Application Accelerator for VMware Tanzu

Application Accelerator for VMware Tanzu 1.0

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

https://docs.vmware.com/

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

Contents

Application Accelerator for VMware Tanzu v1.0

1

This chapter includes the following topics:

Application Accelerator overview

Application Accelerator for VMware Tanzu helps you bootstrap developing your applications and deploying them in a discoverable and repeatable way.

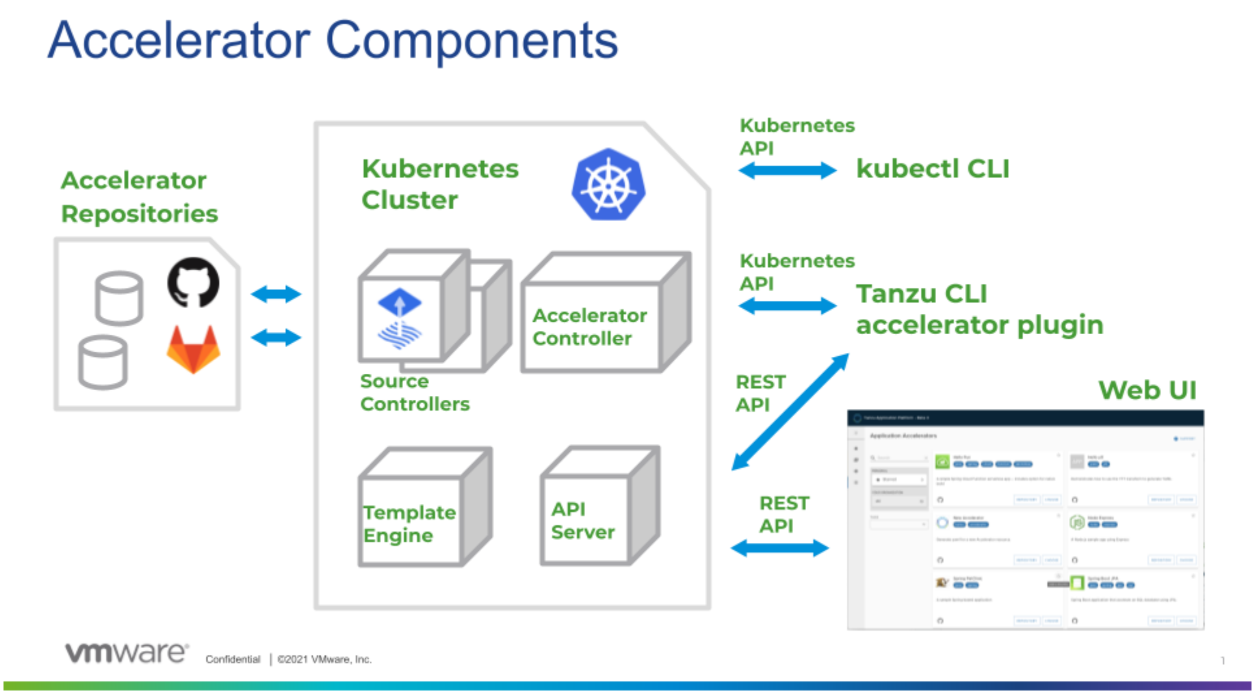
Enterprise Architects author and publish accelerator projects that provide developers and operators in their organization ready-made, enterprise-conformant code and configurations.

Published accelerators projects are maintained in Git repositories. You can then use Application Accelerator to create new projects based on those accelerator projects.

The Application Accelerator user interface(UI) enables you to discover available accelerators, configure them, and generate new projects to download.

Architecture

The following diagram illustrates the Application Accelerator architecture.



How does Application Accelerator work?

Application Accelerator allows you to generate new projects from files in Git repositories. An `accelerator.yaml` file in the repository declares input options for the accelerator. This file also contains instructions for processing the files when you generate a new project.

Accelerator custom resources (CRs) control which repositories appear in the Application Accelerator UI. You can maintain CRs by using Kubernetes tools such as kubectl or by using the Tanzu CLI accelerator commands. The Accelerator controller reconciles the CRs with a Flux2 Source Controller to fetch files from GitHub or GitLab.

The Application Accelerator UI gives you a searchable list of accelerators to choose from. After you select an accelerator, the UI presents input fields for any input options.

Application Accelerator sends the input values to the Accelerator Engine for processing. The Engine then returns the project in a ZIP file. You can open the project in your favorite integrated development environment(IDE) to develop further.

Next steps

Learn more about:

- Chapter 4 Creating accelerators

Application accelerator release notes

2

This documentation contains product features that are currently under development. Features are subject to change, and must not be in contracts, purchase orders, or sales agreements of any kind. This documentation represents no commitment from VMware to deliver these features in any generally available product.

This chapter includes the following topics:

v 1.0.1

Release date: February 8, 2022

> **Note:** Application Accelerator release notes are now included in the Tanzu Application Platform Release notes.

v 1.0.0

Release date: January 11, 2022

New features

New features and resolved issues in this release:

- Upgrade log4j-api dependency to 2.16.0
- Disable the Exec Transform

- Improve App Accelerator TAP GUI plug-in refresh cycle
- Fix Accelerator loading issues for TAP GUI plug-in

v 0.5.1

Release date: December 9, 2021

New features

New features in this release:

- **Change the defaults:** Changed `server.service_type` from `ClusterIP` to `LoadBalancer`

v 0.5.0

Release Date: November 19, 2021

New features

New features in this release:

- **UI removed**: The UI is removed and rewritten to be part of Tanzu Application Platform GUI that is provided with the Tanzu Application Platform.
- **Support ReplaceText based on regex:** For more information, see Replace Text Syntax Reference
- **Change the defaults:** Changed `server.service_type` from `LoadBalancer` to `ClusterIP`
- **Use the default namespace of accelerator-system:** The CLI plug-in now defaults to using the `accelerator-system` for any commands targeting the accelerator resources.

v 0.4.0

Release Date: October 25, 2021

New features

New features in this release:

- **Support for SourceImage:** Allow Accelerators to use source code from an OCI image instead of a Git repository. The CLI also adds a `--local-path` option for create command a push command for pushing local directory content to an OCI image registry.
- **Added helper functions for case conversions:** For more information, see Replace Text Syntax Reference.

  Note: The following release notes are for v0.1.0, v0.2.0 and v0.3.0 also apply to the v0.4.0 release.

v 0.3.0

Release Date: September 27, 2021

New Features

New features in this release:

- **Application Accelerator CLI:** For more information, see Chapter 6 Application Accelerator CLI.
- **Replace text with the contents of a file:** For more information, see Replace Text Syntax Reference.

V 0.2.0

Release Date: August 25, 2021

New features

New features in this release:

- **multi-text inputType in accelerator options:** For more information, see Accelerator options.
- **dependsOn accelerator option property:** For more information, see Accelerator options.
- **interval specification:** The `interval` specification in the Accelerator custom resource configuration file controls the Git repository polling rate. For more information, see API definitions.
- **extraArgs property for YTT transform:** For more information, see Using extraArgs.
- Changes to the REST API exposed by the UI server:
  - GET `/api/accelerators`
  - GET `/api/accelerators/options?name={accelerator-name}`
  - POST `/api/accelerators/zip?name={accelerator-name}`

Breaking changes

Breaking changes in this release:

- **New API group:** Update existing accelerator custom resource configuration files to reflect the new API group `accelerator.apps.tanzu.vmware.com/v1alpha1`. For more information, see Accelerator Custom Resource Definition.
- **Removal of #projectDescription:** `#projectDescription` no longer exists. For existing accelerators that depend on this previously automatic option, create an explicit text option to prompt

V 0.1.0

Release date: July 20, 2021

Known issues

Known issues in this release:

- **Lack of SSO:** The App Accelerator User Interface is unauthenticated. Deployments must not be exposed to the public Internet. VMware is investigating how best to provide SSO (single sign-on) integration.
- **I/O timeout:** If you see an error such as the one following one, you must deactivate the Flux network policy when installing the Flux Source Controller:

  ```
  failed to extract metadata from archive
  Get “http://source-controller.flux-system.svc.cluster.local./gitrepository/default/new-
  accelerator-acc-bjp52/8e78c60837c69ea6350f9196b7eeaf5de7c14deb.tar.gz:
  dial tcp 198.57.214.29:80: i/o timeout
  ```

For more information, see Chapter 3 Installing Application Accelerator for Tanzu Application Platform.

