Red Hat build of Cryostat 2

高度な Cryostat の設定

Last Updated: 2024-01-03

法律上の通知

概要

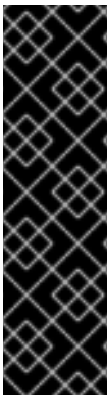
Red Hat build of Cryostat の高度な機能セットを設定して、要件に合わせて Cryostat をカスタマイズできるようにします。高度な Cryostat の設定 のドキュメントでは、API を使用して外部プラグインを Cryostat に登録し、Cryostat をデプロイメントスキーマとより適切に統合できるようにする方法について説明しています。

目次

はじめに

Red Hat build of Cryostat は、JDK Flight Recorder (JFR) のコンテナーネイティブ実装です。これを使用すると、OpenShift Container Platform クラスターで実行されるワークロードで Java 仮想マシン (JVM) のパフォーマンスを安全にモニターできます。Cryostat 2.4 を使用すると、Web コンソールまたは HTTP API を使用して、コンテナー化されたアプリケーション内の JVM の JFR データを起動、停止、取得、アーカイブ、インポート、およびエクスポートできます。

ユースケースに応じて、Cryostat が提供するビルトインツールを使用して、Red Hat OpenShift クラスターに直接レコーディングを保存して分析したり、外部のモニタリングアプリケーションにレコーディングをエクスポートして、レコーディングしたデータをより詳細に分析したりできます。



重要

Red Hat build of Cryostat は、テクノロジープレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は、実稼働環境でこれらを使用することを推奨していません。テクノロジープレビュー機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第1章 PLUGGABLE DISCOVERY API

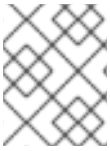
Pluggable Discovery API エンドポイント/api/v2.2/discovery を使用して、外部プラグインを **Cryostat** に登録し、検出可能なアプリケーションターゲットに関する情報を Cryostat に提供できます。

注記

Pluggable Discovery API の代わりに、Cryostat エージェントを Cryostat 検出プラグインとして使用できます。Cryostat エージェントは、JVM で実行されるアプリケーションのプラグインとして機能する Java インストルメンテーションエージェントとして実装されます。Cryostat エージェントは、検出プラグインとしてエージェントのロールが 2 つあるため、JMX ポートよりも柔軟にデプロイできる HTTP API を提供します。検出と Cryostat との接続の両方にエージェントの HTTP API を使用するようにターゲットアプリケーションを設定できます。Cryostat エージェントを使用するようにターゲットアプリケーションを設定する方法は、Java アプリケーションの設定 を参照してください。

1.1. PLUGGABLE DISCOVERY API の概要

Pluggable Discovery API エンドポイント/api/v2.2/discovery を使用して外部プラグインを **Cryostat** に登録し、検出可能なアプリケーションターゲットに関する情報を Cryostat に提供できます。プラグインは、Cryostat への登録に成功した後にそれ自体を登録解除するか、ターゲットアプリケーションに関する更新を Cryostat に継続的にプッシュすることができます。



注記

登録操作の目的は、Cryostat のセキュリティーを強化し、プラグインと Cryostat 間のデータの一貫性を維持することです。

Pluggable Discovery API は、Cryostat をデプロイメントスキーマに統合するための、OpenShift サービスアカウントメカニズムよりも柔軟な方法を提供します。

アプリケーション IP アドレスからポート番号への静的マップを作成するプラグインプログラムを作成する必要がある例を考えてみましょう。プラグインは Pluggable Discovery API を使用してこの情報を Cryostat に転送できるため、Cryostat はターゲットアプリケーションにより適切に接続できます。

Pluggable Discovery API は、次のハンドラーに依存して、/api/v2.2/discovery エンドポイントと **Cryostat** から送信された要求を管理します。

- **DiscoveryRegistrationHandler**
- **DiscoveryRegistrationCheckHandler**
- **DiscoveryGetHandler**
- **DiscoveryPostHandler**
- **DiscoveryDeregistrationHandler**

