の機能を実装できます。

この章には、次のトピックが含まれています。

vRealize Orchestrator ワークフロー ライブラリ内の標準ワークフロー

vRealize Orchestrator には、ワークフローの標準ライブラリが付属しています。このライブラリを使用して、仮想インフラストラクチャでの操作を自動化することができます。標準ライブラリ内のワークフローは、読み取り専用の状態でロックされています。標準ワークフローをカスタマイズするには、そのワークフローを複製する必要があります。作成した複製ワークフローまたはカスタム ワークフローは完全に編集可能です。

ワークフロー ライブラリのコンテンツには、HTML5 ベースの vRealize Orchestrator Client の [ライブラリ] - [ワークフロー] メニューからアクセスできます。クライアントの標準およびカスタム ワークフローは、両方ともタグを使用して設定されます。たとえば、ワークフロー ライブラリの検索ボックスに **SSH** と入力して、[キー ペアの作成] ワークフローにアクセスすることができます。

注： ワークフローを複製しない限り、新しいタグを標準ワークフローに追加することはできません。

vRealize Orchestrator クライアントでのワークフローの作成

vRealize Orchestrator Client を使用して、ワークフローを作成および編集できます。

手順

1. vRealize Orchestrator Client にログインします。
2. [ライブラリ] - [ワークフロー] を選択します。
3. [新しいワークフロー] をクリックします。
4. 新しいワークフローの名前を入力し、[作成]をクリックします。
5. ワークフロー エディタを使用して、ワークフローの変数、入力と出力、スキーマ構造、およびプレゼンテーションを構成できます。
6. ワークフローの編集を終了するには、[保存] をクリックします。

   注： [バージョン履歴] タブでワークフローへの変更を追跡できます。詳細については、『vRealize Orchestrator オブジェクトのバージョン履歴』を参照してください。

次のステップ

vRealize Orchestrator トークンの再生機能を使用して、ワークフローのパフォーマンスを最適化することができます。詳細については、『vRealize Orchestrator クライアントでのワークフロー トークンの再生の使用』を参照してください。

親ワークフローからのワークフローとアクションの編集

vRealize Orchestrator Client で、親ワークフローから直接ワークフローとアクションを編集します。

子ワークフローとアクションを親ワークフローから直接編集することにより、ワークフローの開発を効率化できます。

前提条件

別のワークフロー、アクション、またはその両方を呼び出すワークフローを作成します。

手順

1 vRealize Orchestrator Client にログインします。

2 [ライブラリ] - [ワークフロー] の順に移動し、ワークフローを選択します。

3 [スキーマ] タブを選択します。

4 オブジェクト タイプに応じて、ワークフロー キャンバスの [ワークフロー要素] または [アクション要素] をダブルクリックします。

5 オブジェクトを編集します。

6 子ワークフローまたはアクションの編集を終了するには、[保存] をクリックします。

7 親ワークフローに戻るには、オブジェクト エディタを閉じます。

vRealize Orchestrator 入力フォーム デザイナ

ワークフローで入力パラメータが必要な場合、ダイアログ ボックスが開かれ、ユーザーはここに必要な値を入力します。このダイアログ ボックスのコンテンツ、レイアウト、およびプレゼンテーションは、入力フォーム デザイナで調整できます。

入力フォーム デザイナは、ワークフロー エディタの [入力フォーム] タブにあります。このデザイナーは、ナビゲーション メニュー、デザイン キャンバス、およびプロパティ メニューで構成されています。左側のメニューからデザイン キャンバスに、入力と汎用要素をドラッグできます。キャンバスでは、入力パラメータの位置を設定し、個別の入力タブに編成して、入力パラメータのプロパティを構成できます。

注： 入力フォーム デザイナのワークフロー エディタの [変数] タブのコンテンツを使用することはできません。[入力/出力] タブのパラメータのみを使用できます。

汎用要素

ドロップダウン メニューやパスワード テキスト ボックスなどの汎用要素を入力フォーム デザイナに追加できます。汎用要素は実際の入力パラメータに対応していませんが、入力パラメータにバインドできます。

vRealize Orchestrator クライアントでのワークフロー入力パラメータ ダイアログ ボックスの作成

入力フォーム デザイナを使用して、[ワークフロー入力パラメータ] ダイアログ ボックスを作成およびカスタマイズできます。

前提条件

入力パラメータの定義済みリストがワークフローに存在することを確認します。

手順

1 vRealize Orchestrator Client にログインします。

2 [ライブラリ] - [ワークフロー] の順に移動します。

3 カスタム ワークフローを選択します。

4 [入力フォーム] タブをクリックします。

5 （オプション） 入力ダイアログ ボックスで使用するタブを作成します。

タブを使用して、ダイアログ ボックスの構造を整理できます。

6 入力パラメータを選択します。

7 入力パラメータのプロパティを編集します。

入力パラメータのプロパティの詳細については、vRealize Orchestrator クライアントの入力パラメータのプロパティを参照してください。

8 （オプション） キャンバスに汎用要素を追加し、入力パラメータにバインドします。

9 （オプション） 入力パラメータに外部検証を追加します。詳細については、『アクションを使用した vRealize Orchestrator ワークフローの入力の検証』を参照してください。

10 [保存] をクリックします。

結果

ワークフロー ダイアログ ボックスのレイアウトを作成し、入力パラメータのプロパティを設定しました。

vRealize Orchestrator クライアントの入力パラメータのプロパティ

パラメータのプロパティを設定すれば、ユーザーが vRealize Orchestrator ワークフローを実行する際に指定する入力パラメータに制約を加えることができます。

vRealize Orchestrator を使用すると、ワークフローで使用される入力パラメータの値を定量化するために使用するパラメータのプロパティを定義できます。定義したパラメータのプロパティは、ユーザーが vRealize Orchestrator ワークフローで指定できる入力パラメータのタイプや値に制限を課します。

パラメータのプロパティは入力パラメータを検証し、入力パラメータ ダイアログ ボックスに表示されるテキスト ボックスの表示を変更します。パラメータのプロパティには、パラメータ間の依存関係を作成できるものもあります。

パラメータのプロパティ	説明
[ラベル]	入力パラメータのラベルを設定します。
[表示タイプ]	入力テキスト ボックスの表示タイプを設定します。
[可視性]	入力パラメータの可視性を設定します。
[読み取り専用]	入力テキスト ボックスを読み取り専用として設定します。
[カスタム ヘルプ]	入力パラメータ Signpost の説明を設定します。
[デフォルト値]	入力パラメータのデフォルト値を設定します。
[ステップ]	数値タイプの入力に使用されます。1 回のクリックあたりの入力パラメータの値の増加量を設定します。
[必須項目]	入力パラメータの値が必須かどうかを設定します。

パラメータのプロパティ	説明
[正規表現]	正規表現を使用して入力を検証します。
[最小値]	パラメータの最小値または長さを設定します。
[最大値]	パラメータの最大値または長さを設定します。
[テキスト ボックスの一致]	入力パラメータの値を、別の入力パラメータの値と一致するように設定します。
[値のソース]	[表示]、[値]、[制約] タブでパラメータ プロパティの値のソースを設定します。 **注：** [外部ソース] を使用して、外部アクションの値をインポートできます。使用可能なアクションのフィルタリングは、パラメータ タイプ別に実行されます。