Installing Application Accelerator for Tanzu Application Platform

3

Application Accelerator is a component of VMware Tanzu Application Platform. For information about installing Tanzu Application Platform, see the Tanzu Application Platform documentation.

You can navigate to and interact with the available Accelerator resources using |Tanzu Application Platform GUI.

This chapter includes the following topics:

Uninstalling Application Accelerator for Tanzu Application Platform

Application Accelerator is a component of Tanzu Application Platform. To uninstall Tanzu Application Platform, see the Tanzu Application Platform documentation.

Creating accelerators 4

This topic describes how to create an accelerator in Tanzu Application Platform GUI. An accelerator contains your enterprise-conformant code and configurations that developers can use to create new projects that automatically follow the standards defined in your accelerators.

This chapter includes the following topics:

Prerequisites

The following prerequisites are required to create an accelerator:

- Application Accelerator is installed. For information about installing Application Accelerator, see Chapter 3 Installing Application Accelerator for Tanzu Application Platform

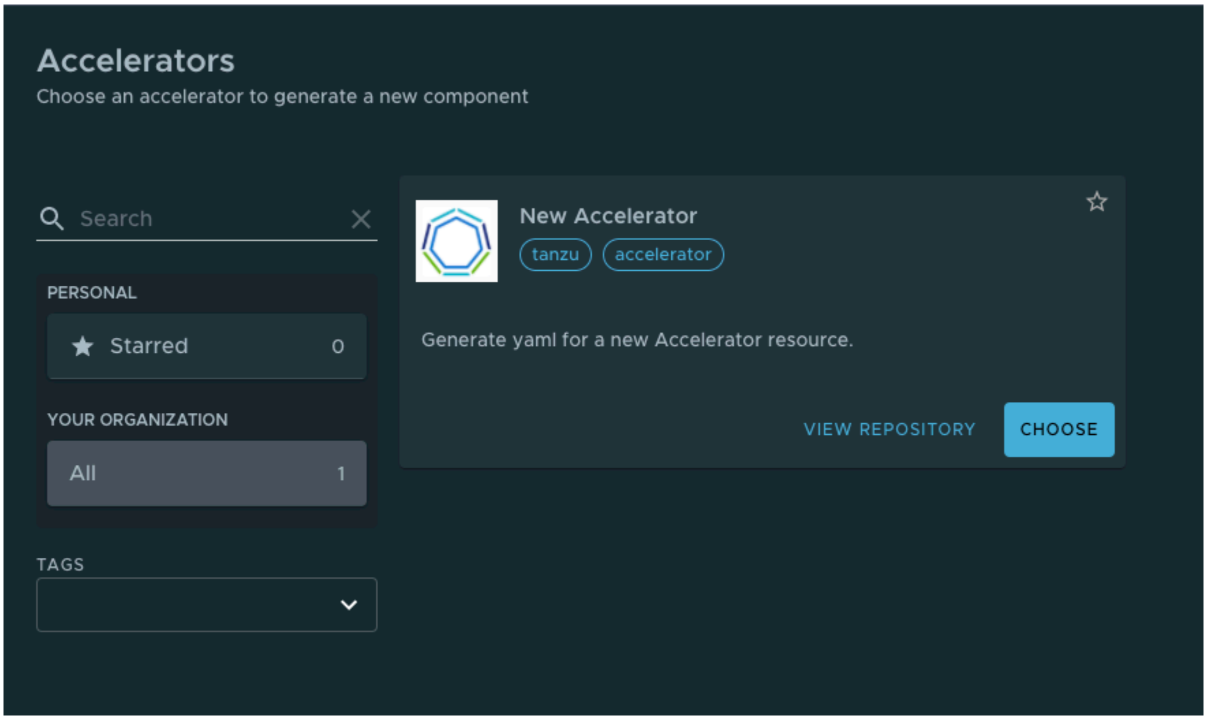
- You can access Tanzu Application Platform GUI from a browser. For more information, see the "Tanzu Application Platform GUI" section in the most recent release for Tanzu Application Platform documentation
- kubectl v1.20 and later. The Kubernetes command line tool (kubectl) is installed and authenticated with admin rights for your target cluster.

Getting started

You can use any Git repository to create an accelerator. You need the URL for the repository to create an accelerator.

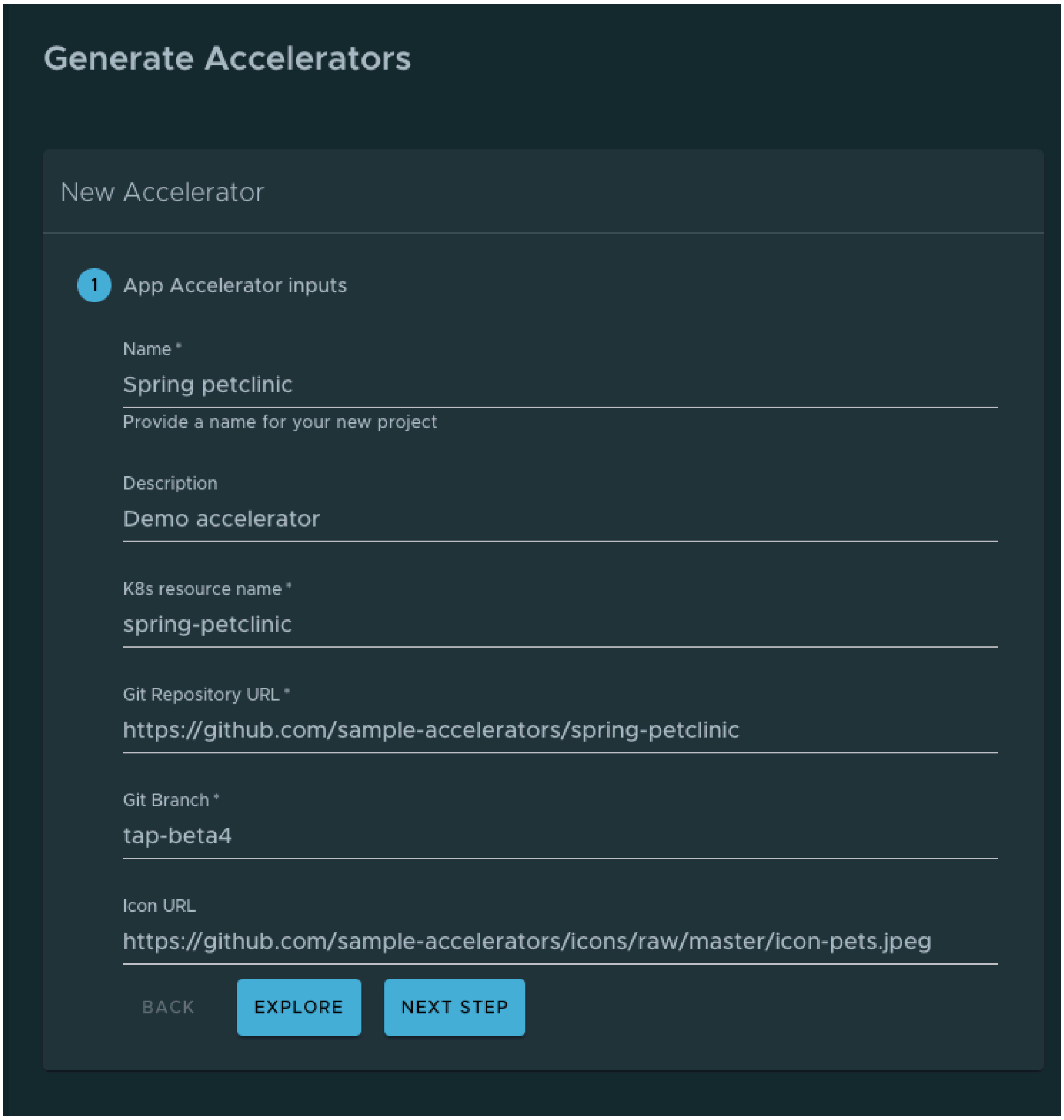
Use the following procedure to create an accelerator:

1 Select **New Accelerator** in the Application Accelerator web UI.



2 Fill in the new project form with the following information:

Field	Description
Name	Your accelerator name.
(Optional) Description	A description of your accelerator.
K8s Resource Name	A Kubernetes resource name.
Git Repository URL	The Git repository URL you use to create an accelerator.
Git Branch	The name of your Git branch.
(Optional) Icon URL	URL of an image to be used as the icon for the accelerator.



