vRealize Code Stream Plug-In SDK Development Guide

vRealize Code Stream 2.2

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see http://www.vmware.com/support/pubs.

EN-002420-00

vmware®

You can find the most up-to-date technical documentation on the VMware Web site at:

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

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

Contents

About the vRealize Code Stream Plug-In SDK Development Guide

The VMware *vRealize Code Stream Plug-in SDK Development Guide* describes how to use the plug-in Software Development Kit (SDK) to develop your own plug-ins for use with vRealize Code Stream.

To help you integrate your own customized tasks into vRealize Code Stream, this information describes how to develop your own plug-ins. It describes the extensibility of vRealize Code Stream for continuous development and release automation. It shows you how to clone and install the plug-in SDK, and describes the concepts and procedures to develop your own plug-ins. It describes the tools and examples provided with the plug-in SDK, and provides instructions to develop your own plug-ins for use with vRealize Code Stream.

Intended Audience

This information is intended for developers who write code to integrate customized tasks into vRealize Code Stream.

VMware Glossary

VMware provides a glossary of terms that might be unfamiliar to you. For definitions of terms as they are used in VMware technical documentation, go to http://www.vmware.com/support/pubs.

Introduction to vRealize Code Stream and the Plug-In SDK

1

vRealize Code Stream is a Dev Ops release automation solution that ensures continuous development and integration of your applications. vRealize Code Stream runs the tasks in each stage, and ensures that each task passes or that failed tasks are skipped, before the pipeline moves to the next stage of development.

When you set up the pipeline for your continuous development workflow, you add stages and tasks to the pipeline. Your pipeline might include several stages. Each stage of a pipeline runs a specific set of tasks. For example, one way to use stages and tasks is as follows:

- In the Development stage, the pipeline might obtain binaries from the latest build, install those binaries on a server, and collect the results of unit tests. If all tests pass, the pipeline continues to the Integration stage.
- In the Integration stage, vRealize Code Stream runs tasks to test the interactions with external services.
- Upon success in the Development and Integration stages, the pipeline advances to the Production stage where vRealize Code Stream finalizes the build services and makes the application bits available for delivery.

You can extend the capability of vRealize Code Stream by creating your own plug-ins using the plug-in SDK to run tasks in your vRealize Code Stream pipelines. For example, you can build your own plug-in to make a REST call to a remote server.

This chapter includes the following topics:

Overview of the Plug-In SDK

The plug-in SDK for vRealize Code Stream provides a set of tools and examples that you use to build your own Java-based plug-ins to integrate customized tasks into your vRealize Code Stream pipelines. Each plug-in that you develop implements a single task.

To use the plug-in SDK, you clone the SDK onto your local machine, and build the SDK so that you can deploy it to your development environment. See Chapter 2, "Preparing to Use the Plug-In SDK," on page 15.

Understanding Terminology

vRealize Code Stream uses its own orchestration engine to run pipelines that are comprised of stages and tasks.

The following concepts apply to vRealize Code Stream.

Table 1-1. vRealize Code Stream Concepts

Component	Description
Pipeline	A collection of tasks, grouped by stages.
Stage	A logical separation that divides your pipeline into units that represent the various stages of your continuous delivery process such as Development, Staging, and Production.
Task	A single item of work done during a stage such as deployment of virtual machines or running unit tests. Tasks can run sequentially or in parallel. An individual vRealize Code Stream plug-in implements each task.
Endpoint	A specialized data type that encapsulates all the information for your tiles to access a resource external to the vRealize Code Stream appliance. For example, the endpoint data type for a REST service might contain the URL and credential information for the service.

The following concepts apply to developing your own plug-ins for vRealize Code Stream.

Table 1-2. Concepts for Plug-In Developers

Component	Description
Plug-in	The basic extensibility unit for vRealize Code Stream. Each plug-in is packaged into a bundle that contains several tiles, data types, and views.
Bundle	The packaging for a plug-in, in a specific folder structure that is compressed into a ZIP file as part of the Maven build process, and which can be deployed to your vRealize Code Stream appliance.
Tile	A Java class that represents an action that your bundle can perform. For example, a Task tile contains the code that runs during a vRealize Code Stream pipeline execution. The tile types include the Task tile, Endpoint tile, and View Callable tile.
Data Type	A YAML representation of a collection of data that can be an input to a tile, or an output from a tile. For example, the Boolean data type represents a true or false value. You can nest data types to make complex structures.
Properties	Key-value pairs that represent the inputs to and outputs of a task.
Tile View	A user interface that your tiles display in vRealize Code Stream. For example, you use the view named `config` to configure the inputs that a Task tile requires when you design a pipeline.

Using the Plug-In SDK to Extend vRealize Code Stream

To extend vRealize Code Stream with your own plug-ins, you create tiles that run during the pipeline executions. Each tile is a self-contained unit of work that you create to perform an action. Tiles provide views for endpoint configuration, task configuration, and task results.

Each plug-in runs a single task. For example:

- A task that performs provisioning in vRealize Automation is a separate vRealize Code Stream plug-in.
- A task to run jobs in Jenkins is a separate vRealize Code Stream plug-in.

You can have multiple instances of a task anywhere in a pipeline. For example, you can have two tasks that perform provisioning in vRealize Automation, and both tasks run the same plug-in. Or, you can have the same type of task run in two stages, such as a task for provisioning in vRealize Automation in the development and production stages, and both tasks run the same plug-in.

Packaging and Deploying Plug-ins

You use Maven build commands to compile each plug-in into a package called a bundle that includes your data types, views, and tiles. You deploy the bundle to your vRealize Code Stream appliance.

For example, you might create a tile to perform the action of making a REST call to a remote server to start a procedure. For more information about tiles, see "Example Tiles," on page 24.

You create data types to represent the input and output properties for your tiles. The data type can be a primitive, complex, or nested type. For more information about data types, see "Data Types and Example Data Type," on page 23.

You can host views that appear in vRealize Code Stream when users create or modify an instance of the task with their own input, or when they view the results of the task after it ran. For more information about views, see "Tile Views," on page 26.

For more information about these terms, see "Understanding Terminology," on page 8.

For more information about deploying bundles, see "Deploy the Sample Plug-In," on page 19.

