Red Hat build of Cryostat 2

Configuring advanced Cryostat configurations

Last Updated: 2023-12-12

Legal Notice

Abstract

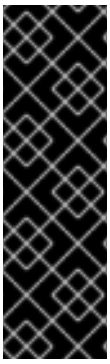
Configure the Red Hat build of Cryostat set of advanced features, so that you can customize Cryostat to suit your requirements. The Configuring advanced Cryostat configurations document describes how to use the API to register external plug-ins with Cryostat, so that you can better integrate Cryostat with your deployment schema.

Table of Contents

PREFACE

The Red Hat build of Cryostat is a container-native implementation of JDK Flight Recorder (JFR) that you can use to securely monitor the Java Virtual Machine (JVM) performance in workloads that run on an OpenShift Container Platform cluster. You can use Cryostat 2.4 to start, stop, retrieve, archive, import, and export JFR data for JVMs inside your containerized applications by using a web console or an HTTP API.

Depending on your use case, you can store and analyze your recordings directly on your Red Hat OpenShift cluster by using the built-in tools that Cryostat provides or you can export recordings to an external monitoring application to perform a more in-depth analysis of your recorded data.



IMPORTANT

Red Hat build of Cryostat is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. PLUGGABLE DISCOVERY API

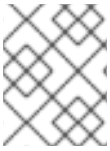
You can use the Pluggable Discovery API endpoint, **/api/v2.2/discovery**, to register an external plug-in with Cryostat and provide information about discoverable application targets to Cryostat.

NOTE

As an alternative to the Pluggable Discovery API, you can use the Cryostat agent as a Cryostat discovery plug-in. The Cryostat agent is implemented as a Java Instrumentation Agent that acts as a plug-in for applications that run on the JVM. The Cryostat agent provides an HTTP API that offers greater deployment flexibility than a JMX port due to the agent's dual role as a discovery plug-in. You can configure your target applications to use the agent's HTTP API for both detection and connectivity with Cryostat. For more information about configuring your target applications to use the Cryostat agent, see Configuring Java applications.

1.1. PLUGGABLE DISCOVERY API OVERVIEW

You can use the Pluggable Discovery API endpoint, **/api/v2.2/discovery**, to register an external plug-in with Cryostat and then provide information about discoverable application targets to Cryostat. A plug-in can either unregister itself after successfully registering with Cryostat or continually push updates to Cryostat about a target application.



NOTE

The purpose of the registration operation is to enhance Cryostat security and to maintain data consistency between the plug-in and Cryostat.

The Pluggable Discovery API provides a more flexible way to integrate Cryostat into your deployment schema that the Red Hat OpenShift service account mechanism.

Consider an example where you need to write a plug-in program that creates a static map of application IP addresses to port numbers. The plug-in can use the Pluggable Discovery API to transport this information to Cryostat, so that Cryostat can better connect with the target application.

The Pluggable Discovery API depends on the following handlers to manage requests sent from the **/api/v2.2/discovery** endpoint and Cryostat:

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

NOTE

For more information about these handlers, see the subsequent document sections.

Before you can use the API's endpoint to interact with Cryostat, you must ensure that the client, which is the plug-in's source, meets the following prerequisites:

- Accepts JSON responses.
- Can send HTTP requests to Cryostat.
- Can enter the correct Cryostat credentials in a **POST** request's Authorization header.
- Can receive **GET** and **POST** requests from Cryostat.
- Publishes information about discoverable targets in JSON by sending **POST** requests to Cryostat.

If you need to register your own discovery plug-in program with Cryostat, you can disable the Cryostat built-in discovery mechanism. Disabling the built-in discovery mechanism helps to reduce duplicate definitions that might open in the Cryostat web console if both the plug-in and Cryostat can access the same target application information.

NOTE

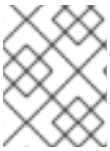
If Cryostat detects that two similar definitions point to the same JVM, Cryostat stores any archived recordings in the same storage location that each definition accesses.

You can set the following environment variable in your Red Hat OpenShift command-line console (CLI) to disable the Cryostat built-in discovery mechanism:

```
CRYOSTAT_DISABLE_BUILTIN_DISCOVERY=true
```

You can also consider preserving the setting for the Cryostat built-in discovery implementation and then complete one of the following actions:

- Create a program that attaches a service locator to the implementation.
- Modify a target application to send target information directly to the implementation.



NOTE

The previously listed actions are outside the scope of the *Configuring advanced Cryostat configurations* document.