Generate Accelerators

New Accelerator

1 App Accelerator inputs

Name *
Spring petclinic
Provide a name for your new project

Description
Demo accelerator

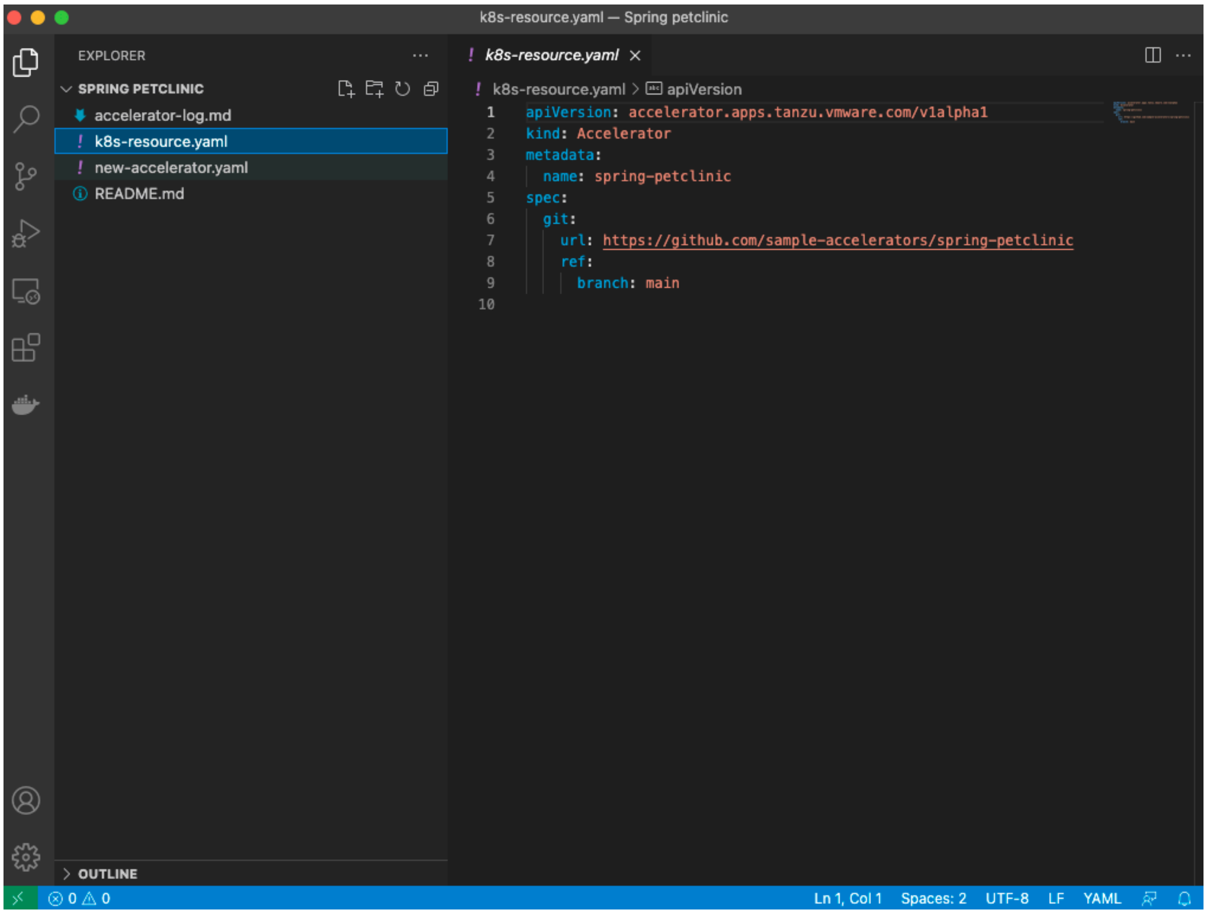
K8s resource name *
spring-petclinic

Git Repository URL *
https://github.com/sample-accelerators/spring-petclinic

Git Branch *
tap-beta4

Icon URL
https://github.com/sample-accelerators/icons/raw/master/icon-pets.jpeg

3 Click **Generate Project** and expand the ZIP file that contains accelerator files.


k8s-resource.yaml — Spring petclinic
EXPLORER
SPRING PETCLINIC
accelerator-log.md
k8s-resource.yaml
new-accelerator.yaml
README.md
OUTLINE
k8s-resource.yaml > apiVersion
Ln 1, Col 1 Spaces: 2 UTF-8 LF YAML

```yaml
apiVersion: accelerator.apps.tanzu.vmware.com/v1alpha1
kind: Accelerator
metadata:
  name: spring-petclinic
spec:
  git:
    url: https://github.com/sample-accelerators/spring-petclinic
    ref:
      branch: main
```

Using application.yaml

The accelerator ZIP file contains a file called `new-accelerator.yaml`. This file contains additional information about the accelerator.

```
accelerator:
  displayName: Simple Accelerator
  description: Contains just a README
  iconUrl: https://raw.githubusercontent.com/simple-starters/icons/master/icon-tanzu-light.png
  tags:
  - simple
  - README
```

Copy this file as `accelerator.yaml` into the Git repository specified earlier for `Git Repository URL`. This enables accelerator attributes to be rendered in Tanzu Application Platform GUI.

Publishing the new accelerator

The accelerator ZIP file contains a file called `k8s-resource.yaml`. This file contains the resource manifest for the new accelerator.

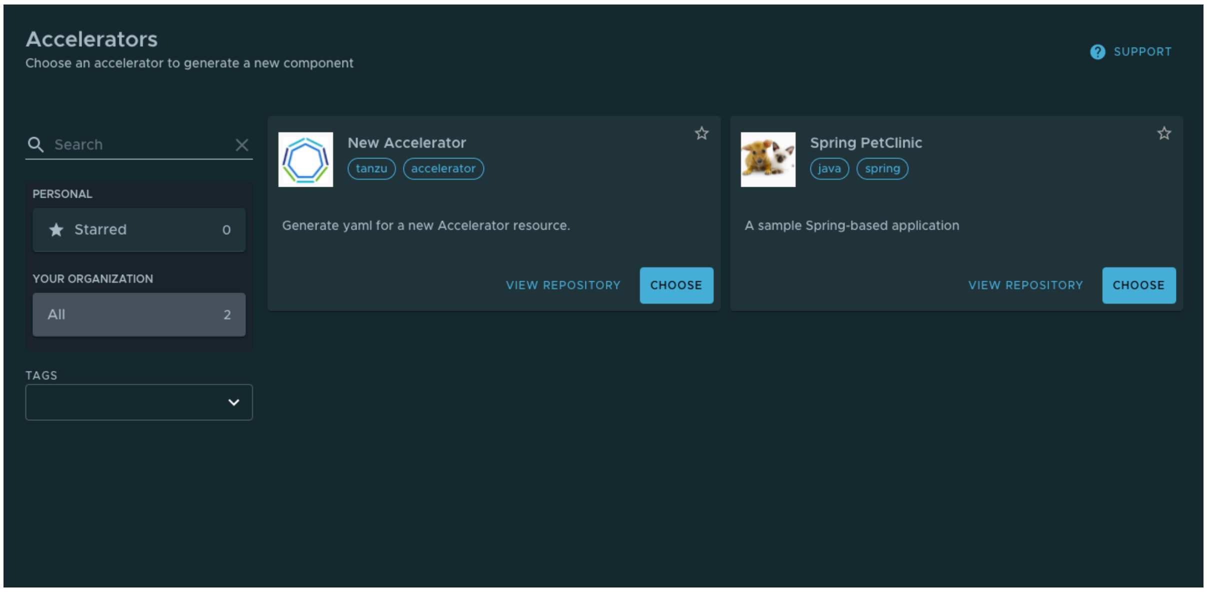
1 Review the content of the `k8s-resource.yaml` file:

```
apiVersion: accelerator.apps.tanzu.vmware.com/v1alpha1
kind: Accelerator
metadata:
name: spring-petclinic
spec:
    git:
    url: https://github.com/sample-accelerators/spring-petclinic
    ref:
    branch: main
```

2 To apply the `k8s-resource.yaml`, run the following command in your terminal in the folder directory where you expanded the ZIP file:

```
kubectl apply -f k8s-resource.yaml --namespace accelerator-system
```

3 Refresh Tanzu Application Platform GUI to reveal the newly published accelerator.



Generate yaml for a new Accelerator resource.

A sample Spring-based application

Note: It might take time for Tanzu Application Platform GUI to refresh the catalog and add an entry for new accelerator.