Maven Commands

The plug-in SDK includes a Maven extension that modifies the default Maven lifecycle so that your plug-ins are packaged as bundles and deployed to your vRealize Code Stream appliance.

The following table describes the modified Maven goals and their parameters.

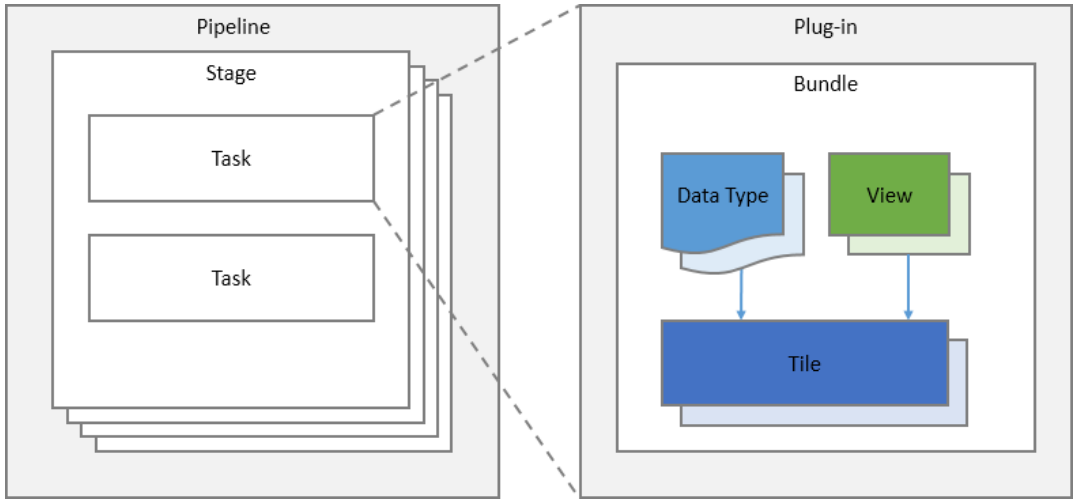
Table 1-3. Maven Goals

Goal	Function	Use	Optional Parameters
package	Builds the bundle.	Run as part of `mvn clean install`. Can be run separately as `mvn package` to only generate the bundle.	`–Dbundle.package.excludes` Specify files to exclude from your bundle using a comma separated list of GLOB patterns.
deploy	Deploys the bundle to vRealize Code Stream.	`mvn clean deploy –Dbundle.deploy.host=VRCS_APPLIANCE`	`–Dbundle.deploy.privateKeyPath` Specify the path to your private key file if you use one to connect to your vRealize Code Stream appliance. Defaults to `$HOME/.ssh/id_rsa`.

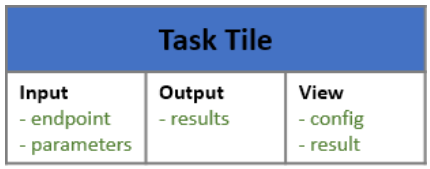
Organizing Work in a Pipeline

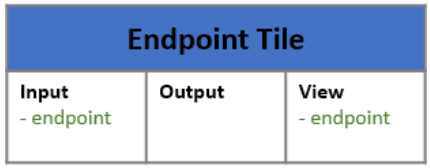
In your plug-in, you create self-contained units of work called tiles. Each tile runs a task in a pipeline in vRealize Code Stream to perform the actions required to run the task.

Each tile, except for view callable tiles, includes data types and views that you create when you develop the plug-in.



Each bundle includes a Task tile, which is required. It may include an Endpoint tile and one or more View Callable tiles, which are optional. The plug-in runs the tiles when it communicates with external systems.





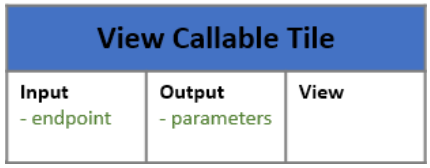
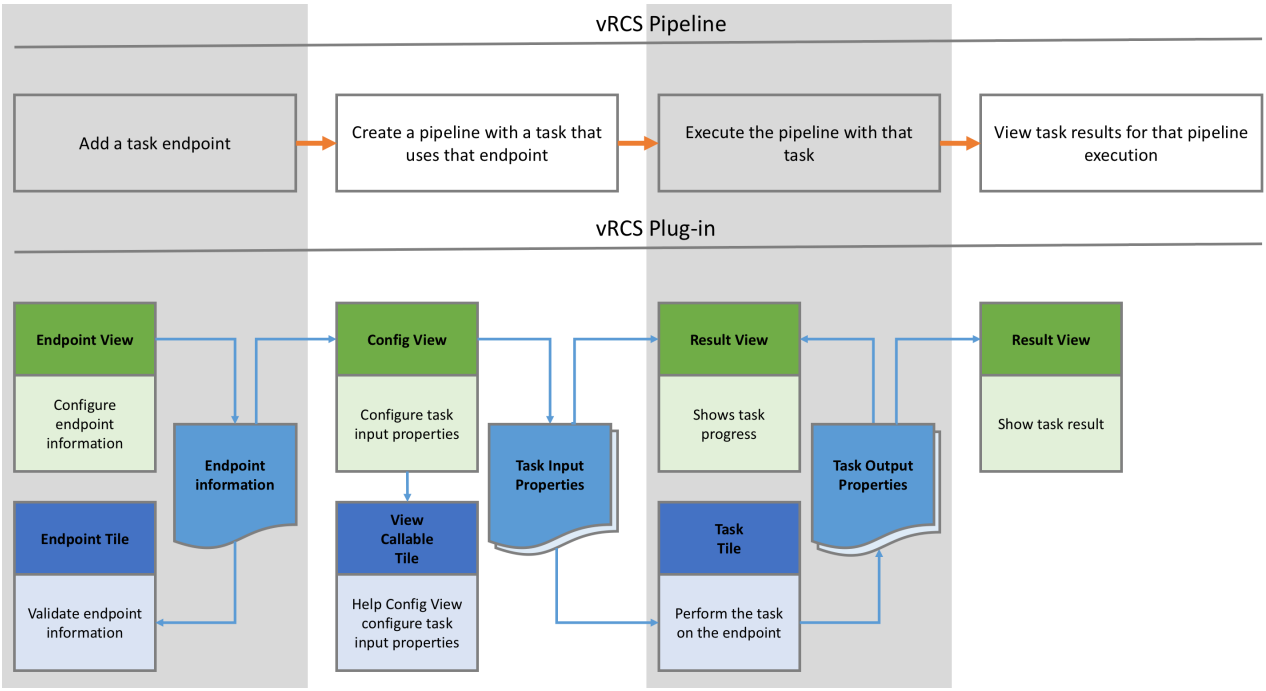