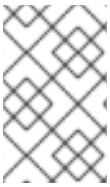
注記

これらのハンドラーの詳細については、後続のドキュメントセクションを参照してください。

API のエンドポイントを使用して Cryostat と対話する前に、プラグインのソースであるクライアントが次の前提条件を満たしていることを確認する必要があります。

- JSON 応答を受け入れます。
- HTTP リクエストを Cryostat に送信できます。
- **POST** 要求の Authorization ヘッダーに正しい Cryostat 認証情報を入力できます。
- Cryostat からの **GET** および **POST** リクエストを受信できます。
- **POST** リクエストを Cryostat に送信することにより、検出可能なターゲットに関する情報を JSON で公開します。

独自の検出プラグインプログラムを Cryostat に登録する必要がある場合は、Cryostat に組み込まれている検出メカニズムを無効にすることができます。組み込みの検出メカニズムを無効にすると、プラグインと Cryostat の両方が同じターゲットアプリケーション情報にアクセスできる場合に、Cryostat Web コンソールで開く可能性がある重複定義を減らすのに役立ちます。



注記

2 つの類似した定義が同じ JVM を指していることを Cryostat が検出すると、Cryostat はアーカイブされた記録を、各定義がアクセスする同じストレージの場所に保存します。

Red Hat OpenShift コマンドラインコンソール (CLI) で次の環境変数を設定して、Cryostat の組み込み検出メカニズムを無効にすることができます。

```
CRYOSTAT_DISABLE_BUILTIN_DISCOVERY=true
```

また、Cryostat 組み込みディスカバリー実装の設定を保持することを検討してから、次のアクションのいずれかを完了することもできます。

- サービスロケーターを実装にアタッチするプログラムを作成します。
- ターゲットアプリケーションを変更して、ターゲット情報を実装に直接送信します。

注記

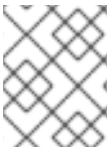
前述のアクションは、**高度な Cryostat の設定** ドキュメントの範囲外です。

1.2. DISCOVERYREGISTRATIONHANDLER

Pluggable Discovery API は、DisoveryRegistrationHandler を使用して、検出されたプラグインを **Cryostat** に登録します。ハンドラーは、プラグインと Cryostat からの **GET** および **POST** 要求を管理します。Cryostat にプラグインを登録しない場合、プラグインは Cryostat にターゲット情報を提供できません。

検出プラグインプログラムが登録目的で **POST** リクエストを Cryostat に送信すると、Cryostat は **callback** URL を読み取り、プラグインに **GET** リクエストを送信します。プラグインが **GET** リクエストに正しく応答すると、Cryostat は最初の **POST** リクエストに応答して登録リクエストを受け入れます。このプロセスにより、プラグインがアクティブになり、Cryostat で使用できるようになります。

失敗した要求は、プラグインが失敗したかオフラインになったことを示している可能性があります。この場合、エンドポイントは Red Hat OpenShift 上のデータベースからプラグインの情報を削除します。



注記

登録プロセスの後、Cryostat は通常の **POST** リクエストをプラグインに送信して、プラグインがまだ実行されていることを確認します。

GET または **POST** リクエストで **id** 要素と **token** 要素を指定することもできます。これらの要素はオプションですが、Cryostat に以前に登録されたプラグインの登録情報を再利用したい場合に使用することを検討できます。

Cryostat は、登録されたプラグインのトークンを作成します。このトークンには、有効期限と認証情報が含まれています。**POST** リクエストに有効な **ID** と **token** 情報が含まれている場合、Cryostat はプラグイン登録情報を再利用してトークンを更新できます。リクエストに **id** 要素または **token** 要素のみが含まれている場合は、プラグインを Cryostat に再登録する必要があります。

Cryostat が **POST** リクエストをプラグインの **callback** コンポーネントに送信した後、プラグインは **POST** リクエストを Cryostat に送信して、プラグインの登録の詳細を更新する場合があります。プラグインは、リクエストに **ID** と **token** 情報を含める必要があります。その後、Cryostat は **DiscoveryRegistrationHandler** を使用してプラグインの詳細を更新できます。Cryostat は更新されたトークンを含む応答をプラグインに送信し、プラグインは Cryostat への今後の要求でこのトークンを使用できます。