Next steps

Learn how to:

- Write an Creating an accelerator.yaml file.
- Configure accelerators with Accelerator Custom Resource Definition.

- Manipulate files using Transform Definition: A Gentle Introduction.
- Use SpEL Samples.

Creating an accelerator.yaml file

This topic describes how to create an accelerator.yaml file. By including an `accelerator.yaml` file in your Accelerator repository, you can declare input options that users fill in using a form in the UI. Those option values control processing by the template engine before it returns the zipped output files. For more information, see the Sample accelerator.

When there is no `accelerator.yaml`, the repository still works as an accelerator but the files are passed unmodified to users.

`accelerator.yaml` has two top-level sections: `accelerator` and `engine`.

Accelerator

This section documents how an accelerator is presented to users in the web UI. For example:

```
accelerator:
  displayName: Hello Fun
  description: A simple Spring Cloud Function serverless app
  iconUrl: https://raw.githubusercontent.com/simple-starters/icons/master/icon-cloud.png
  tags:
  - java
  - spring

  options:
  - name: deploymentType
    inputType: select
    choices:
    - value: none
      text: Skip Kubernetes deployment
    - value: k8s-simple
      text: Kubernetes deployment and service
    - value: knative
      text: Knative service
    defaultValue: k8s-simple
    required: true
```

Accelerator metadata

These properties are in accelerator listings such as the web UI:

- **displayName**: A human-readable name.
- **description**: A more detailed description.
- **iconUrl**: A URL pointing to an icon image.
- **tags**: A list of tags used to filter accelerators.

Accelerator options

The list of options is passed to the UI to create input fields for each option.

The following option properties are used by both the UI and the engine.

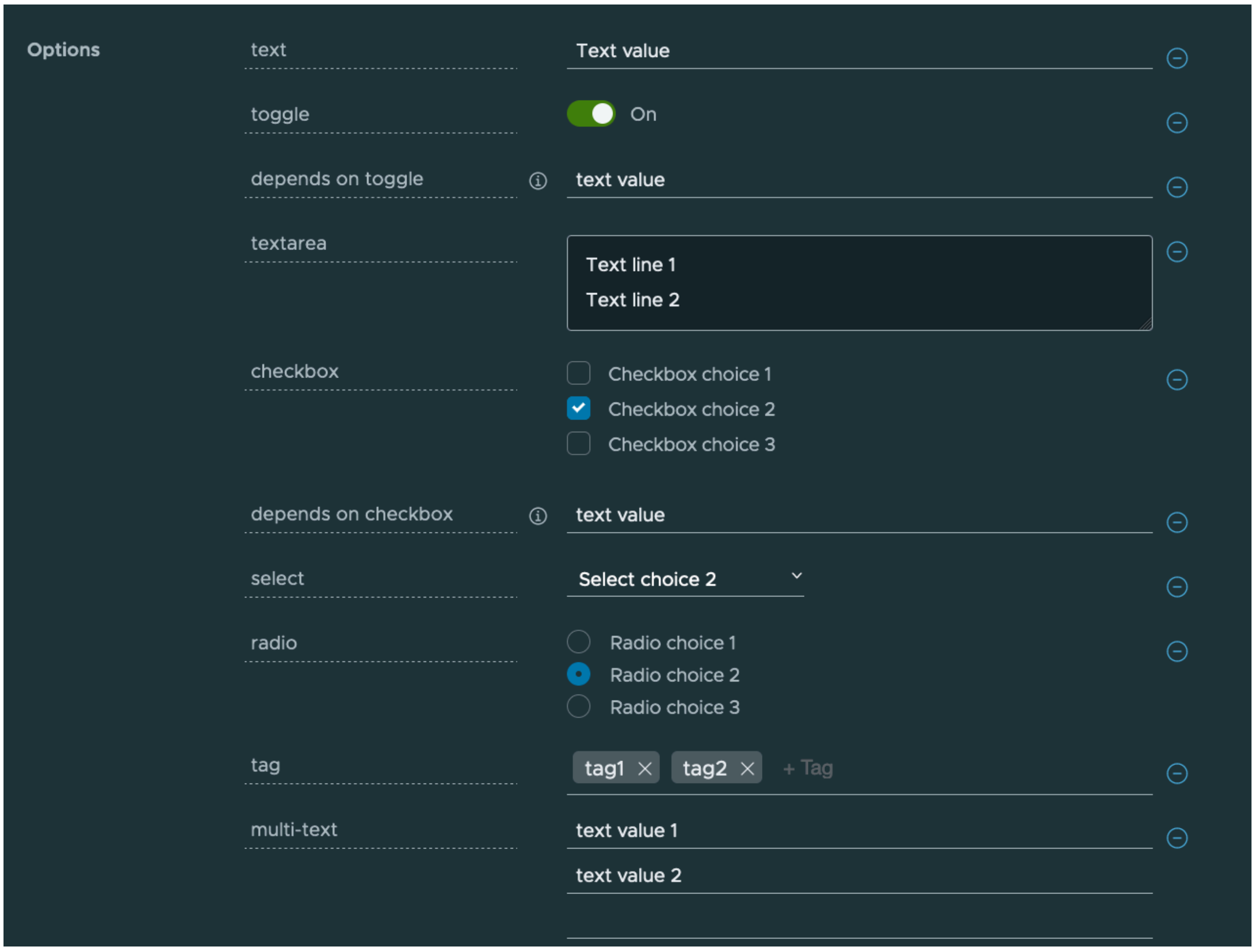
- **name**:Each option must have a unique, camelCase name. The option value entered by a user is made available as a SpEL Samples variable name. For example, `#deploymentType`.
- **dataType**:Data types that work with the UI are `string`, `boolean`, `number` and arrays of those, as in `[string]`, `[number]`, *etc.* Most input types return a string, which is the default. Use Boolean values with `checkbox`.
- **defaultValue**:This literal value pre-populates the option. Ensure its type matches the dataType. For example, use `["text 1", "text 2"]` for the dataType `[string]`. Options without a `defaultValue` can trigger a processing error if the user doesn't provide a value for that option.

The following option properties are for UI purposes only.

- **label**: A human-readable version of the `name` identifying the option.
- **description**: A tooltip to accompany the input.
- **inputType**:
  - `text`: The default input type.
  - `textarea`: Single text value with larger input that allows line breaks.
  - `checkbox`: Ideal for Boolean values or multi-value selection from choices.
  - `select`: Single-value selection from choices using a drop-down menu.
  - `radio`: Alternative single-value selection from choices using buttons.
- **choices**:This is a list of predefined choices. Users can select from the list in the UI. Choices are supported by `checkbox`, `select`, and `radio`. Each choice must have a `text` property for the displayed text, and a `value` property for the value that the form returns for that choice. The list is presented in the UI in the same order as it is declared in `accelerator.yaml`.
- **required**: `true` forces users to enter a value in the UI.
- **dependsOn**:This is a way to control visibility by specifying the `name` and optional `value` of another input option. When the other option has a matching value, or any value if no `value` is specified, then the option with `dependsOn` is visible. Otherwise, it is hidden. Ensure the value matches the dataType of the `dependsOn` option. For example, a multi-value option such as a `checkbox` uses `[matched-value]`.

Examples

The screenshot and `accelerator.yaml` file snippet that follows demonstrates each `inputType`. You can also see the sample demo-input-types on GitHub.


Options

text	Text value
toggle	On
depends on toggle	text value
textarea	Text line 1 Text line 2
checkbox	☐ Checkbox choice 1 ✓ Checkbox choice 2 ☐ Checkbox choice 3
depends on checkbox	text value
select	Select choice 2
radio	○ Radio choice 1 ● Radio choice 2 ○ Radio choice 3
tag	tag1 tag2 + Tag
multi-text	text value 1 text value 2

```
accelerator:
  displayName: Demo Input Types
  description: "Accelerator with options for each inputType"
  iconUrl: https://raw.githubusercontent.com/sample-accelerators/icons/master/icon-tanzu-
light.png
  tags: ["demo", "options"]

  options:

  - name: text
    display: true
    defaultValue: Text value

  - name: toggle
    display: true
    dataType: boolean
    defaultValue: true

  - name: dependsOnToggle
    label: 'depends on toggle'
    description: Visibility depends on the value of the toggle option being true.
    dependsOn:
      name: toggle
    defaultValue: text value
  - name: textarea
    inputType: textarea
    display: true
    defaultValue: |
      Text line 1
      Text line 2

  - name: checkbox
    inputType: checkbox
    display: true
    dataType: [string]
    defaultValue:
      - value-2
    choices:
      - text: Checkbox choice 1
        value: value-1
      - text: Checkbox choice 2
        value: value-2
      - text: Checkbox choice 3
        value: value-3

  - name: dependsOnCheckbox
    label: 'depends on checkbox'
    description: Visibility depends on the checkbox option containing a checked value value-2.
    dependsOn:
      name: checkbox
      value: [value-2]
    defaultValue: text value

  - name: select
    inputType: select
    display: true
    defaultValue: value-2
    choices:
      - text: Select choice 1
        value: value-1
      - text: Select choice 2
        value: value-2
      - text: Select choice 3
        value: value-3

  - name: radio
    inputType: radio
    display: true
    defaultValue: value-2
    choices:
      - text: Radio choice 1
        value: value-1
      - text: Radio choice 2
        value: value-2
      - text: Radio choice 3
        value: value-3
```

```
engine:
  type: YTT
```

Engine

The engine section describes how to take the files from the accelerator root directory and transform them into the contents of a generated project file.