Table 1-4. Tiles for vRealize Code Stream Plug-Ins

Tile Type	Function	Tag	Typical Inputs	Typical Outputs	Views
Task Tile	Represents the work done by a task when the pipeline runs.	VRCS_TASK	Endpoint data type Other task inputs	Results of the task execution	Configuration Result
Endpoint Tile	Validates an endpoint for a task.	VRCS _ENDPOINT	Endpoint data type	None. The tile succeeds when the endpoint is valid.	Endpoint
View Callable Tile	Called by views of other tiles to gather and format user input.	VRCS_VIEW _CALLABLE	Endpoint data type	Information about the task inputs supported by the endpoint.	None

These components interact with pipelines in vRealize Code Stream in the following manner.



The top part of the diagram illustrates the task lifecycle in a vRealize Code Stream pipeline. The bottom area illustrates the plug-in lifecycle, and indicates how it aligns with the lifecycle of the task.

How vRealize Code Stream Uses Tiles

When you create a plug-in, you model the work to be done as independent tiles that vRealize Code Stream runs in a pipeline.

For example, with your own REST plug-in you might have vRealize Code Stream make REST calls to a remote server to start a procedure.

After you create your REST plug-in, you perform the following steps in your vRealize Code Stream appliance to support the action of making REST calls to the REST endpoint.

Table 1-5. How Views and Tiles Map to Pipeline Lifecycle Stages

Pipeline Lifecycle Stage	Components	What Occurs
Add an endpoint for a task	■ Endpoint tile ■ Endpoint view	You create an endpoint, and in the Add Endpoint page select your REST plug-in. The Endpoint view displays the properties that comprise the Endpoint data type that you define in your REST plug-in. These properties might include the URL, user name, and password. When you save your endpoint: ■ The Endpoint view passes the endpoint data type to the endpoint tile. ■ The Endpoint tile validates the information users enter, such as the URL, user name, and password, against the REST endpoint. The Endpoint tile attempts to pass the user name and password credentials to the REST endpoint to ensure that the credentials are valid and connect to the REST endpoint. It then sends a request to the URL, and gets a response. ■ When the information is valid, the endpoint tile succeeds and stores the information. ■ If the information is not valid, an error message occurs and indicates that the endpoint tile cannot read from or write to the URL.
Configure the task	■ Config view ■ View callable tile	You create a pipeline that uses your REST endpoint. Then, you add a stage, add a task, and configure the task. When you add a task, you select the REST task as the provider. When you configure the task, the Config view provides the input and output parameters as defined in the Task tile of your REST plug-in. ■ For example, the Endpoint data type is an input parameter for the Config view and can be displayed as a drop down list of REST endpoints. ■ Other input parameters for the REST request might include the REST method, relative path, headers, expected status codes, and the expected response body. In the task Config view, users can preview the response, which is provided by a view callable tile. The preview is based on the input parameters that the user provides. ■ The Config view calls the view callable tile to confirm whether the information that the user entered is valid. ■ View callable tiles can also work with the Config view to provide more information, such the types of headers the server supports. After you configure your task, the Config view saves all the information for the Task tile to run the action.
Execute the pipeline	Task tile	When you run the pipeline, the Task tile runs the action based on the input properties, and generates the output properties to display in the Result view. You can design the Task tile in your plug-in to run repeatedly until the work it has started on the endpoint is completed, or has failed to complete due to an error.
View the pipeline execution results	Result view	The Result view indicates that the task is in progress, and reports the results of the pipeline execution. When the task finishes, the Result view indicates success or failure.

For more information about sample tiles, see "Example Tiles," on page 24.

Components of the Plug-In SDK

You must install the plug-in SDK, and develop plug-ins on your local development machine.

The plug-in SDK provides the following components that you use to develop your own plug-ins.

Table 1-6. Components of the Plug-In SDK

Components	Purpose
Sample plug-in code examples	Build and test sample plug-ins. The sample plug-in includes samples tiles and sample views.
Maven tools	Help developers throughout the development lifecycle to: ■ Generate plug-ins with archetype. ■ Test with the test framework. ■ Deploy plug-ins with the `mvn deploy` command.
Compile time dependencies	The Java classes needed to compile tiles and write and run unit tests for them.

Preparing to Use the Plug-In SDK 2

Before you use the vRealize Code Stream plug-in SDK, you must be familiar with the prerequisites and the directory structure of the plug-in SDK. Then you can clone and install the plug-in SDK, and deploy the sample plug-in to your vRealize Code Stream appliance.

When you are familiar with the plug-in SDK, you can begin to create your own custom plug-ins to run tasks in your pipelines.

This chapter includes the following topics:

Prerequisites

Before you create your own plug-ins for use with vRealize Code Stream, ensure that your environment meets the prerequisites.

To build your own plug-ins, the following procedures assume that you have an independent vRealize Code Stream appliance for development, before you deploy the plug-in to production. The procedures also assume that you are familiar with Maven tooling, and that you have considered the security aspects of your environment.

To build your own plug-ins, you have:

- Downloaded and installed Java Development Kit 8 on your development machine.
- Installed Apache Maven.

  If you already installed the JDK on your development machine, you have the build tools and samples required to create your own plug-ins.
- Verified that you can log in to GitHub and that you can access `https://github.com/vmwaresamples/vrcs-sdk-samples`.

When you install the vRealize Code Stream Plug-In SDK, it installs all its dependencies into your local Maven cache.

After you perform the prerequisites, review the sample plug-in source code that is included with your vRealize Code Stream appliance.

Prepare Your Development Appliance

By default, the server checks and reloads the plug-ins every 10 minutes. To reduce the interval for the server to reload the plug-ins, you update the vRealize Code Stream properties file. This procedure is optional.