1.2. DISCOVERYREGISTRATIONHANDLER

The Pluggable Discovery API uses **DisoveryRegistrationHandler** to register a discovered plug-in with Cryostat. The handler manages **GET** and **POST** requests from the plug-in and Cryostat. If you do not register a plug-in with Cryostat, the plug-in cannot provide target information to Cryostat.

When your discovery plug-in program sends a **POST** request to Cryostat for registration purposes, Cryostat reads the **callback** URL and sends a **GET** request to the plug-in. If the plug-in correctly responds to the **GET** request then Cryostat accepts the registration request by responding to the initial **POST** request. This process ensures that the plug-in is active and available to Cryostat.

A failed request might indicate that the plug-in failed or went offline. In this case, the endpoint removes the plug-in's information from the database on Red Hat OpenShift.

NOTE

After the registration process, Cryostat sends regular **POST** requests to the plug-in to ensure that the plug-in is still running.

You can also specify **id** and **token** elements in your **GET** or **POST** request. These elements are optional, but you can consider using them in situations where you want to reuse registration information for a plug-in that was previously registered with Cryostat.

Cryostat creates a token for the registered plug-in, and this token contains expiry and authorization information. If a **POST** request contains valid **id** and **token** information, Cryostat can reuse the plug-in registration information and refreshes the token. If a request contains only an **id** element or a **token** element, you must re-register the plug-in with Cryostat.

After Cryostat sends a **POST** request to the plug-in's **callback** component, the plug-in might send a **POST** request to Cryostat to refresh the plug-in's registration details. The plug-in must include its **id** and **token** information in its request. Cryostat can then use **DiscoveryRegistrationHandler** to refresh the plug-in's details. Cryostat sends a response that includes the updated token to the plug-in, and the plug-in can use this token for future requests to Cryostat.

NOTE

Cryostat can issue a **POST** request to the plug-in at regular intervals to remind the plug-in about re-registering with Cryostat by using the same **id** and **token** information. If the plug-in ignores this request, the token might expire and the plug-in must complete a full registration with Cryostat.

Example of a POST request sent from an external plug-in

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

Example of a POST response that Cryostat sends to a plug-in

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

1.3. DISCOVERYREGISTRATIONCHECKHANDLER

The **Pluggable Discovery** API uses the **DisoveryRegistrationCheckHandler** handler to enable plug-ins to periodically check their own registration status on a Cryostat server instance.

The **DisoveryRegistrationCheckHandler** handler manages **GET** requests from the plug-in to Cryostat. By using the handler, external plug-ins can periodically verify the Cryostat server instance to which they are registered is still active and recognizes the prior registration of the plug-in.

Similar to how the Cryostat **callback** URL endpoint checks plug-in instances, where Cryostat reads the **callback** URL and sends a **GET** request to the plug-in, the **DisoveryRegistrationCheckHandler** handler works in the same way, but sends the request in the opposite direction. That is, the plug-in sends a **GET** request to the Cryostat server to check its registration status on the Cryostat server. If the request fails, for example, if an **Unexpected 401** or **Unexpected 404** error response is received, the plug-in can discard its existing registration information and attempt to register again.

Example of a GET request sent from an external plug-in

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

Example of a Cryostat response when the GET request check succeeds and Cryostat recognizes the current registration of the plug-in

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

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

Example of a Cryostat response when the GET request check fails because Cryostat does not recognize the plug-in registration details

```
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

The **DiscoveryGetHandler** collates deployment schemas and opens these schemas in a hierarchical tree view, so that Cryostat can interact with any registered discoverable plug-ins to integrate with a specific deployment schema.

When you create a deployment with at least one pod, and a service maps to the deployment or pod on Red Hat OpenShift, Red Hat OpenShift creates an endpoint object, **/api/v2.2/discovery**, for all combinations of the **Pod IP** address and the **Service** port. The **DiscoveryGetHandler** receives information from an endpoint **GET** request and then collates deployment schema information in JSON format.

The following example demonstrates how the handler opens results in a hierarchical tree view in JSON format. In the example, the root of the tree is the **UNIVERSE** node. This node contains child **Realm** node types, which stem from Cryostat's built-in discovery mechanism and the plug-ins discovered by the Pluggable Discovery API.

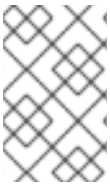

```
{
    "data": {
        "result": {
            "children": [
                {
                    "children": [],
                    "labels": {},
                    "name": "Custom Targets",
                    "nodeType": "Realm"
                },
                {
                    "children": [
                        {
                            "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

A plug-in registered with Cryostat sends a **POST /api/v2.2/discovery/:id** request to **DiscoveryPostHandler**. The **id** parameter of the request refers to the ID of the registered plug-in. This handler processes any subtrees associated with the plug-in and then maps a deployment schema.