アクションを使用した vRealize Orchestrator ワークフローの入力の検証

外部アクションを使用して、カスタム ワークフローの入力を検証します。

前提条件

入力パラメータを使用してカスタム ワークフローを作成します。詳細については、『vRealize Orchestrator クライアントでのワークフローの作成』を参照してください。

入力フォーム デザイナを使用して、ワークフロー入力の外部検証を作成できます。外部検証では、入力パラメータ値にエラーが含まれている場合に文字列値を返すアクション スクリプトが使用されます。入力パラメータ値が有効の場合、外部検証は何も返しません。

手順

1 vRealize Orchestrator Client にログインします。

2 検証アクションを作成します。

   a [ライブラリ] - [アクション] の順に移動します。

   b [新規アクション] をクリックします。

   c [サマリ] タブに必要な情報を入力します。

   d 検証アクションの入力パラメータを入力します。

   注： 検証アクションの入力パラメータの名前は、検証されるワークフローの入力パラメータの名前と同じである必要があります。

e [スクリプト] タブに検証アクションのスクリプトを入力します。

```
if (in_1=="invalid") {
                                                    return "in_1 can't be invalid!";
}
                                    if (in_2=="invalid") {
                                                    return "in_2 can't be invalid!";
}

//inputs are valid, return nothing
```

注： 前のスクリプトは単純な例であり、使用可能な検証スクリプトのすべての範囲を表しているわけではありません。

f [保存] をクリックします。

3 外部検証を適用します。

a [ライブラリ] - [ワークフロー] の順に移動します。

b カスタム ワークフローを選択します。

c [入力フォーム] タブを選択します。

d 画面の左上にあるクリップボード アイコンを選択します。

e vRealize Orchestrator 検証要素をキャンバスにドラッグします。

f 検証要素を選択し、検証ラベルを入力して、検証アクションを選択します。

g （オプション） 追加の検証要素を作成します。

h [保存] をクリックします。

4 ワークフローを実行します。

検証でエラーが発生した場合は、文字列を返します。検証に成功すると、検証は何も返さず、ワークフローの実行が続行されます。

結果

これで、カスタム vRealize Orchestrator ワークフローの外部検証が作成されました。

vRealize Orchestrator クライアントでのユーザー操作の要求

ワークフローが完了する前に、追加のユーザー入力を要求できます。

追加のユーザー操作を必要とするワークフローは、ユーザーから必要な入力パラメータを受け取るまで処理を一時停止します。ワークフローは必要な情報を入力できるユーザーを定義し、その定義に応じて操作の要求を送信します。ユーザー入力を待機しているワークフローは、vRealize Orchestrator Client ダッシュボードの [最近のワークフローの実行] パネルおよび右上の通知メニューに表示されます。

vRealize Orchestrator クライアントでのワークフローのスケジュール設定

スケジュール設定を利用すると vRealize Orchestrator ワークフローの実行を自動化できます。

ワークフローの実行をスケジューリングする際は、スケジュール設定タスクを実行する日付、時刻、および実行間隔を設定します。

手順

1 vRealize Orchestrator Client にログインします。

2 [ライブラリ] メニューからワークフローを選択し、ワークフロー パネルで [スケジュール] をクリックします。

3 [全般]、[スケジュール設定]、および [ワークフロー] カテゴリでスケジュール設定されたタスクのパラメータを構成します。

注： [ワークフロー] パラメータ カテゴリは、入力パラメータが必要なワークフローにのみ表示されます。

パラメータ	説明
[名前]	スケジュール設定タスクの名前。
[説明]	スケジュール設定タスクの目的についての簡単な説明。
[開始]	ワークフローの実行がスケジュール設定されている最初の日付および時間。
[過去の場合は開始する]	スケジュール設定されている時間を過ぎてしまっている場合に、ワークフローを開始するかどうかを選択します。[はい] にすると、スケジュール設定されたワークフローはすぐに開始されます。[いいえ] にすると、繰り返しが設定されている次回のスケジュールでワークフローが開始されます。
[スケジュール]	スケジュール設定タスクの繰り返しパターンとイベント トリガ エントリを設定します。
[終了日]	[繰り返しなし] が選択されている場合にのみ表示されます。スケジュール設定されたタスクが終了する日付および時間を設定します。
[ワークフロー]	ワークフローの入力パラメータを設定します。

4 [作成] をクリックします。

結果

これでワークフローのスケジュール設定タスクが作成されました。[アクティビティ] - [スケジュール設定済み] に、スケジュール設定されたワークフローが表示されます。スケジュール設定タスクを削除するには、スケジュール パネルの [削除] をクリックします。

vRealize Orchestrator クライアントでのスケジュール設定タスクの編集

スケジュール設定されたタスクを編集することで、スケジュール設定されたワークフローのパラメータ（日付、時刻、繰り返しなど）を変更できます。

前提条件

スケジュール設定されたワークフロー タスクを作成します。

手順

1 vRealize Orchestrator Client にログインします。

2 [アクティビティ] - [スケジュール設定済み] からスケジュール設定タスクを選択します。

3 ワークフロー パネルで [編集] をクリックします。

4 スケジュールを編集し、[保存] をクリックします。

注： スケジュール設定タスクの作成時には、入力パラメータのセットは読み取り専用となっており、編集できません。これらのパラメータを変更するには、このワークフロー用に新しいスケジュール設定タスクを作成します。

ワークフローでのオブジェクト参照の検索

ワークフロー開発者は、オブジェクト参照情報を使用して、開発ライフサイクルを最適化できます。

vRealize Orchestrator Client では、オブジェクト参照情報を検索することができます。この機能には、次の 2 種類の検索機能があります。

- [依存関係の検索]：ワークフロー内のオブジェクトの依存関係に関する情報を検索します。依存関係には、その他のワークフロー、アクション、リソース要素、および構成要素が含まれます。
- [使用量の検索]：選択したワークフローが vRealize Orchestrator Client ライブラリ内のその他のワークフローで使用されているかどうかを確認します。

オブジェクト参照に関する情報には、ワークフロー エディタから、またはカード ビュー、リスト ビュー、ツリー ビューのいずれかの vRealize Orchestrator Client ライブラリからアクセスできます。vRealize Orchestrator Client ライブラリのさまざまなタイプのコンテンツ編成の詳細については、vRealize Orchestrator Client でのコンテンツの編成を参照してください。

次の手順は、ワークフロー エディタからオブジェクト参照にアクセスする方法を示しています。

前提条件

少なくとも 1 つのオブジェクト参照を含むワークフローを開発します。

手順

1 vRealize Orchestrator Client にログインします。

2 [ライブラリ] - [ワークフロー] の順に移動し、ワークフローを選択します。

3 オブジェクトの依存関係に関する情報を検索するには、[依存関係の検索] をクリックします。

注： 依存関係のポップアップ ウィンドウでは、リストから参照されているオブジェクトを選択できます。オブジェクトを選択すると、選択したオブジェクトの詳細を表示したり編集したりできる別の vRealize Orchestrator Client タブが開きます。