The following procedure shows you how to update the vRealize Code Stream properties file to set the interval to reload plug-ins.

Prerequisites

- Install and build the plug-in SDK, and build the sample plug-in bundles. See "Install and Build the Plug-In SDK," on page 18.

Procedure

1 Use SSH to log in to your vRealize Code Stream appliance with the following command:

```
ssh root@VRCS_APPLIANCE
```

2 Edit the properties file with the following command:

```
vi /usr/lib/vcac/server/webapps/release-management-service/WEB-
INF/classes/configuration/application.properties
```

3 To have the server reload your plug-ins more frequently, change the interval in the vRealize Code Stream properties file to the value you desire. For example, to change the interval to 30 seconds, enter:

```
fms.server.plugins.interval=30
```

4 Restart vRealize Code Stream for the changes to take effect with the following command:

```
service vcac-server restart
```

The restart can take about 15 minutes.

You have changed the interval in the vRealize Code Stream properties file so that the server reloads your plug-ins more frequently.

What to do next

Clone the GitHub repository of the plug-in SDK. See "Obtain the Plug-In SDK," on page 16.

Obtain the Plug-In SDK

To use the plug-in SDK, you must clone the repository and install the plug-in SDK. Then you can start to use the plug-in SDK, test the sample plug-in, and deploy it to your existing vRealize Code Stream appliance.

The plug-in SDK includes subdirectories named `lib` and `samples`. It also includes a `pom.xml` for Maven.

Prerequisites

- Performed the prerequisites. See "Prerequisites," on page 15.
- Verify that you can access the location in GitHub to clone the plug-in SDK. See `https://github.com/vmwaresamples/vrcs-sdk-samples`.

Procedure

1 To clone the plug-in repository to your development machine, run the following command:

```
git clone https://github.com/vmwaresamples/vrcs-sdk-samples.git vrcs-plugin-sdk-2.2.0
```

This step creates a folder for the plug-in SDK.

2 Change to the folder with the command:

```
cd vrcs-plugin-sdk-2.2.0
```

The directories named `lib` and `samples` appear, with the file named `pom.xml`.

3 Review the directory structure and files.

You have cloned and installed the vRealize Code Stream plug-in SDK.

What to do next

Review the directory structure of the plug-in SDK. See "Directory Structure of the Plug-In Bundle," on page 17. Then install and build the plug-in SDK. See "Install and Build the Plug-In SDK," on page 18.

Directory Structure of the Plug-In Bundle

After you clone the files from GitHub, the required files are extracted to your local machine.

The folders and files resemble the following structure:

```
vrcs-plugin-sdk-2.2.0
├── samples
│     └── rest-sample-plugin
├── lib
│     ├── tile-api.jar
│     ├── tile-api-test.jar
│     ├── tile-api-test-dependency.jar
│     ├── vrcs-simple-plugin-archetype.jar
│     ├── vrcs-bundle-maven-plugin.jar
│     └── vrcs-bundle-maven-plugin.pom
└── pom.xml
```

The files in the SDK folder perform specific functions.

Table 2-1. Files in the SDK Folder

Directories and Files	Function
`rest-sample-plugin`	An SDK sample that can make REST calls against a remote server.
`tile-api.jar`	Interfaces that your Java-based tiles must implement. All the files in the `lib` folder are installed in your Maven cache.
`tile-api-test.jar`	A unit test framework that allows tiles to be run independently for test-driven development.
`tile-api-test-dependency.jar`	A dependency JAR file that contains all the classes required to independently run the tiles.
`vrcs-simple-plugin-archetype.jar`	A Maven archetype of a simple vRealize Code Stream plug-in.
`vrcs-bundle-maven-plugin.jar/.pom`	A Maven plug-in that allows seamless plug-in development using Apache Maven.

The directory named `lib` is where you run the `mvn clean install` command the first time when you build the plug-in SDK.

The directory structure for any plug-in includes the following folders and files:

```
my-plugin
├── src/main
│     ├── java
│           └── (Java package structure)
```

```
│                   ├── MyTaskTile.java
│                   ├── MyEndpointTile.java
│                   └── MyViewCallableTile.java
│       └── bundle
│             ├── datatypes
│                   └── MyEndpoint.yaml
│             ├── ui
│                   ├── config.js
│                   ├── result.js
│                   ├── endpoint.js
│                   └── misc
│             └── bundle.yaml
└── pom.xml
```

The files in the plug-in SDK perform specific functions.

Table 2-2. Files in the Plug-In SDK

Directories and Files	Description
`src/main/java`	Includes the Java package structure and the tile files. ■ `MyTaskTile.java` provides execution logic that is called when a pipeline runs. ■ `MyEndpointTile.java` validates all endpoint data that your plug-in needs. ■ `MyViewCallableTile.java` helps gather inputs for the Task tile.
`src/main/bundle`	Includes the data types, view files, and the bundle YAML file. ■ `datatypes/MyEndpoint.yaml` is a YAML representation of any endpoint data that your plug-in must run. ■ `ui/config.js` provides the custom logic and user interface to configure the inputs for your Task tile when you create a pipeline. ■ `ui/result.js` provides the custom logic and user interface to display the results of your Task tile after a pipeline runs. ■ `ui/endpoint.js` provides the custom logic and user interface to configure any endpoint data that your plug-in might require. ■ `misc` is optional. You can add any other files your views might need in folders here, and they can be loaded by calling `VRCS.view.alpaca.importResource('misc/fileName')` in the appropriate `.js` file.
`bundle.yaml`	Main file that specifies the layout of your bundle.
`pom.xml`	File that you modify with your plug-in folder name as the artifact ID. You add any Java dependencies required to it.

Install and Build the Plug-In SDK

You use Maven to build vRealize Code Stream plug-ins. You must install and build the plug-in SDK so that you can use the samples to create your own plug-ins.

In this procedure, you run the install commands twice so that the second install builds the plug-in SDK sample. You can then deploy the sample to your vRealize Code Stream appliance.

Prerequisites

■ Clone the plug-in SDK. See "Obtain the Plug-In SDK," on page 16.