The YAML notation here defines what is called a transform. A transform is a function on a set of files. It uses a set of files as input. It produces a modified set of files as output derived from this input.

Different types of transforms do different tasks:

- Filtering the set of files: that is, removing or keeping, only files that match certain criteria.
- Changing the contents of files. For example, replacing some strings in the files.
- Renaming or moving files: that is, changing the paths of the files.

The notation also provides the composition operators `merge` and `chain`, which enable you to create more complex transforms by composing simpler transforms together.

The following is an example of what is possible. To learn the notation, see Transform Definition: A Gentle Introduction.

Engine example

```
engine:
  include:
    ["**/*.md", "**/*.xml", "**/*.gradle", "**/*.java"]
  exclude:
    ["**/secret/**"]
  let:
    - name: includePoms
      expression:
        "#buildType == 'Maven'"
    - name: includeGradle
      expression: "#buildType == 'Gradle'"
  merge:
    - condition:
        "#includeGradle"
      include: ["*.gradle"]
    - condition: "#includePoms"
      include: ["pom.xml"]
    - include: ["**/*.java", "README.md"]
      chain:
        - type: ReplaceText
          substitutions:
            - text: "Hello World!"
              with: "#greeting"
  chain:
    - type: RewritePath
      regex: (.*)simpleboot(.*)
      rewriteTo: "#g1 + #packageName + #g2"
    - type: ReplaceText
      substitutions:
        - text: simpleboot
          with: "#packageName"
  onConflict:
    Fail
```

Engine notation descriptions

This section describes the notations in the preceding example.

`engine` is the 'global' transformation node. It produces the final set of files to be zipped and returned from the accelerator. As input, it receives all the files from the accelerator repository root. The properties in this node dictate how this set of files is transformed into a final set of files zipped as the accelerator result.

`engine.include` filters the set of files, retaining only those matching a given list of path patterns. This ensures that that the accelerator only detects files in the repository that match the list of patterns.

`engine.exclude` further restricts which files are detected. The example ensures files in any directory called `secret` are never detected.

`engine.let` defines additional variables and assigns them values. These derived symbols function such as options, but instead of being supplied from a UI widget, they are computed by the accelerator itself.

`engine.merge` executes each of its children in parallel. Each child receives a copy of the current set of input files. These are files remaining after applying the `include` and `exclude` filters. Each of the children therefore produces a set of files. All the files from all the children are then combined, as if overlaid on top of each other in the same directory. If more than one child produces a file with the same path, the transform resolves the conflict by dropping the file contents from the earlier child and keeping the contents from the later child.

`engine.merge.chain` specifies additional transformations to apply to the set of files produced by this child. In the example, `ReplaceText` is only applied to Java files and `README.md`.

`engine.chain` applies transformation to all files globally. The chain has a list of child transformations. These transformations are applied after everything else in the same node. This is the global node. The children in a chain are applied sequentially.

`engine.onConflict` specifies how conflict is handled when an operation, such as merging, produces multiple files at the same path: - `Fail` raises an error when there is a conflict. - `UseFirst` keeps the contents of the first file. - `UseLast` keeps the contents of the last file. - `Append` keeps both by using `cat <first-file> <second-file>`.

Transform Definition: A Gentle Introduction

When the accelerator is executed via the Accelerator Engine, it produces a ZIP file containing a set of files. The purpose of the `engine` section is to describe precisely how the contents of that ZIP file is to be created.

```
accelerator:
   ...
engine:
  <transform-definition>
```

Why 'Transforms'?

The result of running an accelerator is produced somehow from the contents of the accelerator itself. It is made up out of subsets of the files taken from the accelerator `<root>` directory (and its subdirectories). The files can be copied verbatim or transformed in a number of ways before being added to the result.

As such the yaml notation in the `engine` section defines a transformation that takes as input a set of files (the stuff in the `<root>` directory of the accelerator) and produces as output another set of files (the files to be put into the ZIP file).

Every transform has a `type`. Different types of transform have different behaviors and different yaml properties that control precisely what they do.

Let's look at a simple example to make this clearer. A transform of type `Include` is a 'filter. It takes as input a set of files and produces as output a subset of those files, retaining only those files whose path matches any one of a given list of `patterns`.

So if our accelerator has something like this:

```
engine:
  type: Include
  patterns: ['**/*.java']
```

Then this accelerator produces a ZIP file containing all the `.java` files from the accelerator `<root>` (or its subdirectories) but nothing else.

Transforms can also operate on the contents of a file (instead of just selecting it for inclusion).

For example:

```
type: ReplaceText
substitutions:
- text: hello-fun
  with: "#artifactId"
```

This transform looks for all occurrences of a string `hello-fun` in all its input files and replaces them with an `artifactId` (which is the result of evaluating a SpEL expression).

Combining Transforms

It should be obvious from the above examples that transforms like `ReplaceText` and `Include` are too 'primitive' to be useful just by themselves.

They are meant to be used as smaller building blocks to be composed into more complex accelerators.

To combine transforms we provide two operators called `Chain` and `Merge`. These operators are 'recursive' in the sense that they compose a number of 'child' transforms to create a more complex transform. This allows building up arbitrarily deep/complex trees of nested transform definitions.

Let's try to understand what each of these two operators does with an example; and then also try to understand the typical way that they would be used together.

Chain

Since transforms are functions whose input and output are of the same type (a set of files), you can take the ouptut of one function and feed it as input to another. This is what `Chain` does. In mathematical terms, `Chain` is *function composition*.

For an example of when/why we might want to do that, consider the `ReplaceText` transform again. Used by itself, it replaces text strings in *all* the accelerator input files. But what if we wanted to apply this replacement only to a subset of the files? Then we can use an `Include` filter to select only a subset of files of interest and chain that subset into `ReplaceText`. For example:

```
type: Chain
transformations:
- type: Include
  patterns: ['**/pom.xml']
- type: ReplaceText
  substitutions:
  - text: hello-fun
    with: "#artifactId"
```

Merge

Chaining `Include` into `ReplaceText` limits the scope of `ReplaceText` to a subset of the input files. But unfortunately it also eliminates all the other files from the result. For example:

```
engine:
  type: Chain
  transformations:
  - type: Include
    patterns: ['**/pom.xml']
  - type: ReplaceText
    substitutions:
    - text: hello-fun
      with: "#artifactId"
```

The above accelerator produces a ZIP file that only contains `pom.xml` files and nothing else.

What if we also wanted to have other files in that ZIP? Maybe we want to include some Java files as well (but we don't want to apply the same text replacement to them).

It may be tempting to write something like this:

```
engine:
  type: Chain
  transformations:
  - type: Include
    patterns: ['**/pom.xml']
  - type: ReplaceText
    ...
  - type: Include
    patterns: ['**/*.java']
```

Unfortunately that doesn't work. The reason is that if we chain non-overlapping includes together like this then the result will be an empty result set. The reason is that the first include retains only `pom.xml` files. These files are fed to the next transform in the chain. The second include only retains `.java` files, but since there are only `pom.xml` files left in the input at this point... the result is an empty set.

This is where `Merge` comes in. A `Merge` takes the outputs of several transforms executed independently on the same input sourceset and combines or 'merges' them together into a single sourceset.

So for example:

```
engine:
  type: Merge
  sources:
  - type: Chain
    - type: Include
      patterns: ['**/pom.xml']
    - type: ReplaceText
      ...
  - type: Include
    patterns: ['**/*.java']
```

The above accelerator produces a result which includes both:

- the `pom.xml` files with some text replacements applied to them.
- verbatim copies of all the `.java` files.