The Cryostat **POST** request defines a subtree **REALM** node, so that the plug-in handler can publish its child node types in the **REALM** node during the request process. The plug-in takes responsibility for providing the correct information for theCryostat **REALM** node and for placing node types in the correct subtree positions. The plug-in must provide a token with the **POST** request that it sends to the **DiscoveryPostHandler**, so that the plug-in can bypass the authorization header check that the plug-in previously passed.



NOTE

After a plug-in sends updates to Cryostat, Cryostat replaces the previous information that a plug-in has sent. Cryostat stores this information in a database. A plug-in must send a complete list or tree of discoverable targets to Cryostat on each request.

Example of a plug-in's POST request that details important target application information

```
[
    {
        "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": {}
        }
    }
]
```

Example of Cryostat response to the plug-in's POST request

```
{
    "data": {
        "result": null
    },
    "meta": {
```

```
        "mimeType": "JSON",
        "status": "OK"
    }
}
```

1.6. DISCOVERYDEREGISTRATIONHANDLER

If a plug-in registered with Cryostat as a **plug-in**, and then the plug-in needs to shutdown, the plug-in typically issues a request to un-register itself as a **plug-in** from Cryostat. Cryostat sends a **DELETE /api/v2.2/discovery/:id?token=:token** request to **DiscoveryDeregistrationHandler** that then unregisters the plug-in from Cryostat.

NOTE

Cryostat sends regular **POST** requests to the plug-in after the registration process to ensure that the plug-in is still running. If the plug-in does not respond to one of these requests, Cryostat starts the process of unregistering the plug-in.

The handler removes both the plug-in's **REALM** subtree from the discovery schema and the plug-in's registration information from Cryostat.

Example of DELETE /api/v2.2/discovery/:id?token=:token request that the handler processes

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

1.7. ERROR HANDLER CODES

The Pluggable Discovery API returns messages when any of its handlers either completes a task or experiences issues when registering a plug-in with Cryostat.

A handler can return any of the following message types when the handler attempts to register a plug-in with Cryostat based on **GET** and **POST** requests:

- **200**: The handler successfully completes the task. For example, **DiscoveryDeregistrationHandler** returns a message in JSON format with the unregistered plug-in defined in the **id** element of the message.
- **400**: JSON document structure is invalid or the **id** element was written in an invalid format.
- **401**: Plug-in did not pass the Authorization header step for registration. If the token expired or you unregistered the plug-in, the handler also returns this error message.
- **404**: Plug-in **id** element was not found. Plug-in might have failed a **callback** check. Consider re-registering the plug-in.

CHAPTER 2. GRAPHQL API

The GraphQL API endpoint, **/api/v2.2/graphql**, automatically runs shorter and simpler queries against target JVMs. These queries can run against a target JVM's active and archived recordings. Additionally, the API can run queries against general Cryostat archives. You can customize the queries to automate the following tasks for active or archived recordings:

You can customize the queries to automate the following tasks for active or archived recordings:

- Archive
- Delete
- Start
- Stop

When creating a custom query, you must specify specific information for the GraphQL API endpoint in your query. A **POST** request can then handle the information and send the information to Cryostat. The following example specifies information for the Graph QL API:

```
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
```

The GraphQL API is more powerful than the HTTP REST API, which has limited workload capabilities. For example, the HTTP REST API would require you to create an API request for each copy of a recording that you want to start on each scaled replica that is inside a container on OpenShift. The GraphQL API can achieve this task in one API request, which improves the API's performance and reduces any network traffic for your Cryostat instance.

Another limited workload capability for the HTTP REST API is that this API requires more user intervention, such as requiring you to write custom clients to parse API JSON responses when performing iterative actions on response data. The GraphQL API does not require you to complete this operation.

Additional resources

See Introduction to GraphQL (GraphQL)

2.1. CUSTOM QUERY CREATION WITH THE GRAPHQL API

You can use an HTTP client, such as **HTTPie**, to interact with the GraphQL API for generating custom queries.

When using the API to create a query, you must specify specific values for data types and fields, so that a function can use these values to accurately locate the data you require.