4 選択されたワークフローが使用されている場所に関する情報を検索するには、[使用量の検索] をクリックします。

アクションの管理

6

アクション スクリプトを追加して、vRealize Orchestrator ワークフローを変更できます。

vRealize Orchestrator Client は、事前定義済みのアクションのライブラリとカスタム アクション スクリプトのアクション エディタを提供します。アクションは、ワークフローのビルディング ブロックとして使用する個々の機能を表します。

アクションは JavaScript 機能です。アクションでは複数の入力パラメータを使用でき、戻り値は 1 つになります。アクションでは vRealize Orchestrator API のすべてのオブジェクトのほか、プラグインを使用して vRealize Orchestrator にインポートしたすべての API のオブジェクトを呼び出すことができます。

ワークフローを実行すると、アクションはワークフローの変数から入力パラメータを取得します。これらの変数は、ワークフローの初期入力パラメータか、ワークフロー セット内の他の要素が実行時に設定した変数のいずれかになります。

アクション エディタには、スクリプトのオートコンプリート機能と、使用可能なスクリプト タイプとそのドキュメントを含む API エクスプローラが含まれています。

この章には、次のトピックが含まれています。

- vRealize Orchestrator クライアントでのアクションの作成
- アクションの実行およびデバッグ
- Python、Node.js、PowerShell スクリプトの主要な概念
- Python、Node.js、PowerShell スクリプトのランタイムの制限

vRealize Orchestrator クライアントでのアクションの作成

vRealize Orchestrator Client を使用して、アクション スクリプトを作成、編集、および削除できます。

アクションを作成するときに、次のランタイムを使用できます。

- Python 3.7
- Node.js 14
- PowerCLI 11/Powershell 6.2

- PowerCLI 12.3.0/Powershell 7.1

  注： PowerCLI ランタイムには PowerShell と次のモジュールが含まれています：VMware.PowerCLI、PowerNSX、PowervRA。

前提条件

Python、Node.js、または PowerShell スクリプトの作成前に、これらのランタイムを使用する vRealize Orchestrator 互換のスクリプトを開発するための主要な概念を理解していることを確認します。Python、Node.js、PowerShell スクリプトの主要な概念 を参照してください。

手順

1. vRealize Orchestrator Client にログインします。
2. [ライブラリ] - [アクション] の順に移動します。
3. [新規アクション] をクリックします。
4. [全般] タブで、アクションの名前とモジュール名を入力します。

   注： 名前とモジュール名は、アクションごとに一意である必要があります。アクション名は有効な JavaScript 関数である必要があります。アクション名は 1 つの単語とし、文字、数字、ドル記号 ($) およびアンダースコア (_) のみを使用できます。モジュール名は、ドット (.)で区切った単語で構成する必要があります。

5. （オプション） アクションの説明、バージョン番号、タグ、およびグループ権限を作成します。
6. [スクリプト] タブで、アクションの入力を追加し、出力の戻り値のタイプを選択して、スクリプトを書き込みます。

   注： [タイプ] ドロップダウン メニューから [Zip] を選択すると、外部スクリプト ソースと、該当する場合は依存モジュールをインポートできます。

7. アクションの編集を終了するには、[保存] をクリックします。

   アクションが保存されたことがメッセージに示されます。

次のステップ

vRealize Orchestrator アクションを使用方法の例については、Python を使用して vRealize Orchestrator で Amazon Web Services を統合する方法を参照してください。

アクションの実行およびデバッグ

アクションの実行およびデバッグをアクション エディタから直接実行することで、アクションを改善することができます。

アクションを vRealize Orchestrator クライアントのアクション エディタから直接実行およびデバッグできるようになりました。この機能を使用すると、ワークフローに統合されているアクションが想定どおりに実行されるようになります。

vRealize Orchestrator クライアントでのアクションの実行

ワークフロー デザイナとして、アクションをワークフローに統合する前に実行します。

前提条件

アクションを作成します。vRealize Orchestrator クライアントでのアクションの作成を参照してください。

手順

1 vRealize Orchestrator クライアントにログインします。

2 [ライブラリ] - [アクション] の順に移動し、実行するアクションを選択します。

3 [実行] をクリックします。

4 要求された入力パラメータを指定し、[実行] をクリックします。

アクションの実行が完了したら、[結果/入力] タブをクリックします。アクションの実行でエラーが発生した場合は、このタブに赤で表示されます。アクション実行の詳細は、[アクション結果] 要素から表示できます。

注： アクション実行の結果は保存されません。

vRealize Orchestrator クライアントでのアクションのデバッグ

ワークフロー デザイナとして、スクリプトにブレークポイントを挿入してアクションをデバッグします。

vRealize Orchestrator には、アクションのスクリプトと入力プロパティのデバッグに使用できる組み込みデバッグツールが含まれています。デバッグ プロセスをアクション エディタで開始するには、アクションのスクリプト行にブレークポイントを挿入します。

注： 組み込みデバッグ ツールは、デフォルトの JavaScript ランタイムを使用するアクションでのみ動作します。別のランタイムを使用するアクション スクリプトをデバッグする方法の例については、Amazon Web Services アクションのデバッグを参照してください。

前提条件

アクションを作成します。vRealize Orchestrator クライアントでのアクションの作成を参照してください。

手順

1 vRealize Orchestrator クライアントにログインします。

2 [ライブラリ] - [アクション] の順に移動し、デバッグするアクションを選択します。

3 アクション エディタで、デバッグするアクション スクリプトの行にブレークポイントを追加します。

4 [デバッグ] をクリックします。

5 アクションの入力パラメータを入力し、[実行] をクリックします。

デバッグ モードでのアクションの実行が開始されます。

6 ブレークポイントに到達してアクションの実行が中断されたら、以下のいずれかのオプションを選択します。

オプション	説明
続行	別のブレークポイントに到達するか、アクションの実行が完了するまで、アクションの実行を再開します。
ステップイン	現在のアクション関数にステップインします。デバッガが関数の現在の行の中を進行できない場合は、[ステップ オーバー] 処理が実行されます。
ステップ オーバー	デバッガは、現在の関数の次の行に進みます。
ステップ リターン	デバッガは、現在の関数が戻るときに実行される行に移動します。

7 （オプション）[デバッガ] タブで、式を追加します。

8 （オプション）[デバッガ] タブで、変数の値を編集します。

Python、Node.js、PowerShell スクリプトの主要な概念

vRealize Orchestrator で使用するスクリプトを作成する際には、スクリプトの構造とフォーマットが正しいことを確認する必要があります。

サポート対象のランタイム

vRealize Orchestrator アクションとワークフローを開発する場合は、次のランタイムを使用できます。

- Python 3.7
- Node.js 14
- PowerCLI 11/Powershell 6.2
- PowerCLI 12.3.0/Powershell 7.1

> **注：** PowerCLI ランタイムには PowerShell と次のモジュールが含まれています：VMware.PowerCLI、PowerNSX、PowervRA。