注記

Cryostat は、定期的にプラグインに **POST** リクエストを発行して、同じ **ID** と **token** 情報を使用して Cryostat に再登録することをプラグインに知らせることができます。プラグインがこのリクエストを無視すると、トークンの有効期限が切れる可能性があり、プラグインは Cryostat への完全な登録を完了する必要があります。

外部プラグインから送信される POST リクエストの例

```
{
    "realm": "my-plugin",
    "callback": "http://my-plugin.my-namespace.svc.local:1234/callback"
}
```

Cryostat がプラグインに送信する POST 応答の例

```
{
"data": {
    "result": {
        "id": "922dd4f4-9d7c-4ae2-8982-0903868226a6",
        "token": "<key_value>"
    }
},
"meta": {
    "status": "Created",
    "type": "application/json"
}
```

1.3. DISCOVERYREGISTRATIONCHECKHANDLER

Pluggable Discovery API は、**DisoveryRegistrationCheckHandler** ハンドラーを使用して、プラグインが Cryostat サーバーインスタンス上の登録ステータスを定期的にチェックできるようにします。

DisoveryRegistrationCheckHandler ハンドラーは、プラグインから Cryostat への **GET** リクエストを管理します。ハンドラーを使用することにより、外部プラグインは、登録先の Cryostat サーバーインスタンスが引き続きアクティブであることを定期的に確認し、プラグインの以前の登録を認識することができます。

Cryostat **callback** URL エンドポイントがプラグインインスタンスをチェックする方法 (Cryostat が**callback** URL を読み取り、プラグインに **GET** リクエストを送信する方法) と同様に、**DisoveryRegistrationCheckHandler** ハンドラーも同じ方法で動作しますが、リクエストを逆方向に送信します。つまり、プラグインは Cryostat サーバーに **GET** リクエストを送信し、Cryostat サーバー上の登録ステータスを確認します。リクエストが失敗した場合、たとえば、**Unexpected 401** または **Unexpected 404** エラー応答を受信した場合、プラグインは既存の登録情報を破棄し、再度登録を試みることができます。

外部プラグインから送信される GET リクエストの例

```
$ http -v https://my-cryostat.my-namespace.cluster.local:8181/api/v2.2/discovery/<plugin-registration-id>?token=<current-plugin-registration-token>
```

GET リクエストのチェックが成功し、Cryostat がプラグインの現在の登録を認識した場合の Cryostat の応答例

```
HTTP/1.1 200 OK
content-encoding: gzip
content-length: 86
content-type: application/json

{
 "data": {
    "result": null
 },
 "meta": {
    "mimeType": "JSON",
    "status": "OK"
 }
}
```

Cryostat がプラグインの登録詳細を認識しないために GET リクエストのチェックが失敗した場合の Cryostat の応答例

```
HTTP/1.1 404 Not Found
content-encoding: gzip
content-length: 95
content-type: application/json

{
 "data": {
    "reason": null
 },
 "meta": {
    "status": "Not Found",
    "type": "text/plain"
 }
}
```

1.4. DISCOVERYGETHANDLER

DiscoveryGetHandler はデプロイメントスキーマを照合し、これらのスキーマを階層ツリービューで開きます。これにより、Cryostat は、登録された検出可能なプラグインと対話して、特定のデプロイメントスキーマと統合できます。

少なくとも 1 つの Pod を含むデプロイメントを作成し、サービスが Red Hat OpenShift 上のデプロイメントまたは Pod にマップされると、Red Hat OpenShift は、**Pod IP** アドレスと **Service** ポートのすべての組み合わせに対して、エンドポイントオブジェクト **/api/v2.2/discovery** を作成します。**DiscoveryGetHandler** は、エンドポイント **GET** リクエストから情報を受け取り、デプロイスキーマ情報を JSON 形式で照合します。