Procedure

1 From the command line, change directory to the SDK folder with the following command:

```
cd vrcs-plugin-sdk-2.2.0
```

2 Enter the following commands to install the SDK libraries into your local Maven cache.

```
cd lib
```

```
mvn clean install
```

3 Enter the following commands to change to the root directory and build the SDK and the sample plug-in.

```
cd ..
```

```
mvn clean install
```

This step builds the SDK sample to deploy to your vRealize Code Stream appliance.

You have installed the plug-in SDK and built the sample plug-in.

What to do next

Deploy the sample plug-in to your vRealize Code Stream appliance.

Deploy the Sample Plug-In

You can deploy a sample plug-in to your vRealize Code Stream appliance so that you can test it in your own environment.

Each folder in the directory named `samples` represents a vRealize Code Stream plug-in, which includes a bundle that is packaged during the build process. You can deploy a bundle to your vRealize Code Stream appliance, and load it by running the commands in the plug-in directory, as shown in the following procedure.

Perform this procedure in your development and test environments, but not in production environments.

Prerequisites

- Clone the plug-in SDK. See "Obtain the Plug-In SDK," on page 16.
- Understand how to use the debug option. See "Testing and Debugging Your Tiles and Views," on page 30.

Procedure

1 On the machine where you cloned the plug-in SDK, at a command prompt open the folder named `samples`.

2 Open the directory named `rest-sample-plugin`.

3 Deploy the plug-in to your vRealize Code Stream appliance.

a Enter, but do not run the following command:

```
mvn clean deploy -Dbundle.deploy.host=VRCS_APPLIANCE
```

b In this command, replace **`VRCS_APPLIANCE`** with the hostname or IP address of your own vRealize Code Stream appliance.

When the results indicate that the build is successful, all sample plug-ins are deployed to your appliance.

4 Log in to your appliance and verify that all files are deployed to the appliance.

   a Use SSH to log in to your vRealize Code Stream appliance with the following command:

   ```
   ssh root@VRCS_APPLIANCE
   ```

   b Open the directory named `/var/lib/codestream/plugins`.

   c Verify that the sample plug-in is deployed. For example, if you deploy the REST sample, the folder named `rest-sample-plugin` appears in this directory.

   Your new plug-in is loaded and running in your vRealize Code Stream appliance.

5 To ensure that the cache is updated, and that your newly deployed plug-in loaded properly, log in to your vRealize Code Stream appliance and create your pipeline.

   a Log in to the appliance where you deployed your plug-in, and include the debug option.

   For example, log in to your appliance using the form:
   `mycodestream-20.yourcompany.com/vcac/org?debug=true`

   b In your vRealize Code Stream appliance, click the **Code Stream** tab.

   c Click the **Endpoints** tab, click **Add**, and create a REST endpoint.

   d Click the **Pipelines** tab, and click **Add**.

   e Enter a name for the new pipeline, and click **Stages**.

   f Click **Add Stage**, and in the new stage, click **Add Task**.

   g From the list of tasks, click your REST task, and provide a proper task name.

   h Configure your REST task, run the task, and view the results.

You have deployed the sample plug-in to your vRealize Code Stream appliance.

What to do next

Create your own plug-in. See "Developing Your Own Plug-In By Modifying the SDK Sample," on page 29.

Developing Your Own Plug-In 3

To create your own plug-in to integrate tasks with vRealize Code Stream, you can either copy the samples provided with the vRealize Code Stream plug-in SDK and modify them as you need. Or, you can use the archetype provided with the plug-in SDK to create your own plug-in from the Maven archetype.

This chapter includes the following topics:

Before You Create Your Own Plug-In

You must be familiar with the directory structure provided with the plug-in SDK. You must also understand how to create bundles, tiles, and views.

To perform the actions required before you create your own plug-in, follow the steps in Chapter 2, "Preparing to Use the Plug-In SDK," on page 15.

Become Familiar With the Directory Structure

The sample plug-in provided with the plug-in SDK show you how to create your own plug-ins.

Before you create your own plug-ins, familiar yourself with the directory structure and files provided by the plug-in SDK. See "Directory Structure of the Plug-In Bundle," on page 17.

Bundles and Example Bundle.yaml

You use Maven to compile each plug-in into a bundle that is used to package your files for deployment. Each bundle includes data types, views, and tiles.

You must create your file named `bundle.yaml`. This file connects all the components into a single plug-in for vRealize Code Stream, and establishes the relationships between the components.

The following code is for the sample `bundle.yaml` file that corresponds to the sample plug-in.

```
---
  bundleId: "vrcs.my-plugin" #NOTE: Plug-in namespace, tiles, dataTypes.
  version: "1.0.0" #NOTE: Bundle version. Revise as needed.
  modelVersion: "1.5.0" #NOTE: Document model version for bundle.yaml.
  apiVersion: "2.2.0" #NOTE: Corresponds to the Code Stream version your bundle targets.
  description: "{CHANGE_THIS_DESCRIPTION}"
  documentation: "{CHANGE_THIS_LINK}"
```

```
author: "{UPDATE_THIS_AUTHOR}"
datatypes:
  - "datatypes/MyEndpoint.yaml"
tiles:
  -
    tileId: "vrcs.my-plugin:MyTile" #See note 1.
    displayName: "My vRCS plugin"
    inputProperties:
      myEndpoint:
        type: "vrcs.my-plugin:MyEndpoint" #See note 2.
      myInputs:
        type: "String[]" #See note 3.
    outputProperties:
      myResult:
        type: "Integer"
    executorType: "JAVA"
    executor: "com.vmware.vrcs.tile.{YOUR_PACKAGE_NAME}.MyTile.java"
    tags:
      - VRCS_TASK #See note 4.
    views:
      -
        path: "/config.js"
        viewName: "config"
        viewType: "alpaca"
      -
        path: "/result.js"
        viewName: "result"
        viewType: "alpaca"
    categories: ["ALL"] # Required only for execution tiles and is always ["ALL"]
  -
    tileId: "vrcs.my-plugin:MyEndpointTile" #See note 5.
    displayName: "My vRCS plugin endpoint"
    inputProperties:
      myEndpoint:
        type: "vrcs.my-plugin:MyEndpoint"
    executorType: "JAVA"
    executor: "com.vmware.vrcs.tile.{YOUR_PACKAGE_NAME}.MyEndpointTile.java"
    tags:
      - VRCS_ENDPOINT #See note 6.
    views:
      -
        path: "/endpoint.js"
        viewName: "endpoint"
        viewType: "alpaca"
  -
    tileId: "vrcs.my-plugin:MyViewCallableTile" #See note 7.
    inputProperties:
      myEndpoint:
        type: "vrcs.my-plugin:MyEndpoint"
    outputProperties:
      myNumInputs:
        type: "Integer"
    executorType: "JAVA"
    executor: "com.vmware.vrcs.tile.{YOUR_PACKAGE_NAME}.MyViewCallableTile.java"
```