新しいランタイムには任意のカスタム ソース コードを追加できますが、コンテキストと入力を受け入れ、結果を vRealize Orchestrator エンジンとの間で受け渡しするには、適切な関数フォーマットを使用する必要があります。

スクリプト作成の推奨事項

スクリプト作成タスクを簡素化するために、[スクリプト可能タスク] 要素をワークフローのスキーマに追加できます。vRealize Orchestrator アクションを使用して、より複雑なスクリプト作成タスクを実行できます。

アクションを使用することによるメリットは次の 2 つです。

- ワークフローとは別にアクションを作成、更新、インポート、およびエクスポートできます。
- アクションはスタンドアローンのオブジェクトで、独自の環境で実行およびデバッグが可能なため、開発プロセスをスムーズに進めることができます。アクションの実行およびデバッグ を参照してください。

スクリプト関数の要件

スクリプト関数のデフォルト名は **handler** です。関数は、コンテキストと入力の 2 つの引数を受け入れます。コンテキストは、システム情報を含むマップ オブジェクトです。たとえば、`vroURL` には、呼び出す vRealize Orchestrator インスタンスの URL を含めることができます。また、`executionId` には、ワークフロー実行のトークン ID を含めます。

入力は、アクションに提供されるすべての入力を含むマップ オブジェクトです。たとえば、`myInput` というアクションで入力を定義した場合、ランタイムに応じて、`inputs.myInput` や `inputs["myInput"]` のような入力引数からアクセスできます。関数から返るものはすべて、アクションの結果です。したがって、アクションの戻り値のタイプは、vRealize Orchestrator でスクリプトが返すコンテンツのタイプに対応している必要があります。プリミティブ型の数値を返す場合は、アクションの戻り値のタイプを数値タイプにする必要があります。文字列を返す場合、アクションの戻り値のタイプは文字列タイプにする必要があります。複雑なオブジェクトを返す場合は、`Properties` または `Composite Type` のいずれかに戻り値のタイプをマッピングする必要があります。アレイにも同様の原則が適用されます。

Python、Node.js、PowerShell ランタイムでサポートされている入力および出力パラメータ タイプ：

- `String`
- `Number`
- `Boolean`
- `Date`
- `Properties`
- `Composite Type`

エントリ ハンドラの定義

デフォルトでは、エントリ ハンドラの値は `handler.handler` です。この値は、vRealize Orchestrator エンジンが、`handler` という関数を含む `handler.py`、`handler.js` または `handler.ps1` という ZIP パッケージの最上位レベルのファイルを検索することを意味します。関数とハンドラ ファイルの名前の違いは、エントリ ハンドラの値に反映させる必要があります。たとえば、メイン ハンドラが `index.js` で、関数が `callMe` という名前だった場合、エントリ ハンドラの値は **`index.callMe`** に設定する必要があります。

外部 IDE でのランタイム スクリプトのデバッグ

vRealize Orchestrator は、外部 IDE での Python および Node.js スクリプトのデバッグをサポートしています。外部 IDE で PowerShell スクリプトをデバッグすることはできません。

Python、Node.js、PowerShell スクリプトのランタイムの制限

一部の Python、Node.js、または PowerShell スクリプトでは、vRealize Orchestrator Client のメモリとタイムアウトの値の変更が必要になる場合があります。

vRealize Orchestrator Client は、Python、Node.js、および PowerShell アクション スクリプトで、メモリとタイムアウトに以下のデフォルト値を使用します。

- メモリ：64 MB
- タイムアウト：180 秒

アクション スクリプトがこれらのデフォルト値のいずれかまたは両方を超えると、アクションの実行は失敗します。たとえば、アクション スクリプトでサードパーティの依存モジュールを複数使用している場合です。このようなシナリオでは、デフォルトの 64 MB のメモリ制限は十分ではない可能性があります。

リソース不足によるアクション実行の失敗を回避するには、アクション エディタからメモリとタイムアウトの値を変更します。

注： スクリプトを、スクリプト実行が可能な複数のタスク要素に分割し、ワークフローに追加することも検討できます。

手順

1. vRealize Orchestrator Client にログインします。
2. [ライブラリ] - [アクション] の順に移動し、アクションを選択します。
3. [スクリプト] タブを選択します。
4. [ランタイム制限] で、メモリとタイムアウトの値を変更します。
5. [保存] をクリックします。
6. 新しいランタイム制限をテストするには、[デバッグ] をクリックします。

構成要素の管理

7

構成要素は、展開した vRealize Orchestrator サーバ全体で、定数の構成に使用できる変数のリストです。

構成要素を使用して、vRealize Orchestrator サーバで実行されているすべてのワークフロー、アクション、およびポリシーで変数を使用できるようにすることができます。

構成要素の変数を使用するワークフロー、アクション、またはポリシーを含むパッケージを作成すると、vRealize Orchestrator は、パッケージ内にその構成要素を自動的に含めます。構成要素を含むパッケージを別の vRealize Orchestrator サーバにインポートすると、構成要素の変数値もインポートすることができます。たとえば、実行元の vRealize Orchestrator サーバによって異なる変数値が必要なワークフローを作成する場合に、構成要素内でこれらの変数を設定すると、そのワークフローをエクスポートして、別の vRealize Orchestrator サーバで使用することができます。このため、構成要素を使用すると、サーバ間でワークフロー、アクション、ポリシーをより簡単に交換できます。

注： 5.1 以前の vRealize Orchestrator からエクスポートされた構成要素から、構成要素の変数値をインポートすることはできません。

この章には、次のトピックが含まれています。

vRealize Orchestrator クライアントでの構成要素の作成

構成要素を使用すると、vRealize Orchestrator サーバ内で共通の変数を設定できます。サーバで実行中のすべての要素は、構成要素で設定した変数を使用することができます。

手順

1. vRealize Orchestrator Client にログインします。
2. [アセット] - [構成] の順に移動します。
3. [新しい構成] を選択します。
4. 構成要素名を入力します。
5. [変数] タブを選択します。

6 ローカル変数を作成するには、[新規] をクリックします。

a 変数名を入力します。

b 変数タイプを選択します。

注： 構成変数の配列を作成するには、[配列] チェックボックスを選択します。

c （オプション） 構成変数の値を入力します。

d [保存] をクリックします。

7 構成要素の作成を完了するには、[保存] をクリックします。

次のステップ

構成要素を使用して、ワークフロー、アクション、またはポリシーに変数を提供することができます。

ポリシーの管理

8

ポリシーは、システムのアクティビティを監視するイベント トリガです。ポリシーは、特定の vRealize Orchestrator オブジェクトのステータスまたはパフォーマンスの変更によって発行された、事前定義済みイベントに応答します。

ポリシーとは、特定の事前定義済みイベントが vRealize Orchestrator または vRealize Orchestrator がプラグインを介してアクセスするテクノロジーにおいて発生したときに、特定のワークフローまたはスクリプトを実行する一連のルール、ゲージ、しきい値、およびイベント フィルタです。vRealize Orchestrator は、ポリシーの実行中、常にポリシー ルールを評価します。たとえば、`VC:HostSystem` および `VC:VirtualMachine` タイプの vCenter Server オブジェクトの動作を監視するポリシー ゲージおよびしきい値を実装できます。