次の例は、ハンドラーを開くと JSON 形式の階層ツリービューがどのように表示されるかを示しています。この例では、ツリーのルートは **UNIVERSE** ノードです。このノードには、Cryostat の組み込み検出メカニズムと Pluggable Discovery API によって検出されたプラグインに由来する子 **Realm** ノードタイプが含まれます。


```
{
  "data": {
    "result": {
      "children": [
        {
          "children": [],
          "labels": {},
          "name": "Custom Targets",
          "nodeType": "Realm"
        },
        {
          "children": [
            {
              "labels": {},
              "name": "service:jmx:rmi:///jndi/rmi://cryostat:9091/jmxrmi",
              "nodeType": "JVM",
              "target": {
                "alias": "io.cryostat.Cryostat",
                "annotations": {
                  "cryostat": {
                    "HOST": "cryostat",
                    "JAVA_MAIN": "io.cryostat.Cryostat",
                    "PORT": "9091",
                    "REALM": "JDP"
                  },
                  "platform": {}
                },
                "connectUrl": "service:jmx:rmi:///jndi/rmi://cryostat:9091/jmxrmi",
                "labels": {}
              }
            }
          ],
          "labels": {},
          "name": "JDP",
          "nodeType": "Realm"
        }
      ],
      "labels": {},
      "name": "Universe",
```

```
            "nodeType": "Universe"
        }
    },
    "meta": {
        "status": "OK",
        "type": "application/json"
    }
}
```

1.5. DISCOVERYPOSTHANDLER

Cryostat に登録されたプラグインは、**POST/api/v2.2/discovery/:id** リクエストを **DiscoveryPostHandler** に送信します。リクエストの **id** パラメーターは、登録されたプラグインの ID を参照します。このハンドラーは、プラグインに関連付けられたサブツリーを処理してから、デプロイメントスキーマをマップします。

Cryostat **POST** リクエストはサブツリー **REALM** ノードを定義するため、プラグインハンドラーはリクエストプロセス中に **REALM** ノードで子ノードタイプを公開できます。プラグインは、Cryostat **REALM** ノードに正しい情報を提供し、ノードタイプを正しいサブツリーの位置に配置します。プラグインは、**DiscoveryPostHandler** に送信する **POST** リクエストでトークンを提供する必要があります。これにより、プラグインは、以前に通過した認証ヘッダーチェックをバイパスできます。

注記

プラグインが Cryostat に更新を送信すると、Cryostat はプラグインが送信した以前の情報を置き換えます。Cryostat は、この情報をデータベースに保存します。プラグインは、リクエストごとに、検出可能なターゲットの完全なリストまたはツリーを Cryostat に送信する必要があります。

重要なターゲットアプリケーション情報の詳細を示すプラグインの POST リクエストの例

```
[
    {
        "labels": {},
        "nodeType": "JVM",
        "name": "service:jmx:rmi:///jndi/rmi://myapp.svc.local:9091/jmxrmi",
        "nodeType": "JVM",
        "target": {
            "alias": "com.MyApp",
            "annotations": {
                "cryostat": {},
                "platform": {}
        },
            "connectUrl": "service:jmx:rmi:///jndi/rmi://myapp.svc.local:9091/jmxrmi",
            "labels": {}
        }
    }
]
```

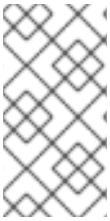
プラグインの POST リクエストに対する Cryostat の応答の例

```
{
    "data": {
```

```
        "result": null
    },
    "meta": {
        "mimeType": "JSON",
        "status": "OK"
    }
}
```

1.6. DISCOVERYDEREGISTRATIONHANDLER

プラグインが Cryostat に **plug-in** として登録され、その後プラグインをシャットダウンする必要がある場合、プラグインは通常、Cryostat から **plug-in** として自身を登録解除するリクエストを発行します。Cryostat は **DELETE /api/v2.2/discovery/:id?token=:token** リクエストを **DiscoveryDeregistrationHandler** に送信し、Cryostat からプラグインを登録解除します。



注記

Cryostat は、プラグインが引き続き実行されていることを確認するために、登録プロセスの後に定期的な **POST** リクエストをプラグインに送信します。プラグインがこれらのリクエストのいずれにも応答しない場合、Cryostat はプラグインの登録を解除するプロセスを開始します。