Table 3-1. Details for Sample Bundle.yaml Code

Reference	Code	Description
Note 1	`tileId: "vrcs.my-plugin:MyTile"`	This tile runs when a user runs a pipeline that contains the task named `My VRCS plug-in`.
Note 2	`type: "vrcs.my-plugin:MyEndpoint"`	Endpoint data type.
Note 3	`myInputs: type: "String[]"`	Represents an array of strings.
Note 4	`tags: - VRCS_TASK`	Indicates that this tile can be run as a task in a vRealize Code Stream pipeline. It requires the configuration and result views to be specified views.
Note 5	`tileId: "vrcs.my-plugin:MyEndpointTile"`	This tile validates the endpoint data, and ensures that the endpoint is reachable.
Note 6	`tags: - VRCS_ENDPOINT`	Indicates that this tile is the type of tile that validates an endpoint, and requires the endpoint view in the next section be specified views:
Note 7	`tileId: "vrcs.my-plugin:MyViewCallableTile"`	The views of the other tiles call this tile to gather information such as the number of inputs that the endpoint can accept.

For more information about the components of bundles, see "Packaging and Deploying Plug-ins," on page 9.

Data Types and Example Data Type

You must create the data types that represent your tile inputs. Your data types represent the input and output properties, and include the name and type of data.

The data type can be a primitive type. Supported primitive types include:

- String. The most common data type used for string inputs and outputs.
- SecureString. Using SecureString is similar to the use of string, but is encrypted when it is stored, and its values are masked when they appear in the user interface.
- Boolean. True and false values.
- Integer. Any positive or negative integer
- Number. Non-integer numbers.
- JSON. Represents free-form JSON object data.
- Arrays. Can use for arrays of any data type, including custom arrays. You specify an array by adding the suffix `[]` to the data type name. For example, `String[]`.

You can wrap multiple primitive types together to create your own data types. You can nest the data types, which means that you place one custom data type inside another custom data type. An example REST data type might include the endpoint URL, user name, and user password.

You embed the data type in your plug-in. Owners of plug-ins that interact with your plug-in know that they must consume your data type.

For example, an endpoint named `MyEndpoint` has a URL, a timeout, and a username/password.

To reflect these attributes of the endpoint, you create a file named `MyEndpoint.yaml`, and store this file in the folder named `datatypes` in the bundle directory.

The file named `MyEndpoint.yaml` has the following structure:

```
---
  name: "vrcs.my-plugin:MyEndpoint"
  properties:
    url:
      type: "String"
    username:
      type: "String"
    password:
      type: "SecureString"
    timeout:
      type: "Integer"
```

When a tile requires this endpoint as an input, you can use this data type to pass all the information at once.

Endpoint data types can persist in the vRealize Code Stream to use across multiple pipelines.

Example Tiles

Before you develop your own plug-in, you must create tiles to represent the actions for your bundle to perform.

Each bundle runs multiple tiles, and each tile includes the data types and views to run your action in vRealize Code Stream.

You use several types of tiles.

- The Task tile executes code that runs in a vRealize Code Stream pipeline execution.
- The Endpoint tile validates the endpoints. For example, an endpoint can be a source location of files to download.
- The View Callable tile provides the back end work for the user interface to compose the views.

You create the Task, Endpoint, and View Callable tiles as follows:

- Create the Task tile to represent the work done by an action when the pipeline runs.
- (Optional) Create the Endpoint tile to validate any endpoint data that your plug-in requires.
- (Optional) Create the View Callable tile, which is used by the configuration, results, endpoint views.

For more information about the tiles, see "Organizing Work in a Pipeline," on page 9.

The following code is an example of how a Task tile named `MyTile.java` might function to start a remote procedure using REST on the Endpoint tile named `MyEndpoint`.

```
public class MyTile implements TileExecutable {

    private static final Logger logger = Logger.getLogger(MyTile.class.getName());

    @Override
    public void handleExecute(TileExecutableRequest request, TileExecutableResponse response) {
        // Gather inputs from the request represented as a TileProperties object
        TileProperties myEndpoint =
request.getInputProperties().getAsProperties("myEndpoint"); // See note 1.
        List<String> myInputs =
request.getInputProperties().getAsStringArray("myInputs");      // See note 2.
        MyRestClient client = new
MyRestClient(myEndpoint.getAsString("url"),                  // See note 3.

myEndpoint.getAsString("username"),             // See note 4.
```

```
myEndpoint.getAsString("password"));             // See note 5.

        if (request.isFirstExecution()) {
            // Begin an asynchronous task on MyEndpoint and inform the framework that you wish
to be called again in 1 sec
            String statusUrl = client.begin(myInputs);
            // Because the JAVA classes that represent tiles are stateless, any context relating
to the execution must be stored as an output property
            // using the prefix __ so that they are hidden from the final tile outputs. In this
case we store a url to get to check the task status.
            response.getOutputProperties().setAsString("__statusUrl", statusUrl);.
            response.setCompleted(false);
            response.setExecutionIntervalSeconds(1);
        } else {
            // Check if we should timeout the request
            int timeout = myEndpoint.getAsInteger("timeout", 10));              // See note 6.
            int duration = request.getDurationSeconds();                        // See note 7.
            if (timeout > duration) {
                response.setFailed("Failed with timeout after " + duration + " seconds");
                return;
            }

            // Monitor the asynchronous task on MyEndpoint
            int statusUrl = response.getOutputProperties.getAsString("__statusUrl");
            int myResult = client.check(statusUrl);
            if (myResult == 0) {
                // The remote procedure is running. Keep checking every 5 sec till we get a
result and update the number of attempts
                response.setProgressMessage("Process is still running");
                response.setCompleted(false);
                response.setExecutionIntervalSeconds(5);
            } else if (myResult < 0) {
                // The remote procedure has failed, get the error and communicate it back to the
user
                response.setFailed("Failed with error: " + myResult);
            } else {
                // The remote procedure succeeded, capture the output and we are done
                response.getOutputProperties().setInteger("myResult", myResult);
                response.setCompleted(true);
            }
        }
    }
}
```