この章には、次のトピックが含まれています。

vRealize Orchestrator クライアントでのポリシーの作成および適用

ポリシーを使用して、特定のイベントに対する vRealize Orchestrator システムの動作を監視することができます。

手順

1 vRealize Orchestrator Client にログインします。

2 [ライブラリ] - [ポリシー] の順に移動します。

3 [新規ポリシー] を選択します。

空のポリシーが作成されました。

4 ポリシー名とバージョン番号を入力します。

5 [変数] タブを選択します。

6 ローカル変数を作成するには、[新規] をクリックします。

a 変数名を入力します。

b 変数タイプを選択します。

注： ポリシー変数の配列を作成するには、[配列] チェックボックスを選択します。

c 変数の値を入力します。

注： 構成要素の変数の値をインポートするには、[構成にバインド] を使用できます。

d [保存] をクリックします。

7 [定義] タブでは、ポリシー要素の追加とイベント ハンドラの設定を行うことができます。

ポリシー要素の詳細については、vRealize Orchestrator クライアントのポリシー要素を参照してください。

8 [保存] をクリックします。

ポリシーが構成されました。

次のステップ

ポリシーを開始するには、ポリシーを選択し、[実行] をクリックします。ポリシーの実行名と、必要な入力パラメータ（要求された場合）を入力します。

ポリシーのステータスを表示するには、[アクティビティ] - [ポリシーの実行] の順に移動します。

vRealize Orchestrator クライアントのポリシー要素

ポリシー要素を使用して、イベントが発生したときに事前定義された vRealize Orchestrator ワークフローまたはスクリプトを実行できます。

オブジェクトによってトリガされたイベントに対する応答としてワークフローまたはスクリプト実行をトリガするためのポリシー要素を追加できます。定期イベント要素を使用して、ワークフローまたはスクリプト実行をスケジュール設定できます。ルート要素を使用して、ポリシーの開始または停止の動作を設定できます。ポリシー要素には、ポリシー要素を実行する必要があるタイミングを定義するイベント ハンドラを含めることができます。

注： ポリシー要素を有効にするイベント ハンドラは、ワークフローまたはアクション スクリプトのいずれかになります。ワークフローとスクリプトの両方をイベント ハンドラに追加すると、ポリシーによってスクリプト トリガが無視され、ワークフロー トリガのみが使用されます。

イベント ハンドラ	説明
[OnInit]	ポリシー要素は、ポリシーを開始するたびにトリガされます。
[OnExit]	ポリシー要素は、ポリシーを停止するたびにトリガされます。
[OnExecute]	定期イベント要素によって使用されます。定期イベント要素で指定された時間内にポリシー要素をトリガします。

注： vRealize Orchestrator データベースにプラグインされたテクノロジーは、一意のイベント ハンドラを保持できます。たとえば、SNMP プラグインを使用すると、SNMP ベースのポリシー要素を作成するときに、[OnTrap] イベント ハンドラを使用できます。

ポリシー要素は、[ポリシー編集] ウィンドウの [定義] タブで構成されます。

vRealize Orchestrator クライアントでのポリシーの実行の管理

vRealize Orchestrator Client を使用して、ポリシーの優先度と、vRealize Orchestrator サーバが再起動されたときのポリシーのサーバ起動時の動作を管理できます。

前提条件

ポリシーを作成して実行します。詳細については、『vRealize Orchestrator クライアントでのポリシーの作成および適用』を参照してください。

手順

1. vRealize Orchestrator クライアントに管理者としてログインします。
2. [アクティビティ] - [ポリシー実行] に移動します。
3. 管理するポリシー実行をクリックします。
4. [停止] をクリックします。

   ポリシーの状態が [停止] に変更されます。
5. [全般] タブで、ポリシーの優先度とサーバ起動時の動作を設定します。
6. ポリシーを再起動するには、[実行] をクリックします。

   ポリシーの状態が [実行中] に変更されます。

リソース要素の管理

9

ワークフローでは、vRealize Orchestrator とは独立して作成したオブジェクトを属性として使用することができます。ワークフロー内で外部オブジェクトを属性として使用するには、そのオブジェクトをリソース要素としてサーバにインポートします。

vRealize Orchestrator ワークフロー内でリソース要素として使用できるオブジェクトには、イメージ ファイル、スクリプト、XML テンプレート、HTML ファイルなどがあります。vRealize Orchestrator サーバ上で実行されるワークフローの場合、vRealize Orchestrator にインポートされたすべてのリソース要素を使用することができます。

オブジェクトをリソース要素として vRealize Orchestrator にインポートすることにより、そのオブジェクトを 1 か所で変更し、そのリソース要素を使用しているすべてのワークフローに対して変更内容を自動的に反映させることができます。

1 つのリソース要素の最大サイズは 16 MB です。

リソース要素には、インポート、エクスポート、リストア、更新、および削除の操作を行うことができます。

パッケージの管理

10

vRealize Orchestrator Client を使用してパッケージを作成、エクスポート、およびインポートします。パッケージを利用すれば、ワークフロー オブジェクトをエクスポートして他の vRealize Orchestrator インスタンスで使用できます。

パッケージには、ワークフロー、アクション、ポリシー、構成要素、またはリソース要素を含めることができます。

パッケージに要素を追加すると、vRealize Orchestrator によって依存関係がチェックされ、従属要素がパッケージに追加されます。たとえば、アクションまたは他のワークフローが使用されるワークフローを追加した場合は、vRealize Orchestrator によってそのアクションと他のワークフローがパッケージに追加されます。

パッケージをインポートすると、サーバによって、コンテンツのさまざまな要素のバージョンと、対応するローカル要素のバージョンが比較されます。この比較により、ローカル要素とインポートされた要素のバージョンの違いが明確になります。ユーザーは、パッケージをインポートするか、特定の要素を選択してインポートするかを判断できます。

vRealize Orchestrator Client で作成されるほとんどのオブジェクト（リソース要素を除く）の場合、パッケージがこれらのオブジェクトをエクスポートおよびインポートする唯一の方法です。

パッケージはデジタル著作権管理を使用して、受信側サーバでのパッケージのコンテンツの使用方法を制御できます。パッケージは vRealize Orchestrator によって署名され、データ保護のために暗号化されます。パッケージは、X509 証明書を使用して、度のユーザーが要素のエクスポートや再配布を行うかを追跡できます。

vRealize Orchestrator クライアントでのパッケージの作成

パッケージ内のワークフロー、ポリシー、アクション、プラグイン参照、リソース要素、および構成要素をエクスポートおよびインポートできます。パッケージ オブジェクトに関連するすべての依存要素は、パッケージに自動的に追加され、バージョン間の互換性が確保されます。依存要素を削除するには、最初に関連するパッケージ オブジェクトを削除する必要があります。

vRealize Orchestrator Client で作成されるほとんどのオブジェクト（リソース要素を除く）の場合、パッケージがこれらのオブジェクトをエクスポートおよびインポートする唯一の方法です。