ハンドラーは、検出スキーマからプラグインの **REALM** サブツリーを削除し、Cryostat からプラグインの登録情報を削除します。

ハンドラーが処理する DELETE /api/v2.2/discovery/:id?token=:token リクエストの例

```
{
    "data": {
        "result": "bcc0f3a6-dc48-402e-a3d6-9fbb63beff78"
    },
    "meta": {
        "mimeType": "JSON",
        "status": "OK"
    }
}
```

1.7. エラーハンドラーコード

Pluggable Discovery API は、ハンドラーのいずれかがタスクを完了した際、または Cryostat へのプラグインの登録時に問題が発生した際に、メッセージを返します。

ハンドラーは、**GET** および **POST** リクエストに基づいて Cryostat にプラグインを登録しようとすると、次のメッセージタイプのいずれかを返すことができます。

- **200**: ハンドラーがタスクを正常に完了しました。たとえば、**DiscoveryDeregistrationHandler** は、メッセージの **id** 要素で定義された未登録のプラグインを含むメッセージを JSON 形式で返します。
- **400**: JSON ドキュメント構造が無効であるか、**id** 要素が無効な形式で記述されています。

- **401**: プラグインは登録のための Authorization ヘッダーステップを通過できませんでした。トークンの有効期限が切れた場合、またはプラグインの登録を解除した場合も、ハンドラーはこのエラーメッセージを返します。

- **404**: プラグイン **id** 要素が見つかりませんでした。プラグインが **callback** チェックに失敗した可能性があります。プラグインの再登録を検討してください。

第2章 GRAPHQL API

GraphQL API エンドポイントである **/api/v2.2/graphql** は、ターゲット JVM に対してより短くて単純なクエリーを自動的に実行します。これらのクエリーは、ターゲット JVM のアクティブおよびアーカイブされたレコーディングに対して実行することができます。さらに、API は一般的な Cryostat アーカイブに対してクエリーを実行できます。クエリーをカスタマイズして、アクティブまたはアーカイブされたレコーディングの以下のタスクを自動化できます。

クエリーをカスタマイズして、アクティブまたはアーカイブされたレコーディングの以下のタスクを自動化できます。

- アーカイブ
- 削除
- 起動
- 停止

カスタムクエリーを作成する場合、クエリーに GraphQL API エンドポイントの具体的な情報を指定する必要があります。続いて、**POST** リクエストで情報を処理し、その情報を Cryostat に送信することができます。次の例では、Graph QL API の情報を指定します。

```
POST /api/beta/graphql HTTP/1.1
Accept: application/json, /;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 171
Content-Type: application/json
Host: localhost:8181
User-Agent: HTTPie/3.1.0
```

GraphQL API は、ワークロード機能が制限されている HTTP REST API よりも強力です。たとえば、HTTP REST API では、OpenShift 上のコンテナー内にあるスケーリングされた各レプリカで開始するレコーディングのコピーごとに、API リクエストを作成する必要があります。GraphQL API は、1 つの API 要求でこのタスクを実現できます。これにより、API のパフォーマンスが向上し、Cryostat インスタンスのネットワークトラフィックを削減できます。

HTTP REST API のもうひとつの制限されたワークロード機能として、この API は、応答データに対して反復的なアクションを実行する際に API JSON 応答を解析するカスタムクライアントを書く必要があるなど、ユーザーの介入をより必要とすることが挙げられます。GraphQL API では、この操作を完了する必要はありません。

関連情報

GraphQL の概要 (GraphQL) を参照してください。

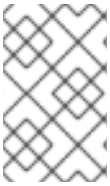
2.1. GRAPHQL API を使用したカスタムクエリーの作成

HTTPie などの HTTP クライアントを使用して GraphQL API と対話し、カスタムクエリーを生成することができます。

API を使ってクエリーを作成する場合、データ型やフィールドに特定の値を指定し、関数がこれらの値を使用して必要なデータを正確に検索できるようにする必要があります。