Table 3-2. Details for Task Tile

Reference	Code	Description
Note 1	`TileProperties myEndpoint`	Tile properties can be nested for complex data types or JSON.
Note 2	`List<String> myInputs`	Array properties have specialized getters that return `List<T>`.
Note 3 Note 4	`myEndpoint.getAsString("url")` `myEndpoint.getAsString("username"),`	Since `myEndpoint` is a nested `tileProperties` object, we can use the same getters used on `inputProperties` to read its fields.

Table 3-2. Details for Task Tile (Continued)

Reference	Code	Description
Note 5	`myEndpoint.getAsString("password"));`	Secure strings are accessed like Strings.
Note 6	`myEndpoint.getAsInteger("timeout", 10));`	Typed getters exist for all the primitive data types, and they accept default values.
Note 7	`request.getDurationSeconds();`	The request also has metadata such as how long your tile has been running.

For concepts about using tiles to create and run actions in vRealize Code Stream pipelines, see "Organizing Work in a Pipeline," on page 9.

For more information about creating your own tiles, see "Using the Plug-In SDK to Extend vRealize Code Stream," on page 8.

Tile Views

Several views serve as the user interfaces for your plug-in. You create these views so that they can capture input and display the results when the plug-in runs.

In the sample version of `bundle.yaml`, these views are defined in the files named `config.js`, `result.js`, and `endpoint.js`.

In `bundle.yaml`, the names of the views must always be `config`, `result`, and `endpoint`. The values for the views are in the path relative to the folder named `ui`, which you can change to be specific for your needs.

The following views are typical views for each plug-in.

Table 3-3. Views for Plug-Ins

View Name	Mandatory	Owned by	Description
config	Yes	Task tile	Allows you to set up and modify the task as desired. This view is the interactive form that captures user input for the task to run. This view appears in the `Edit` and `View` modes in a pipeline.
result	Yes	Task tile	Displays the result when a user attempts to run a task. For example, if the task fails, the Result view displays information about the cause of the failure, and the inputs used so that you can debug the failure. This view is read-only. All the interactive input controls in this view will be disabled by default.
endpoint	No	Endpoint tile	This interactive form captures user input to represent and configure a remote endpoint.

For an example REST task tile, the views might provide the following information.

- Config view. Provides the input and output parameters as defined in your REST plug-in.
- Result view. Indicates that the task is in progress, and reports the results of the pipeline execution. When the task finishes, the Result view indicates success or failure.
- Endpoint view. Presents properties from the Endpoint data type that you define in your REST plug-in, such as the URL, user name, and password.

For more information about creating your own views, see "Framework to Create Views," on page 27.

To understand the sample version of `bundle.yaml`, see "Bundles and Example Bundle.yaml," on page 21.

Framework to Create Views

To simplify your experience of creating the views for your tiles, the plug-in SDK includes a schema-based framework on top of Alpaca JS.

With the schema-based framework, you can easily generate highly interactive forms based on a set of JSON schema without getting too involved with the HTML and CSS.

You define your Alpaca configuration in JavaScript. The plug-in framework provided with the SDK generates any needed HTML, and injects all the necessary JavaScript and CSS dependencies into the page. For more information, see the Alpaca documentation at http://alpacajs.org/documentation.html.

The view framework includes the following third-party JavaScript libraries that you can use directly in your views:

- jQuery 1.11.1
- Bootstrap 3.1.1
- Bootstrap Tokenfield 0.12.1
- Ace.js 1.2.5

For example plug-in code, see the documentation in GitHub at `https://github.com/vmwaresamples/vrcs-sdk-samples`.

After you develop the code for your plug-in, to run the plug-in, you compile your bundle, and deploy it in your vRealize Code Stream appliance. See "Tile Views," on page 26.

Developing Your Own Plug-In Using Maven Archetype

You can create your own plug-in from the Maven archetype provided with the plug-in SDK.

You build an archetype, which creates a blank boilerplate plug-in. To create a plug-in from the archetype, follow this procedure.

For more information to create your plug-in from a sample plug-in, see the documentation in GitHub at `https://github.com/vmwaresamples/vrcs-sdk-samples`.

Prerequisites

- Become familiar with the properties used to create plug-ins from the archetype, and the required naming conventions. See "Properties for Archetype Plug-Ins," on page 28.

Procedure

1. On the machine where you cloned the plug-in SDK, create a folder where you will create your own plug-in.

   You can generate your own plug-in from any folder.

2. To build a new archetype, run the following command.

   ```
   mvn archetype:generate -DarchetypeArtifactId=vrcs-simple-plugin-archetype -
   DarchetypeVersion=2.2.0 -DarchetypeGroupId=com.vmware.vrcs
   ```

3. Follow the prompts and enter the properties for the plug-in, and use the required naming conventions, as defined in "Properties for Archetype Plug-Ins," on page 28.

4. When you are prompted, to agree with the values enter **`Y`**. Or, to change the values, enter **`N`**.

   The result indicates that the build was successful.

5 Verify that the build created your plug-in and file structure.

   a Open the plug-in directory, and verify that the `pom.xml` and the directory named `src` appear.

   b Open the directory named `src`, and verify that a directory named `main` appears.

   This directories named `bundle` and `java` appear.

6 Deploy your plug-in to your vRealize Code Stream appliance.

   a Open the plug-in bundle directory for your archetype plug-in.

   b To build and deploy the bundle, enter the following command, and replace **`VRCS_APPLIANCE`** with your own vRealize Code Stream appliance.

   ```
   mvn clean deploy –Dbundle.deploy.host=VRCS_APPLIANCE
   ```

   This command deploys the bundle without reloading the plug-in.

   c On your vRealize Code Stream appliance, verify that your plug-in was deployed.