Shortened notation

It gets cumbersome and verbose to combine transforms like `Include`, `Exclude` and `ReplaceText` with explicit `Chain` and `Merge` operators. Also there is a 'natural' and very common composition pattern to using them (*i.e.* select an interesting subset using includes/excludes; apply a chain of additional transformations to the subset; then merge the result with other stuff produced by other transforms).

That is why we provide a 'swiss army knife' transform (aka the `Combo` transform) that combines 'Include', 'Exclude', 'Merge' and 'Chain' together in a natural way. Here's an example / template of what it looks like:

```
type: Combo
include: ['**/*.txt', '**/*.md']
exclude: ['**/secret/*']
merge:
- <transform-definition>
- ...
chain:
- <transform-definition>
- ...
```

Each of the properties in this `Combo` transform is optional (as long as you specify at least one).

Notice how each of the properties `include`, `exclude`, `merge` and `chain` corresponds to the name of a type of transform (but spelled with lower case letters).

Intuitively if you specify only one of the properties the `Combo` transform behaves exactly the same as if you used that type of transformation by itself.

So, for example:

```
merge: ...
```

Behaves the same as:

```
type: Merge
sources: ...
```

When you do specify multiple properties at the same time then the 'combo' transform composes them together in a "logical way" using a combination of Merge and Chain under the hood.

So for example:

```
include: ['**/*.txt', '**.md']
chain:
- type: ReplaceText
  ...
```

Is the same as:

```
type: Chain
transformations:
- type: Include
  patterns: ['**/*.txt', '**.md']
- type: Chain
  trasformations:
  - type: ReplaceText
    ...
```

When you use all of the properties of `Combo` at once:

```
include: I
exclude: E
merge:
- S1
- S2
chain:
- T1
- T2
```

This is equivalent to:

```
type: Chain
transformations:
- type: Include
  patterns: I
- type: Exclude
  patterns: E
- type: Merge
  sources:
  - S1
  - S2
- T1
- T2
```

TODO: Add a boxes and arrows 'picture' of the above combo transform?

A Combo of one?

Note that you can use the `Combo` as a convenient shorthand for a single type of annotation (i.e. while you *can* use it to combine multiple types, and while that is its main purpose; that doesn't mean you *have* to). For example:

```
include: ["**/*.java"]
```

This is a `Combo` transform (remember: `type: Combo` is optional). But rather than combining multiple types of transforms, it only defines the `include` property. This makes it behaves exactly the same an `Include` transform:

```
type: Include
patterns: ["**/*.java"]
```

Therefore usually it is more convenient to use a Combo transform to denote a single `Include`, `Exclude`, `Chain` or `Merge` transform as it is slightly shorter to write it as a Combo than writing it with an explicit `type:` property.

A Common Pattern with Merge Transforms

It is a common and useful pattern to use merges with overlapping contents to apply a transformation to a subset of files and then replace these changed files within a bigger context.

For example:

```
engine:
  merge:
  - include: ["**/*"]
  - include: ["**/pom.xml"]
    chain:
    - type: ReplaceText
        subsitutions: ...
```

The above accelerator will copy all files from acceleator `<root>` whilst applying some text replacements only to `pom.xml` files (other files are copied verbatim). Let's understand exactly how this works by picking it appart.

Transform A is applied to the files from accelerator `<root>`. It selects all files. Note that this *also* includes `pom.xml` files.

Transform B is *also* applied to the files from accelerator `<root>` (remember that `Merge` passes the same input independently to each of its child transforms). Transform B selects `pom.xml` files and replaces some text in them.

So both **Transform A** and **Transform B** output `pom.xml` files. The fact that both result sets contain the same file (and with different contents in them in this case) is a kind of conflict that has to be resolved. By default, `Combo` follows a very simple rule to resolve such conlicts: just take the contents from the last child. So essentially it behaves as if you overlaid both result sets one after another into the same location, and the contents of the latter are overwriting any previous files that were already placed there by the earlier.

In this example that means that, while both **Transform A** and **Transform B** produce contents for `pom.xml`, the contents from **Transform B** 'wins' so you get the version of the `pom.xml` that has text replacements applied to it (rather than the verbatim copy from **Transform A**).

Conditional Transforms

Every `<transform-definition>` can have a `condition` attribute.

```
  - condition: "#k8sConfig == 'k8s-resource-simple'"
    include: [ "kubernetes/app/*.yaml" ]
    chain:
      - type: ReplaceText
        substitutions:
         - text: hello-fun
           with: "#artifactId"
```

When a transform's condition is `false` then that particular transform is 'disabled'. What this means is that it gets replaced with a transform that 'does nothing'. Now what that exactly means is a little subtle because, perhaps surprisingly, 'doing nothing' actually means something different depending on the context you are in.

- When in the context of a 'Merge' a disabled transform behaves like something that returns an empty set. Intuition: a Merge adds stuff together using a kind of union; adding an empty set to union essentially does nothing.
- When in the context of a 'Chain' however, a disabled transform behaves like the 'identity' function instead (i.e. `lambda (x) => x`). Intuition: when you chain functions together a value is passed through all functions in succession. Thus, each function in the chain gets the chance to 'do something' by returning a different/modified value. So, if you are a function in a chain, then to 'do nothing', means 'return the input you received unchanged as your output'.

If this all sounds a bit confusing, fortunately there is an easy 'rule of thumb' you can use to understand and predict the effect a disabled transform will have in the context of your accelerator definition.

The rule is this: if a transform's condition evaluates to false, then just pretend it isn't there. In other words, your accelerator will behave the same as if you just deleted (or commented out) that transform's yaml text entirely from the accelerator definition file.

Let's look at two different examples to illustrate both cases.

Conditional 'Merge' transform

Our first example has a conditional transform in a Merge context:

```
merge:
  - condition: "#k8sConfig == 'k8s-resource-simple'"
    include: [ "kubernetes/app/*.yaml" ]
    chain:
      ...
  - include: [ "pom.xml" ]
    chain:
      ...
```

If the condition of **Transform A** above is `false` it gets replaced with an 'empty set' because it is being used in a `Merge` context. This has the same effect as if the whole of **Transform A** was deleted or commented out:

```
merge:
  - include: [ "pom.xml" ]
    chain:
      ...
```

The result in this example is that if the condition is `false`, only `pom.xml` file will end up in the result.

Conditional 'Chain' transform

In our next example some conditional transforms are used in a `Chain` context:

```
merge:
- include: [ '**/*.json' ]
  chain:
  - type: ReplaceText
    condition: '#customizeJson'
    substitutions: ...
  - type: JsonPrettyPrint
    condition: '#prettyJson'
    indent: '#jsonIndent'
```

Note: the `JsonPrettyPrint` transform type is purely 'hypothetical'. We *could* have such a transform but we don't provide it at the moment.

In the above example both **Transform A** and **Transform B** are conditional and used in a `Chain` context. **Transform A** is chained after the `include` transform. Whereas **Transform B** is chained after **Transform A**. When either of these conditions is `false`, the corresponding transform will behave like the identity function (whatever set of files it gets as input is exactly what it returns as output).

You can see this behaves in accordance with our 'rule of thumb'. For example if **Transform A**'s condtion is `false`. Then it behaves just as if **Transform A** wasn't there: **Transform A** is chained after `include` so it receives the `include`'s result, returns it unchanged, and this is passed to **Transform B**. So in orther words... the result of the `include` is passed as is to **Transform B**. This is exactly what would also happen if **Transform A** wasn't there.

A small Gotcha with using Conditionals in Merge Transforms

As discussed above, it is a useful pattern to use merges with overlapping contents. But you have to be careful using this in combination with conditional transforms.

Reconsider our earlier example:

```
engine:
  merge:
  - include: ["**/*"]
  - include: ["**/pom.xml"]
    chain:
    - type: ReplaceText
      subsitutions: ...
```

Now we add a little twist. Let's say we only wanted to include pom files if the user selects a `useMaven` option.

We might be tempted to simply add a 'condition' to **Transform B** so as to disable it when that option isn't selected:

```
engine:
  merge:
```

```
  - include: "**/*"
  - condition: '#useMaven'
    include: ["**/pom.xml"]
    chain:
    - type: ReplaceText
      subsitutions: ...
```

Sadly, this doesn't do what you might expect. The final result will *still* contain `pom.xml` files. To understand why, remember the 'rule of thumb' for disabled transforms. The rule says that, if a transform is disabled, we pretend it simply isn't there. So when `#useMaven` is `false` the example reduces to:

```
engine:
  merge:
  - include: ["**/*"]
```

This accelerator simply copies all files from acceleator `<root>` *including* `pom.xml`.