1 つのクエリーで複数の操作を実行するワークフローを自動化できるユースケースを考えてみましょう。たとえば、**HTTPie** クライアントからのリクエストで Cryostat にクエリーを送り、Cryostat が以下のタスクを実行できるようにすることができます。

1. ターゲットとなるすべての JVM アプリケーションのスナップショット記録を取ることができます。
2. 各アプリケーションの記録情報を Cryostat アーカイブにコピーします。
3. 検出されたターゲット JVM アプリケーションの連続監視記録を自動的に開始する自動化ルールを作成します。



注記

クエリーにはさまざまな可能性があるので、クエリーのデータ型やフィールドの値を正確に決定してください。そうしないと、要件に合わないクエリー結果を取得する可能性があります。

以下の例は、HTTP クライアントを使用して GraphQL API と対話し、Cryostat データに対して単純なクエリーと複雑なクエリーを生成する方法を示しています。

Cryostat インスタンスと対話するすべての既知のターゲット JVM を決定するための単純なクエリーの例

```
$ https :8181/api/v1/targets
HTTP/1.1 200 OK
content-encoding: gzip
content-length: 223
content-type: application/json

 {
 targetNodes {
     name
     nodeType
     labels
     target {
         alias
         serviceUri
     }
 }
}
```

前の例では、**targetNodes** 要素に **name** などの特定の値を設定しています。クエリーを実行すると、クエリーは、指定された基準に一致するターゲット JVM のリストを返します。

特定の Pod に属する Cryostat インスタンスに表示されるすべてのターゲットアプリケーションを特定するための複雑なクエリーの例

```
$ https -v :8181/api/v2.2/graphql query=@graphql/target-nodes-query.graphql
POST /api/v2.2/graphql HTTP/1.1
Accept: application/json, /;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 171
Content-Type: application/json
```

```
Host: localhost:8181
User-Agent: HTTPie/2.6.0

 {
environmentNodes(filter: { name: "<application_pod_name>" }) {
    descendantTargets {
      doStartRecording(recording: {
        name: "myrecording",
        template: "Profiling",
        templateType: "TARGET",
         duration: 30
        }) {
        name
        state
      }
    }
 }
}
```

前の例では、**environmentNodes** 要素に以下の特定の値を設定しており、この例では JVM ターゲットアプリケーションは返されません。

- **<application_pod_name>**: クエリーが、Cryostat と同じ namespace で動作する特定の Pod をターゲットにしていることを確認します。
- **descendantTargets**: JVM ターゲットオブジェクトの配列を提供します。
- **doStartRecording**: GraphQL API は、クエリーがリストする各ターゲット JVM に対して JFR 記録を開始します。

クエリーを実行すると、クエリーは、アクティブなアプリケーションノードのリストなど、開始した JFR 記録に関する情報を返します。

Red Hat OpenShift の場合、このクエリーは **Deployment** と **DeploymentConfig** の API オブジェクト、および Cryostat と対話する Pod を返します。

この複雑なクエリーは、GraphQL API が、Cryostat および Red Hat OpenShift と対話する任意の JVM オブジェクトを返す単一の API リクエストを実行する方法を示しています。続いて、API リクエストは、それらの返されたオブジェクトの JFR 記録を開始します。

第3章 JMC AGENT プラグイン

JMC Agent プラグインを使用して、JMC Agent 実装を Cryostat に追加できます。次に、JMC Agent を使用して、カスタム JFR イベントを実行中のターゲット JVM アプリケーションに追加できます。この操作では、JVM アプリケーションを再起動する必要はありません。

関連情報

- Red Hat build of Cryostat での JDK Flight Recorder の使用 を参照してください。

3.1. JMC AGENT プラグインを使用したカスタムイベントの追加

Cryostat を JMC Agent と組み合わせて使用すると、実行中の JVM アプリケーションで問題を診断する必要がある場合に詳細情報を提供できます。

JMC Agent JAR ファイルは、ターゲット JVM アプリケーションと同じ Red Hat OpenShift コンテナーに存在する必要があります。そうしないと、Cryostat はアプリケーションで JMC Agent 機能を使用することはできません。