Consider a use case where you could automate a workflow that performs multiple operations in a single query. As an example, an **HTTPie** client request could send a query to Cryostat so that Cryostat could perform the following tasks:

1. Take snapshot recordings of all your target JVM applications.
2. Copy recording information from each application into the Cryostat archive.
3. Create an automated rule that automatically starts a continuous monitoring recording for any detected target JVM applications.

NOTE

Many querying possibilities exist, so ensure you accurately determine values for the data types and fields of a query. Otherwise, you might get query results that do not match your requirements.

The following examples demonstrate using the HTTP client to interact with the GraphQL API for generating a simple query and a complex query on Cryostat data.

Example of a simple query for determining all known target JVMs that interact with a Cryostat instance

```
$ 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
     }
 }
}
```

The previous example sets specific values for the **targetNodes** element, such as **name**. After you run the query, the query returns a list of any target JVMs that match the criteria specified.

Example of a complex query for determining all target applications visible to a Cryostat instance that belong to a particular pod

```
$ 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
        }
     }
 }
}
```

The previous example sets the following specific values for the **environmentNodes** element, and the example does not return JVM target applications:

- **<application_pod_name>**: ensures the query targets a specific pod that operates in the same namespace as Cryostat.
- The **descendantTargets**: Provides an array of JVM target objects.
- **doStartRecording**: The GraphQL API starts JFR recording for each target JVM that the query lists.

After you run the query, the query returns information about the JFR recording that you started, such as a list of active application nodes.

For Red Hat OpenShift, this query would return **Deployment** and **DeploymentConfig** API objects, and pods that interact with Cryostat.

The complex query demonstrates how the GraphQL API can perform a single API request that returns any JVM objects that interact with Cryostat and Red Hat OpenShift. The API request then starts a JFR recording on those returned objects.

CHAPTER 3. JMC AGENT PLUGIN

You can use the JMC Agent Plugin to add the JMC Agent implementation to Cryostat. You can then use the JMC Agent to add custom JFR events into a running target JVM application. This operation does not require you to restart your JVM application.

Additional resources

- See Using JDK Flight Recorder with Red Hat build of Cryostat

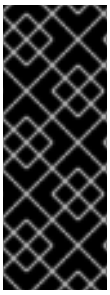
3.1. ADDING CUSTOM EVENTS BY USING THE JMC AGENT PLUGIN

Cryostat in combination with the JMC Agent can provide you with more information when you need to diagnose issues with your running JVM application.

The JMC Agent JAR file must be in the same Red Hat OpenShift container as the target JVM application. Otherwise, Cryostat cannot use the JMC Agent functionality on the application.

From the Cryostat web console, you can upload probe templates and then insert these templates into the JVM application. You can remove these template probes at a later stage, if required. A probe template describes a set of objects that Cryostat can process, so that Cryostat can complete a sequence of JMC Agent operations on the JVM application.

When you start a target JVM application with the JMC Agent, Cryostat automatically detects if the application is running with the JMC Agent.



IMPORTANT

For RHEL, the JMC package is provided by the CodeReady Linux Builder (CRB), also known as *Builder*, repository. You must enable the CRB repository on RHEL, so that you can install JMC on RHEL. CRB packages are built with the Source Red Hat Package Manager (SRPM) as productized RHEL packages, so CRB packages regularly receive updates. See, Downloading and installing JDK Mission Control (JMC) (Using JDK Flight Recorder with Red Hat build of Cryostat)