There are several ways to avoid this pitfall. One is to make sure the `pom.xml` files are not included in **Transform A** by explicitly excluding them:

```
  ...
  - include: ["**/*"]
    exclude: ["**/pom.xml"]
  ...
```

Another way is to apply the 'exclusion of pom.xml' conditionally in a `Chain` after the main transform:

```
engine:
  merge:
  - include: ["**/*"]
  - include: ["**/pom.xml"]
    chain:
    - type: ReplaceText
        subsitutions: ...
  chain:
  - condition: '!#useMaven'
    exclude: ['**/pom.xml']
```

Merge conflict

A subtlety that needs explaining is that the representation of the 'Set of files' upon which transforms operate is 'richer' than what can be physically stored on a typical file system. A key difference is that our 'Set of Files' allows for multiple files with the same path to exist at the same time. Of course, when files are initially read from a physical file system, or a ZIP file, this situation does not arise. However, as transforms are applied to this input, it is possible to produce results that have more than one file with the same path (and different contents).

We have in fact already seen some typical examples where this happens through a `merge` operation. Recall this example:

```
merge:
- include: ["**/*"]
- include: ["**/pom.xml"]
  chain:
  - type: ReplaceText
    subsitutions: ...
```

The result of the above `merge` will have two files with path `pom.xml` (assuming there was a `pom.xml` file in the input). *Transform A* produces a `pom.xml` that is a verbatim copy of the input file; *Transform B* produces a modified copy with some text replaced in it.

It is not possible to have two files on disk with the same path. Therefore this conflict has to be resolved before we can write the result to disk (or pack it into a ZIP file).

As the example shows, merges are likely to give rise to these conflicts. So it is somewhat intuitive to call this a 'Merge conflict'. It is however important to understand these kinds of conlficts can also arise from other operations such as, for example, `RewritePath`:

```
type: RewritePath
regex: '.*.md'
rewriteTo: "'docs/README.md'"
```

The above example will rename any `.md` file to `docs/README.md`. Assuming the input contains more than one `.md` file, then the output will contain multiple files with path `docs/README.md`. Again we have a conflict because we can only have one such file exist in a physical file system or ZIP file.

Resolving 'Merge' Conflicts

By default when a conflict arises the engine doesn't really do anything with it. Our internal representation for `Set of Files` allows for multiple files with the same path. Thus, the engine simply carries on manipulating the files as is. This isn't really a problem, until the files will need to be materialized to disk (or ZIP file). If a conflict is still present at that time then an error will be raised.

This means that if your accelerator produces these kinds of conflicts then they need to be resolved before files can be materialized to disk. To this end we provide the UniquePath transform transform. This transform allows specifying explicitly what should be done when more than one file has the same path. For example:

```
chain:
- type: RewritePath
  regex: '.*.md'
  rewriteTo: "'docs/README.md'"
- type: UniquePath
  strategy: Append
```

The result of the above transform is that all `.md` files are gathered up and concatenated into a single file at path `docs/README.md`. Other possible resolution strategies could be that you keep only the contents of one of the files (see Conflict resolution).

Note that Combo transform transform also comes with some convenience support for conflict resolution built in (it automatically selects the `UseLast` strategy if none is explicitly supplied. This means that in practice you probably will rarely, if ever, need to explicitly specify a conflict resolution strategy.

Understanding file ordering

As mentioned above, our 'Set of Files' represenation is richer than the files on a typical file system. We already stated that it allows for multiple files with the same path. Another way in which it is 'richer' is that the files in the set are 'ordered' (i.e. a 'FileSet' is actually more like an ordered List than like an unordered Set).

In most situations, the order of files in a 'FileSet' doesn't really matter. However in conflict resolution it *is* actually significant. If we look at the `RewritePath` example again, you might ask about the order the various `.md` files will be appended to eachother. This ordering is directly determined by the order of the files in the input set.

That begs the question 'so what is that order?'. In general, when files are read from disk to create a `FileSet`, we can not assume a specific order. Yes, the files will be read and processed in some sequential order, but the actual order is not well-defined, it depends on implementation details of the underlying file system. The accelerator engine therefore does not guarantee a specific order in this case, it only guarantees that it *preserves* whatever ordering it gets from the file system, and processes files in accordance with that order.

As an accelerator author you should probably avoid relying on the file order produced from reading directly from a file system. Thus the `RewritePath` example above is something you probably shouldn't do... *unless* you do not particularly care about the ordering of all the different sections of the produced `README.md` file.

If, however, you do care and want to control the order explicitly, then you can make use of the fact that `Merge` will process its children in order and reflect this order in the resulting output Set of Files. For example:

```
chain:
  - merge:
      - include: ['README.md']
      - include: ['DEPLOYMENT.md']
        chain:
          - type: RewritePath
            rewriteTo: "'README.md'"
  - type: UniquePath
    strategy: Append
```

In this example we *know* without a doubt that `README.md` (from the first child of `merge`) comes before `DEPLOYMENT.md` (from the second child of `merge`). So in this example we can control the merge order directly (by changing the order of the merge children).

Conclusion

This concludes our 'Gentle' introduction. This introduction was focussed on an intuitive understanding of the `<transform-definition>` notation, which is used to describe precisely how the accelerator engine should generate new project content from the files in the accelerator root.

From here on you may want to move on to reading one of the following more detailed documents:

- An exhaustive Transforms reference of all built-in transform types,
- A sample, commented accelerator.yaml to learn from a concrete example.

Transforms reference

Available transforms

Here is a list of available transforms and a brief description of their uses. You can use:

- Combo transform as a shortcut notation for many common operations. It combines the behaviors of many of the other transforms.
- Include transform to select files to operate on.
- Exclude transform to select files to operate on.
- Merge transform to work on subsets of inputs and to gather the results at the end.
- Chain transform to apply several transforms in sequence using function composition.
- Let transform to introduce new scoped variables to the model.
- ReplaceText transform to perform simple token replacement in text files.
- RewritePath transform to move files around using regular expression (regex) rules.
- OpenRewriteRecipe transform to apply Rewrite recipes, such as package rename.
- YTT transform to run the `ytt` tool on its input files and gather the result.
- UseEncoding transform to set the encoding to use when handling files as text.
- UniquePath transform to decide what to do when several files end up on the same path.

See also

- Conflict resolution

Combo transform

The `Combo` transform is the Swiss army knife of transforms. You might often use it without even realizing it. Whenever you author a node in the transform tree without specifying `type: x`, you're using `Combo`.

`Combo` combines the behaviors of Include transform, Exclude transform, Merge transform, Chain transform, UniquePath transform, and even Let transform in a way that feels natural.

Syntax reference

Here is the full syntax of `Combo`:

```
type: Combo                  # This can be omitted, because Combo is the default transform
type.
let:                         # See Let.
  - name: <string>
    expression: <SpEL expression>
  - name: <string>
    expression: <SpEL expression>
condition: <SpEL expression>
include: [<ant pattern>]     # See Include.
exclude: [<ant pattern>]     # See Exclude.
merge:                       # See Merge.
  - <m1-transform>
  - <m2-transform>
  - ...
chain:                      # See Chain.
  - <c1-transform>
  - <c2-transform>
  - ...
onConflict: <conflict resolution> # See UniquePath.
```

Behavior

A few things to know about properties of the `Combo` transform:

- They all have defaults.
- They are all optional.
- You must use at least one. An empty, unconfigured `Combo`, like such as any other transform, serves no purpose.

When you configure the `Combo` transform with all properties, it behaves as follows:

1. Applies the `include` as if it were the first element of a Chain transform. The default value is `['**']`; if not present, all files are retained.
2. Applies the `exclude` as if it were the second element of the chain. The default value is `[]`; if not present, no files are excluded. At this point of the chain, only files that match the `include`, but are not excluded by the `exclude`, remain.
3. Feeds all those files as input to all transforms declared in the `merge` property, exactly as Merge transform does. The result of that `Merge`, which is the third transform in the big chain, is another set of files. If there are no elements in `merge`, the previous result is directly fed to the next step.
4. The result of the merge step is prone to generate duplicate entries for the same `path`. So it's implicitly forwarded to a UniquePath transform check, configured with the `onConflict` Conflict resolution. The default policy is to retain files appearing later. The results of the transform that appear later in the `merge` block "win" against results appearing earlier.