Cryostat Web コンソールから、プローブテンプレートをアップロードしてから、これらのテンプレートを JVM アプリケーションに挿入することができます。これらのテンプレートプローブは、必要に応じて後で削除できます。プローブテンプレートは、Cryostat が処理できる一連のオブジェクトを記述し、Cryostat が JVM アプリケーションで一連の JMC Agent 操作を完了できるようにします。

JMC Agent でターゲット JVM アプリケーションを起動すると、Cryostat は、アプリケーションが JMC Agent で実行されているかどうかを自動的に検出します。

重要

RHEL の場合、JMC パッケージは CodeReady Linux Builder (CRB) (**Builder** とも呼ばれる) リポジトリーによって提供されます。RHEL に JMC をインストールできるように、RHEL で CRB リポジトリーを有効にする必要があります。CRB パッケージは製品化された RHEL パッケージとして Source Red Hat Package Manager (SRPM) でビルドされるため、CRB パッケージは定期的に更新を受け取ります。JDK Mission Control (JMC) のダウンロードおよびインストール (Red Hat build of Cryostat での JDK Flight Recorder の使用) を参照してください。

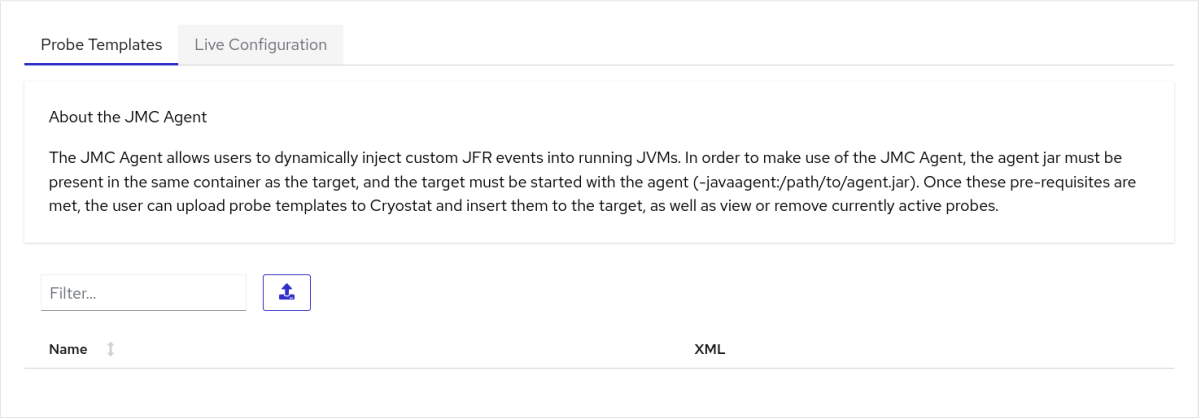
前提条件

- **jmc** パッケージをダウンロードしてインストールしている。
- Adoptium Agent JAR ファイルをダウンロードしている。adoptium/jmc-build (GitHub) を参照してください。
- **--add-opens=java.base/jdk.internal.misc=ALL-UNNAMED** フラグを使用して、Java アプリケーションを起動している。**./<your_application> --add-opens=java.base/jdk.internal.misc=ALL-UNNAMED** はその例です。
- Java アプリケーションの JMC Agent を起動している。JDK Mission Control (JMC) エージェントの開始 (Red Hat build of Cryostat での JDK Flight Recorder の使用) を参照してください。

手順

1. Cryostat Web コンソールから、**Events** メニューに移動します。JMC Agent が Cryostat インスタンスに正常に追加された場合は、**Event Templates** ペインの下に **Probe Templates** ペインが開きます。