前提条件

パッケージに追加することができるワークフロー、アクション、ポリシーなどのオブジェクトが vRealize Orchestrator サーバに含まれていることを確認します。

手順

1 vRealize Orchestrator Client にログインします。

2 [資産] - [パッケージ] に移動します。

3 [新規パッケージ] をクリックします。

4 [全般] タブでパッケージの名前と説明を入力します。

注： vRealize Orchestrator Client でのパッケージ名には特殊文字を使用することはできません。

5 [コンテンツ] タブで [追加] をクリックします。

6 パッケージに追加するオブジェクトを選択し、[追加] をクリックします。

注： 依存要素はパッケージに自動的に追加されますが、パッケージの作成中に [コンテンツ] タブには表示されません。依存要素を表示するには、パッケージの作成後に [コンテンツ] タブを選択します。

7 パッケージの作成を完了するには、[作成] をクリックします。

vRealize Orchestrator クライアントでのパッケージのエクスポート

vRealize Orchestrator Client を使用して、パッケージを別の vRealize Orchestrator 環境にエクスポートできます。

前提条件

エクスポートする vRealize Orchestrator オブジェクトを含むパッケージを作成します。詳細については、『vRealize Orchestrator クライアントでのパッケージの作成 』を参照してください。

手順

1 vRealize Orchestrator Client にログインします。

2 [資産] - [パッケージ] に移動します。

3 パッケージで、[エクスポート] をクリックします。

4 （オプション） 他のエクスポート オプションを選択します。

オプション	説明
パッケージに構成の属性値を追加する	構成要素の属性値をエクスポートします。
パッケージに構成の SecureString 属性値を追加する	`SecureString` 構成属性値をエクスポートします。
パッケージにグローバル タグを追加する	グローバル タグをエクスポートします。

5 パッケージをインポートするユーザーのアクセス権限を設定します。

オプション	説明
コンテンツの表示	ユーザーは、パッケージのコンテンツを表示できます。
パッケージへの追加	ユーザーは、インポートされたパッケージのコンテンツを他のパッケージに追加できます。
コンテンツの編集	ユーザーは、パッケージのコンテンツを編集できます。

6 [OK] をクリックします。

注： 拡張子が `.package` のファイルは、ローカル マシン上のデフォルト フォルダに保存されます。カスタム フォルダを設定するには、ブラウザのストレージ設定を変更します。

結果

パッケージがエクスポートされます。エクスポートされたオブジェクトを別の vRealize Orchestrator 環境で使用できるようになりました。

vRealize Orchestrator クライアントでのパッケージのインポート

vRealize Orchestrator Client を使用して、ワークフロー パッケージをインポートします。パッケージをインポートすることで、ある vRealize Orchestrator サーバのオブジェクトを別のサーバで再利用できます。

前提条件

- 変更した標準の vRealize Orchestrator オブジェクトをすべてバックアップします。
- リモート サーバ上で、インポートするオブジェクトが含まれるパッケージを作成し、エクスポートします。

手順

1 vRealize Orchestrator Client にログインします。

2 [資産] - [パッケージ] に移動します。

3 [インポート] をクリックします。インポートする `.package` ファイルを参照し、[開く] をクリックします。

4 インポートされたパッケージの情報を確認します。

   a [全般] タブには、名前、説明、パッケージに含まれる項目数などの、インポートされるパッケージに関する情報と、証明書の情報が表示されます。

   ファイルをインポートする前に、移行元 vRealize Orchestrator インスタンスの発行者証明書を信頼するよう求められる場合があります。

   b [パッケージ要素] タブには、インポート ファイルに含まれるオブジェクトが一覧で表示されます。パッケージ内のオブジェクトのバージョンが、サーバ上のバージョンより新しい場合は、インポート対象としてそのオブジェクトのバージョンが自動的に選択されます。前のバージョンの vRealize Orchestrator 要素は、手動で選択する必要があります。

c パッケージから構成要素の属性値をインポートしない場合は、[構成の属性値をインポートする] の選択を解除します。

d ドロップダウン メニューから、タグをインポートするかどうかを選択します。

5 [インポート] をクリックします。

vRealize Orchestrator クライアントのトラブルシューティング

11

メトリック、トークンの再生、検証、およびデバッグを使用して vRealize Orchestrator インスタンスのトラブルシューティングおよび監視を行うことができます。

この章には、次のトピックが含まれています。

vRealize Orchestrator クライアントのメトリック データ

vRealize Orchestrator 管理者は、ワークフロー プロファイルおよびシステム ダッシュボードのメトリックを使用して、vRealize Orchestrator システムとワークフローのトラブルシューティングを行うことができます。

プロファイル機能は、ワークフローの実行に関するメトリック データを収集します。ワークフローのプロファイルはデフォルトで有効になっています。自動プロファイルは、[コントロール センター] - [拡張機能のプロパティ] - [profiler-8.7.0] で無効にできます。

vRealize Orchestrator Client のメトリック データのその他のソースは、システム レベルのメトリックを提供するシステム ダッシュボードです。詳細については、『vRealize Orchestrator システム ダッシュボードの使用』を参照してください。

vRealize Orchestrator クライアントでのワークフローのプロファイル

ワークフローの実行をプロファイルして、vRealize Orchestrator のトラブルシューティングや最適化を行うことができます。

vRealize Orchestrator Client のプロファイル機能を使用すると、ワークフローの実行に関する有用なメトリック データを収集できます。このデータを使用してワークフローのパフォーマンスを最適化できます。デフォルトでは、ワークフローの実行は自動的にプロファイルされます。vRealize Orchestrator コントロール センターの [拡張機能プロパティ] ページで自動プロファイルを無効にして、手動でプロファイラを実行できます。手動でプロファイルを実行するには、ライブラリ内でワークフローを見つけ、[アクション] - [プロファイル] を選択します。

前提条件

ワークフローの実行

手順

1 vRealize Orchestrator Client にログインします。

2 [アクティビティ] - [ワークフローの実行] の順に移動します。

3 ワークフローの実行を選択します。

ワークフローの実行スキーマでは、個々のワークフロー アイテムに関するデータを表示できます。データには、合計実行時間、最大実行時間、およびアイテムの実行数が含まれます。この情報は、ページの右上にあるドロップダウン メニューからフィルタできます。

4 [パフォーマンス] タブを選択します。

このタブには、ワークフロー実行の CPU 時間、実行時間、トークン サイズ、およびワークフロー アイテム データに関するメトリック データが表示されます。

注： ワークフローが追加入力を待機しているときなど、ワークフローの実行が一時停止している場合、CPU 時間メトリックは完了前に発生するランタイム スレッドのみを取得します。

次のステップ

プロファイルから収集したデータを使用して、ワークフローを最適化します。

vRealize Orchestrator システム ダッシュボードの使用

管理者は、vRealize Orchestrator Client システム ダッシュボードを使用して、vRealize Orchestrator 環境のノードに関する有用なメトリック データを収集できます。

システム ダッシュボードには、vRealize Orchestrator Client ダッシュボード ページの上部にある [システム] タブをクリックしてアクセスできます。提供されるデータは次のとおりです。