7 Log in to your vRealize Code Stream appliance and create your pipeline.

   a Log in to the appliance where you deployed your plug-in, and include the debug option.

   For example, log in to your appliance using the form:
   `mycodestream–20.yourcompany.com/vcac/org?debug=true`

   b In your vRealize Code Stream appliance, click the **Code Stream** tab.

   c Click the **Endpoints** tab, click **Add**, and select your endpoint from the list.

   d Click **Pipelines**, and click **Add**.

   e Enter a name for the new pipeline, and click **Stages**.

   f Click **Add Stage**, and in the new stage, click **Add Task**.

   g From the list of tasks, click your new task. The task name is based on the name of the plug-in that you created.

You have created your own plug-in from the archetype provided with the plug-in SDK, and deployed your new plug-in to your vRealize Code Stream appliance. The plug-in does not run any action until you add some code to your Task tile.

What to do next

Finish creating your pipeline with this task. Then, run the pipeline and view the results.

Properties for Archetype Plug-Ins

When you create a plug-in from Archetype, you must define the value for the required properties.

The following properties indicate the required properties and examples. When you enter the properties, you must follow the naming conventions.

Table 3-4. Properties for Archetype Plug-Ins

Property	Example	Explanation	Naming Convention
groupId	com.vmware.vrcs	Identifies the project uniquely across all projects.	Must follow the package name rules. You must control the domain name. You can create as many subgroups as desired.
artifactId	simple-sample-plugin	Name of the JAR file without a version number. For a third-party JAR, you must use the name of the JAR as it is distributed.	Choose any name desired, with lowercase letters, and without spaces or symbols.
version	1.0.0	If you distribute the plug-in, you can choose any typical version with numbers and dots, such as 1.0, 1.1, 1.0.1.	Do not use dates, because they are associated with SNAPSHOT (nightly) builds.
package	com.vmware.vrcs.tile	A grouping of related types that provide access protection and name space management.	Package names are written in all lower case. Do not start with a digit or other character that is illegal to use as the beginning of a Java name.
vrcsEndpointID	simpleEndpoint	A unique identifier for the endpoint tile.	Choose any name desired, with lowercase letters, and without spaces or symbols.
vrcsEndpointName	Simple Endpoint	Name of the endpoint that appears at the user interface.	Choose any name desired.
vrcsEndpointType	simpleServer	A unique identifier of the endpoint data type.	Choose any name desired, with lowercase letters, and without spaces or symbols.
vrcsPlugin Author	VMware Inc	Author of the plug-in.	Choose any name desired.
vrcsPluginID	simple-sample	A unique identifier for the plug-in across all plug-ins.	Choose any name desired, with lowercase letters, and without spaces or symbols.
vrcsPluginNamespace	vrcs	A set of symbols that are used to organize objects of various kinds, so that these objects may be referred to by name.	Choose any name desired, with lowercase letters, and without spaces or symbols.
vrcsTaskID	build_job	A unique identifier for the task tile.	Choose any name desired, with lowercase letters, and without spaces or symbols.
vrcsTaskName	Build Job	Name of the task that appears at the user interface.	Choose any name desired.

Developing Your Own Plug-In By Modifying the SDK Sample

The plug-in that is included with the plug-in SDK, which you clone from GitHub, provides a sample implementation to develop your own plug-in.

You can copy the sample plug-in to any folder on your development machine, even outside of the SDK folder structure, as long as it is within a Maven project.

Procedure

1 Copy the rest-sample-plugin to your own workspace.

2 Update the SDK root `pom.xml` file and include the path to the folder.

This step ensures that your plug-in can be built.

What to do next

For more information to develop your own plug-in from the sample, see the documentation at `https://github.com/vmwaresamples/vrcs-sdk-samples`.

Testing and Debugging Your Tiles and Views

If a tile or view for your plug-in fails, you can use the debug options to troubleshoot your plug-in and the Java code in real time.

In your development appliance, you can debug your tiles in real time. You use SSH to log in to your vRealize Code Stream appliance, edit the runtime for your plug-in, enable the debug value, and restart the appliance for the changes to take effect.

The views for your plug-in are cached in the browser. As a result, the changes might not always be visible on the view, even if you modify your code and rebuild your plug-in. If this behavior occurs, you must pass a special query parameter to the URL for your vRealize Code Stream appliance.

Use the procedures that follow to run your tiles in debug mode, and to debug the views for your tiles.

Caution Do not enable debugging in your production environment.

Debug Tile Views

You can debug the views for your tiles when you access your vRealize Code Stream appliance.

To debug the views for your tiles, you modify the URL when you access your vRealize Code Stream appliance.

Procedure

- From your development machine, enter the following URL and update it with your own appliance name:

  ```
  https://your.vcac.appliance.url/vcac/org/qe/?debug=true&code=...
  ```

After you reload the page, the modified view loads, and more debug logs appear in your browser developer console.

When you log in to your appliance again, `debug=true` is not added to the URL by default. If you notice that an item on the view is not updated, you must watch the debug flag during development.

Run Tiles in Debug Mode

Use the following procedure to modify your development appliance so that you can run your tiles in debug mode.

To enable debug mode, you modify the debug setting in the runtime file for the plug-in.

Note The remote debug server runs at VRCS_APPLIANCE:8001, and you can connect your IDE of choice.

Procedure

1 From your development machine, log in to your vRealize Code Stream appliance using SSH:

  ```
  ssh root@VRCS_APPLIANCE
  ```

2 Edit the runtime control file:

```
vi /usr/local/tekton/tektonruntime-ctl.sh
```

3 To enable the debug option, modify the setting:

```
DEBUG_ENABLED=yes
```

4 To restart the server, enter:

```
service tekton-server restart
```

You have enabled debug mode, and restarted the appliance for the changes to take effect.

Using Logs for Troubleshooting

You can use the following logs for troubleshooting problems that occur.

Log	URL	Location
Plug-in Logs	`/var/log/vmware/tekton/tekton_server.log on the vRCS appliance`	vRealize Code Stream appliance
vRealize Code Stream Logs	`/var/log/vcac/catalina.out`	vRealize Code Stream appliance
View Logs	Available when `debug=true` flag is set.	Browser Developer Console

Index

U

V