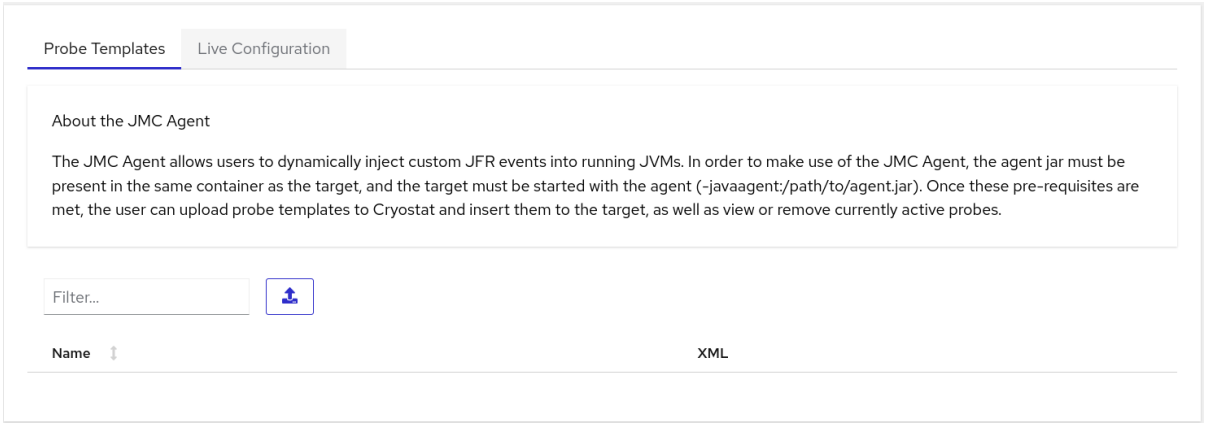
Prerequisites

- Downloaded and installed the **jmc** package.
- Downloaded the Adoptium Agent JAR file. See adoptium/jmc-build (GitHub).
- Started your Java application with the **--add-opens=java.base/jdk.internal.misc=ALL-UNNAMED** flag. For example, **./<your_application> --add-opens=java.base/jdk.internal.misc=ALL-UNNAMED**.
- Started the JMC Agent for your Java application. See Starting a JDK Mission Control (JMC) Agent (Using JDK Flight Recorder with Red Hat build of Cryostat).

Procedure

1. From your Cryostat web console, go to the **Events** menu. If the JMC Agent was successfully added to your Cryostat instance then a **Probe Templates** pane opens under the **Event Templates** pane.

Figure 3.1. The Probe Templates tab on the Cryostat web console



NOTE

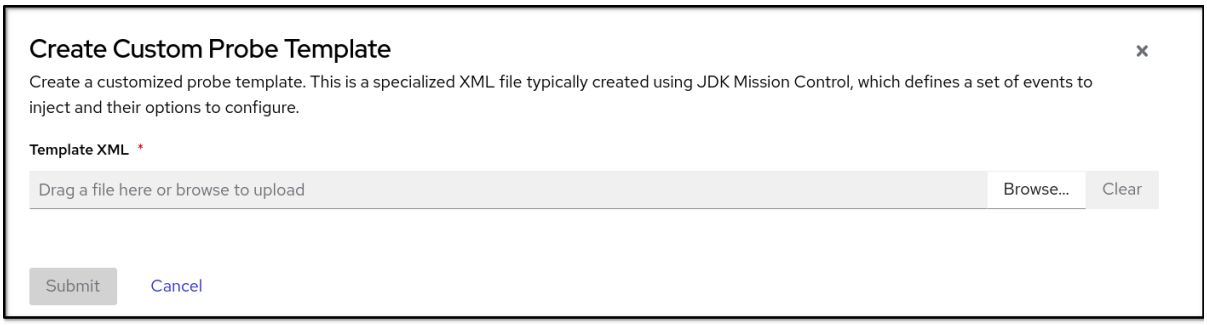
An **Authentication Required** dialog box might open on your web console. If prompted, enter your **Username** and **Password** in the **Authentication Required** dialog box, and click **Save** to provide your JMX credentials to the target JVM.

2. Use your preferred text editor to create an XML configuration file. Populate the file with JFR event information, such as what events Cryostat must perform on the application.
   The following example shows a custom probe template XML file that contains JFR event information. When this file is uploaded to Cryostat, Cryostat can add a custom JFR event, called Cryostat Agent Plugin Demo Event, to an application. Cryostat starts the JFR event when the **retrieveEventProbes** method of the JMC Agent is called.

```
<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. Click the **Upload** button to add a custom event template to Cryostat. A **Create Custom Probe Template** opens on your Cryostat web console.

Figure 3.2. The Create Custom Probe Template window on the Cryostat web console



TIP

Click the **Clear** button if you want to remove the uploaded file from this **Template XML** field.

4. Click the **Browse** button to locate your XML file.
5. After you upload the file, click **Submit**. Your custom probe template file opens in the **Probe Templates** table.
6. Click the overflow menu that is next to your probe template.
7. Click **Insert Probes**. The probes display in the table under the **Probe Templates** tab and the table under the **Live Configuration** tab.
8. *Optional:* Go to the **Live Configuration** tab, where you can view information, such as **Name**, **Class**, and so on, for each active probe.
9. *Optional:* From the **Live Configuration** tab, you can click **Remove All Probes** to delete probes that are listed in the table.

Verification

1. From the **Events** menu, click the **Event Types** tab.
2. Check that the named JFR event from your XML configuration is listed in the table. For the example used in this procedure, **Cryostat Agent Plugin Demo Event** displays in the table.

Additional resources

- See Using JDK Flight Recorder with Red Hat build of Cryostat

Revised on 2023-12-12 18:07:31 UTC