- ノードのステータス
- ノードのプロパティ
- クラスタ設定クラスタ設定は、システム ダッシュボードからのみ表示できます。これらの設定を変更するには、vRealize Orchestrator コントロール センターの [Orchestrator クラスタ管理] ページに移動します。
- スレッド情報
- ヒープ メモリ
- 非ヒープ メモリ
- ファイル システムの使用状況
- 認証データ
- Orchestrator データベース接続プール
- プロセスの入力引数

このデータは、vRealize Orchestrator 環境の個々のノードの状態を監視し、問題のトラブルシューティングに使用できます。個々のノード間を移動するには、システム ダッシュボードの上部にある、ノードに関連付けられているタブをクリックします。

vRealize Orchestrator クライアントでのワークフロー トークンの再生の使用

トークンの再生機能を使用して、ワークフロー実行中のアイテムの移行を表示できます。

トークンの再生機能は、ワークフロー アイテム間のそれぞれの移行のコンテキスト情報を記録します。ワークフロー アイテムごとに、トークンの再生機能はワークフローの実行の開始と終了、およびワークフロー アイテムの実行の終了時に変更された変数を記録します。トークンの再生では、各ワークフロー アイテム対して生成されたスクリプト ログ メッセージも参照されます。

注： ワークフロー アイテムの移行に関するデータは、vRealize Orchestrator PostgreSQL データベースに保存されます。このデータは、ワークフローの実行が削除されるとデータベースから削除されます。

前提条件

- コントロール センターからトークンの再生機能を有効にします。
  a コントロール センターに root としてログインします。
  b [拡張機能のプロパティ] を選択します。
  c [tokenreplay-8.7.0] をクリックします。
  d トークンの再生機能を有効にするには、[有効化] をクリックします。
  e [保存] をクリックします。

  注： vRealize Orchestrator サーバが拡張機能を更新するには、最大 5 分間かかることがあります。
- ワークフローの実行。

注： デフォルトでは、vRealize Orchestrator サーバでのすべてのワークフローの実行において、トークンの再生は自動的に実行されません。各ワークフローのトークンの再生を個別に実行することも、コントロール センターの [拡張機能のプロパティ] ページからすべてのワークフローのトークンの再生の拡張機能を有効にすることもできます。

手順

1 （オプション）vRealize Orchestrator サーバでのすべてのワークフローの実行において、トークンの再生を有効にします。

  注： コントロール センターから機能を有効にせずに個々のトークンの再生を実行するには、ワークフロー エディタ ページの [再生で実行] をクリックします。

  a コントロール センターに root としてログインします。
  b [拡張機能のプロパティ] を選択します。

c [tokenreplay-8.7.0] をクリックします。

d すべてのワークフローでトークンの再生機能を有効にするには、[すべてのワークフローの実行について記録再生] が有効になっていることを確認します。

e [保存] をクリックします。

注： vRealize Orchestrator サーバが拡張機能を更新するには、最大 5 分間かかることがあります。

2 vRealize Orchestrator クライアントに管理者としてログインします。

3 [アクティビティ] - [ワークフローの実行] の順に移動します。

4 ワークフローの実行を選択します。

5 左側のメニューからワークフローの実行項目を選択します。

[変数] と [ログ] のタブには、そのワークフロー アイテムに固有の情報が表示されるようになりました。

vRealize Orchestrator ワークフローの検証

vRealize Orchestrator には、ワークフローの検証ツールが用意されています。ワークフローを検証することで、ワークフローのエラーを特定したり、要素間のデータの流れが正しいかどうか確認したりすることができます。

vRealize Orchestrator ではデフォルトで、ワークフローの実行時に必ずワークフロー検証が実行されます。

ワークフローの検証を実行すると、検証ツールによってエラーや警告のリストが作成されます。このリストのエラーをクリックすると、エラーが含まれるワークフロー要素が強調表示されます。

ワークフロー エディタで検証ツールを実行すると、検出されたエラーを修正するためのクイック修正アクションが提示されます。クイック修正アクションによっては、追加情報や入力パラメータが必要になるものもあります。また、そのままでエラーを解決できるクイック修正アクションもあります。

ワークフローの検証では、要素間のデータのバインドと接続がチェックされます。ただし、ワークフローの各要素が実行するデータ処理はチェックされません。したがって、スキーマ要素の関数が間違っている場合は、有効なワークフローが不適切に実行され、誤った結果をもたらす可能性があります。

vRealize Orchestrator クライアントでのワークフローの検証と検証エラーの修正

ワークフローを実行するには、事前にワークフローを検証しておく必要があります。編集のためにワークフローを開いている場合は、検証エラーの修正のみ可能です。

前提条件

スキーマ要素がリンクされ、バインドが定義された、検証すべき完成したワークフローがあることを確認します。

手順

1 vRealize Orchestrator クライアントに管理者としてログインします。

2 [ライブラリ] - [ワークフロー] の順に移動し、検証するワークフローを選択します。

3 [編集] をクリックします。

4 上部メニューから [検証] をクリックします。

ワークフローが有効な場合は、確認メッセージが表示されます。ワークフローが無効な場合は、エラー リストが表示されます。

5 無効なワークフローの場合は、エラー メッセージをクリックし、問題を解決するための適切な手順を実行します。

検証ツールによって、エラーが発生しているスキーマ要素が強調表示されます(赤色のアイコンが追加されます)。クイック修正アクションが表示される場合もあります。

- 提示されたクイック修正アクションに同意する場合は、そのアクションをクリックして実行します。
- 提示されたクイック修正アクションに同意しない場合は、[ワークフローの検証] ダイアログ ボックスを閉じ、手動でスキーマ要素を修正します。

重要： vRealize Orchestrator によって提示された修正方法が適切かどうか必ず確認してください。

たとえば、未使用の属性を削除するように提示されることがありますが、実は属性が正しくバインドされていないことが原因の場合もあります。

6 検証エラーがすべてなくなるまで前述の手順を繰り返します。

結果

ワークフローが検証され、検証エラーが修正されました。

次のステップ

ワークフローを実行できます。

vRealize Orchestrator クライアントでのワークフロー スクリプトのデバッグ

ワークフロー アイテムのスクリプトにブレークポイントを挿入して、ワークフローの実行をデバッグできます。

ブレークポイントに達した場合は、いくつかの方法でデバッグ プロセスを継続することができます。ワークフロー スキーマから要素をデバッグする場合は、ワークフローの実行に関する一般的な情報の表示、ワークフローの変数の変更、ウォッチ式の追加、ログ メッセージの確認を行うことができます。

注： 非本番環境ですべてのスクリプトのデバッグを実行します。

手順

1 vRealize Orchestrator クライアントに管理者としてログインします。

2 ライブラリから、ワークフローを選択します。

3 ワークフロー スキーマを開き、ワークフロー要素を選択して、[スクリプト] タブをクリックします。

4 ブレークポイントを挿入するには、行番号の左側にある赤い円をクリックします。

注： ブレークポイントは、スクリプトを使用したワークフロー要素にのみ挿入可能です。

5 デバッグ モードでワークフローを実行するには、[デバッグ] をクリックします。

ワークフローで入力パラメータが必要な場合は、そのパラメータを指定します。

6 ブレークポイントに到達してワークフローの実行が停止したら、以下に示すいずれかのオプションを選択します。

オプション	説明
続行	別のブレークポイントに到達するか、ワークフローの実行が完了するまで、ワークフローの実行を再開します。
ステップイン	このオプションを使用して、ワークフロー要素にステップインできます。ワークフロー エディタでデバッグしている場合は、ネストされたワークフロー要素にステップインすることはできません。
ステップ オーバー	スキーマ内の現在の要素をスキップし、次の要素でワークフローの実行を停止します。