図3.1 Cryostat Web コンソールのプローブテンプレートタブ



注記

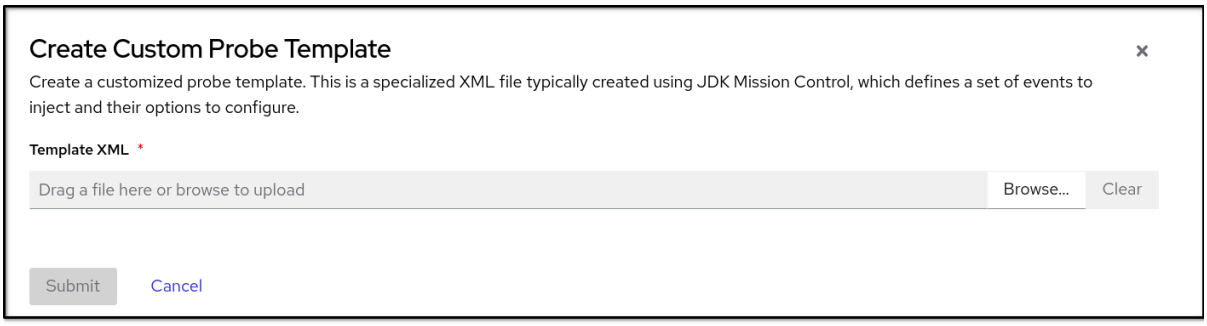
Web コンソールで **Authentication Required** ダイアログボックスが開く場合があります。プロンプトが表示されたら、**Authentication Required** ダイアログボックスに **Username** と **Password** を入力し、**Save** をクリックして、JMX 認証情報をターゲット JVM に提供します。

2. 希望するテキストエディターを使用して XML 設定ファイルを作成します。Cryostat がアプリケーションで実行するイベントなどの JFR イベント情報でファイルを設定します。
以下の例は、JFR イベント情報を含むカスタムプローブテンプレート XML ファイルを示しています。このファイルを Cryostat にアップロードすると、Cryostat は Cryostat Agent Plugin Demo Event と呼ばれるカスタム JFR イベントをアプリケーションに追加できます。Cryostat は、JMC Agent の **retrieveEventProbes** メソッドが呼び出されると JFR イベントを開始します。

```
<jfragent>
   <!-- Global configuration options -->
   <config>
    <classprefix>__JFREvent</classprefix>
    <allowtostring>true</allowtostring>\
    <allowconverter>true</allowconverter>
   </config>
    <events>
     <event id="cryostat.demo.jfr.event9">
        <label>Cryostat Agent Plugin Demo Event</label>
        <description>Event for the agent plugin demo</description>
        <path>io/cryostat/demo/events</path>
        <stacktrace>true</stacktrace>
        <class>io.cryostat.core.agent.AgentJMXHelper</class>
        <method>
            <name>retrieveEventProbes</name>
      <descriptor>()Ljava/lang/String;</descriptor>
        </method>
       <location>WRAP</location>
    </event>
 </events>
</jfragent>
```

3. **Upload** ボタンをクリックして、カスタムイベントテンプレートを Cryostat に追加します。Cryostat Web コンソールで **Create Custom Probe Template**が開きます。

図3.2 Cryostat Web コンソールの Create Custom Probe Template ウィンドウ



ヒント

アップロードしたファイルをこの **Template XML** フィールドから削除する場合は、**Clear** ボタンをクリックします。

4. **Browse** ボタンをクリックして XML ファイルを見つけます。
5. ファイルをアップロードしたら、**Submit** をクリックします。カスタムプローブテンプレートファイルが **Probe Templates** テーブルで開きます。
6. プローブテンプレートの横にあるオーバーフローメニューをクリックします。
7. **Insert Probes** をクリックします。プローブは **Probe Templates** タブの下に表示され、**Live Configuration** タブの下のテーブルに表示されます。
8. **オプション: Live Configuration** タブに移動し、アクティブなプローブごとに、**Name**、**Class** などの情報を表示できます。
9. **オプション： Live Configuration** タブから、**Remove All Probes** をクリックして、表に記載されているプローブを削除できます。

検証

1. **Events** メニューから、**Event Types** タブをクリックします。
2. XML 設定の名前付き JFR イベントがテーブルに一覧表示されていることを確認します。この手順で使用されている例では、**Cryostat Agent Plugin Demo Event** が表に表示されます。

関連情報

- Red Hat build of Cryostat での JDK Flight Recorder の使用 を参照してください。

改訂日時: 2024-01-02