注： デバッガで、現在のブレークポイントをクリックして無視するように指示できます。これにより、ブレークポイントのシンボルが緑色の三角形に変更されます。

7 （オプション）[デバッガ] タブで、ウォッチ式を挿入します。

式を使用して、特定の変数の完了に従うことができます。

8 （オプション）[デバッガ] タブで、変数の値を変更します。

スキーマ要素別ワークフローのデバッグ

ワークフロー デザイナは、個別のスキーマ要素をデバッグできます。

手順

1 vRealize Orchestrator Client にログインします。

2 [ライブラリ] - [ワークフロー] の順に移動し、ワークフローを選択します。

3 [スキーマ] タブを選択します。

4 デバッグするワークフロー要素を選択し、要素の左上にあるデバッグ ボタンをクリックします。

注： ブレークポイントを [ワークフロー要素] スキーマ要素に追加することで、子ワークフローを親ワークフローから直接デバッグできます。デバッガが [ワークフロー要素] スキーマ要素に到達すると、子ワークフローのスキーマ ビューが開きます。

5 デバッグするその他のスキーマ要素についても、この手順を繰り返します。

6 [デバッグ] をクリックします。

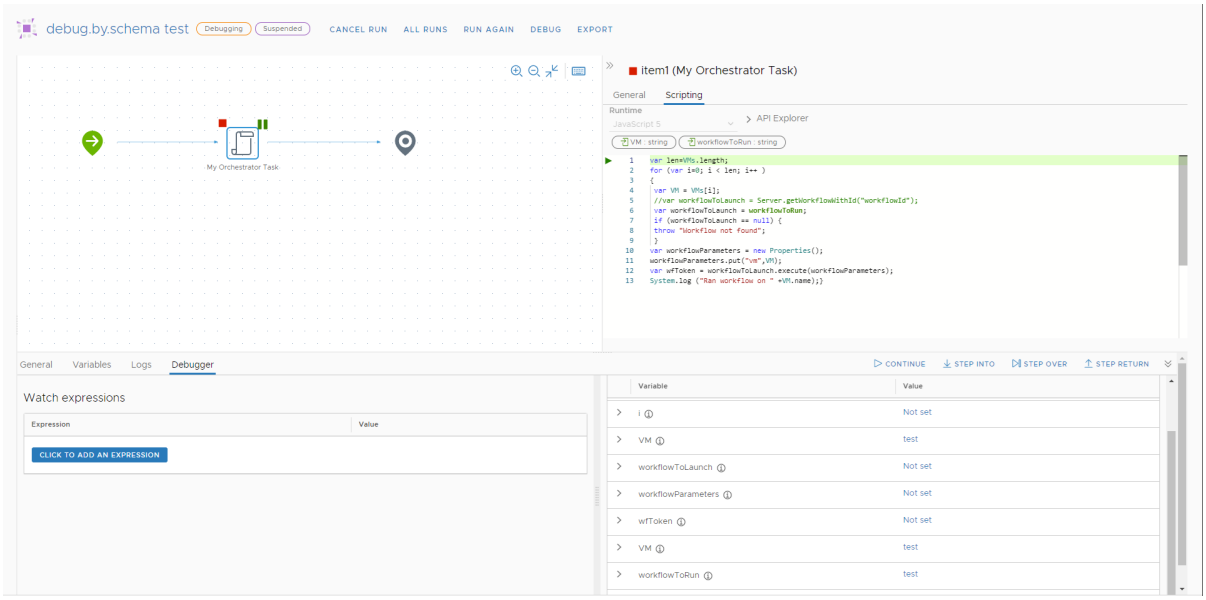
7 要求された入力パラメータ値を指定し、[実行] をクリックします。

ワークフローの実行が開始され、デバッガがブレークポイントが追加されたスキーマ要素に到達すると中断されます。

8 ブレークポイントに到達した場合、次のいずれかのオプションを選択します。

オプション	説明
続行	別のブレークポイントに到達するか、ワークフローの実行が完了するまで、ワークフローの実行を再開します。
ステップイン	現在のワークフロー関数にステップインします。デバッガが関数の現在の行の中を進行できない場合は、[ステップ オーバー] 処理が実行されます。
ステップ オーバー	デバッガは、現在の関数の次の行に進みます。
ステップ リターン	デバッガは、現在の関数が戻るときに実行される行に移動します。

9 （オプション）[変数] タブで、ワークフロー変数の値を編集します。



Python パッケージ用の Photon OS コンテナの構成

Python スクリプトをコンパイルするために使用されるオペレーティング システム (OS) によっては、関連する ZIP アーカイブを vRealize Orchestrator クライアントにインポートした後に、ワークフローまたはアクションが失敗することがあります。

vRealize Orchestrator の Python で使用されるランタイム コンテナの OS は Photon 3.0 をベースとしています。Linux などの、別の OS 用にコンパイルされた Python スクリプト パッケージは、ランタイム コンテナと互換性がありません。この問題により、vRealize Orchestrator のワークフローまたはアクションの一部として Python スクリプトの使用を試みた場合に、Python スクリプトが失敗することがあります。このようなシナリオでは、次のエラー メッセージがログに表示されます。

```
-04:00errorCannot find module action
```

この問題を解決するには、必要な Python パッケージを Photon OS のコンテナ フォルダにインストールする必要があります。

前提条件

Docker をインストールします。Docker の取得を参照してください。

手順

1 Python スクリプトの親フォルダに移動します。

2 コンテナ フォルダを親フォルダにマウントして、Photon のベース イメージでコンテナを作成します。

注： 次のスクリプトは、適切なコンテナを作成するために完全に実行する必要がある、単一の Docker コマンドです。

```
docker run -ti -v
$(pwd)/<name_of_folder_that_containes_your_python_script>/:/
<name_of_folder_that_containes_your_python_script>
photon:3.0
```

3 Python をコンテナにインストールします。

```
tdnf install -y python3-3.7.5-5.ph3 python3-pip-3.7.5-5.ph3
```

4 Python スクリプトを含むコンテナ フォルダに移動します。

5 Python スクリプトとパッケージを追加します。

注： Python スクリプトに必要なパッケージを `lib` フォルダにインストールします。

```
pip3 install <package_name> -t lib/
```

6 コンテナを終了して、コンテナにマウントしたローカル フォルダに移動します。

7 関連するすべてのファイルとフォルダを ZIP アーカイブに圧縮します。

8 ZIP アーカイブを vRealize Orchestrator クライアントにインポートし、アクションの一部として実行して、スクリプトを検証します。