5 Passes that result as the input to the Chain transform defined by the `chain` property. Put another way, the chain is prolonged with the elements defined in `chain`. If there are no elements in `chain`, it's as if the previous result was used directly.

6 If the `let` property is defined in the `Combo`, the whole execution is wrapped inside a Let transform that exposes its derived symbols.

To recap in pseudo code, a giant `Combo` behaves like this:

```
Let(symbols, in:
    Chain(
        include,
        exclude,
        Chain(Merge(<m1-transform>, <m2-transform>, ...), UniquePath(onConflict)),
        Chain(<c1-transform>, <c2-transform>, ...)
    )
)
```

You rarely use at any one time all the features that `Combo` offers. Yet `Combo` is a good way to author other common building blocks without having to write their `type: x` in full.

For example, this:

```
include: ['**/*.txt']
```

is a perfectly valid way to achieve the same effect as this:

```
type: Include
patterns: ['**/*.txt']
```

Similarly, this:

```
chain:
  - type: T1
    ...
  - type: T2
    ...
```

is often preferred over the more verbose:

```
type: Chain
transformations:
  - type: T1
    ...
  - type: T2
    ...
```

As with other transforms, the order of declaration of properties has no impact. We've used a convention that mimics the actual behavior for clarity, but the following applies **T1** and **T2** on all `.yaml` files even though we VMware has placed the `include` section after the `merge` section.

```
merge:
  - type: T1
  - type: T2
include: ["*.yaml"]
```

In other words, `Combo` applies `include` filters before `merge` irrespective of the physical order of the keys in YAML text. It's therefore a good practice to place the `include` key before the `merge` key. This makes the accelerator definition more readable, but has no effect on its execution order.

Examples

The following are typical use cases for `Combo`.

To apply separate transformations to separate sets of files. For example, to all `.yaml` files and to all `.xml` files:

```
merge:                   # This uses the Merge syntax in a first Combo.
  - include: ['*.yaml']      # This actually nests a second Combo inside the first.
    chain:
      - type: T1
      - type: T2
  - include: ['*.yaml']      # Here comes a third Combo, used as the 2nd child inside the
first
    chain:
      - type: T3
      - type: T4
```

To apply **T1** then **T2** on all `.yaml` files that are *not* in any `secret` directory:

```
include: ['**/*.yaml']
exclude: ['**/secret/**']
chain:
  - type: T1
    ..
  - type: T2
    ..
```

Include transform

The `Include` transform retains files based on their `path`, letting in *only* those files whose path matches at least one of the configured `patterns`. The contents of files, and any of their other characteristics, are unaffected.

`Include` is a basic building block seldom used as is, which makes sense if composed inside a Chain transform or a Merge transform. It is often more convenient to leverage the shorthand notation offered by Combo transform.

Syntax reference

```
type: Include
patterns: [<ant pattern>]
condition: <SpEL expression>
```

Examples

```
type: Chain
transformations:
  - type: Include
    patterns: ["**/*.yaml"]
  - type: # At this point, only yaml files are affected
```

See also

- Exclude transform
- Combo transform

Exclude transform

The `Exclude` transform retains files based on their `path`, letting everything in *except* those files whose path matches *at least* one of the configured `patterns`. The contents of files, and any of their other characteristics, are unaffected.

`Exclude` is a basic building block seldom used *as is*, which makes sense if composed inside a Chain transform or a Merge transform. It is often more convenient to leverage the shorthand notation offered by Combo transform.

Syntax reference

```
type: Exclude
patterns: [<ant pattern>]
condition: <SpEL expression>
```

Examples

```
type: Chain
transformations:
  - type: Exclude
    patterns: ["**/secret/**"]
  - type: # At this point, no file matching **/secret/** is affected.
```

See also

- Include transform
- Combo transform

Merge transform

The `Merge` transform feeds a copy of its input to several other transforms, then *merges* the results together using set union.

A `Merge` of **T1**, **T2**, and **T3** applied to input **I** results in **T1(I) ▯ T2(I) ▯ T3(I)**.

An empty merge produces nothing (▯).

Syntax reference

```
type: Merge
sources:
  - <transform>
  - <transform>
  - <transform>
  - ...
condition: <SpEL expression>
```

See also

- Combo transform is often used to express a `Merge` **and** other transformations in a shorthand syntax.

Chain transform

The `Chain` transform uses function composition to produce its final output.

A chain of **T1** then **T2** then **T3** first applies transform **T1**. It then applies **T2** to the output of **T1**, and finally applies **T3** to the output of that. In other words, **T3** o **T2** o **T1**.

An empty chain acts as function identity.

Syntax reference

```
type: Chain
transformations:
  - <transform>
  - <transform>
  - <transform>
  - ...
condition: <SpEL expression>
```

Let transform

The `Let` transform wraps another transform, creating a new scope that extends the existing scope.

SpEL expressions inside the `Let` can access variables from both the existing scope and the new scope.

> **Note:** Variables defined by the `Let` should not shadow existing variables. If they do, those existing variables won't be accessible.

Syntax reference

```
type: Let
symbols:
- name: <string>
  expression: <SpEL expression>
- ...
in: <transform> # <- new symbols are visible in here
```

Execution

The `Let` adds variables to the new scope by computation ofSpEL expressions.

```
engine:
  let:
  - name: <string>
    expression: <SpEL expression>
  - ...
```

Both a `name` and an `expression` must define each symbol where:

- `name` must be a camelCase string name. If a let *symbol* happens to have the same name as a symbol already defined in the surrounding scope, then the local symbol shadows the symbol from the surrounding scope. This makes the variable from the surrounding scope inaccessible in the remainder of the `Let` but doesn't alter its original value.
- `expression` must be a valid SpEL expression expressed as a YAML string. Be careful when using the `#` symbol for variable evaluation, because this is the comment marker in YAML. So SpEL expressions in YAML should enclose strings in quotes or rely on block style.

Symbols defined in the `Let` are evaluated in the new scope in the order they are defined. This means that symbols lower in the list can make use of the variables defined higher in the list but not the other way around.

See also

- Combo transform provides a way to declare a `Let` scope as well as other transforms in a short syntax.

ReplaceText transform

The `ReplaceText` transform allows replacing one or several text tokens in files as they are being copied to their destination. The replacement values are the result of dynamic evaluation of SpEL expressions.

This transform is text-oriented and it requires knowledge of how to interpret the stream of bytes that make up the file contents into text. All files are assumed to use `UTF-8` encoding by default, but you can use the UseEncoding transform transform upfront to specify a different charset to use on some files.

You can use `ReplaceText` transform in one of two ways:

- To replace several literal text tokens.
- To define the replacement behavior using a single regular expression, in which case the replacement SpEL expression can leverage the regex capturing group syntax.

Syntax reference

Syntax reference for replacing several literal text tokens:

```
type: ReplaceText
substitutions:
  - text: STRING
    with: SPEL-EXPRESSION
  - text: STRING
    with: SPEL-EXPRESSION
  - ..
condition: SPEL-EXPRESSION
```

Syntax reference for defining the replacement behavior using a *single* regular expression:

> **Note:** Regex is used to match the entire document. To match on a per line basis, enable multi-line mode by including `(?m)` in the regex.

```
type: ReplaceText
regex:
  pattern: REGULAR-EXPRESSION
  with: SPEL-EXPRESSION
condition: SPEL-EXPRESSION
```

In both cases, the SpEL expression can use the special `#files` helper object. This enables the replacement string to consist of the contents of an accelerator file.See the following example.

Another set of helper objects are functions of the form `xxx2Yyyy()` where `xxx` and `yyy` can take the value `camel`, `kebab`, `pascal`, or `snake`. For example, `camel2Snake()` enables changing from camelCase to snake_case.

Examples

Replacing the hardcoded string `"hello-world-app"` with the value of variable `#artifactId` in all `.md`, `.xml`, and `.yaml` files.

```
include: ['**/*.md', '**/*.xml', '**/*.yaml']
chain:
  - type: ReplaceText
    substitutions:
      - text: "hello-world-app"
        with: "#artifactId"
```

Doing the same in the `README-fr.md` and `README-de.md` files, which are encoded using the `ISO-8859-1` charset:

```
include: ['README-fr.md', 'README-de.md']
chain:
  - type: UseEncoding
    encoding: 'ISO-8859-1'
  - type: ReplaceText
    substitutions:
      - text: "hello-world-app"
        with: "#artifactId"
```

Still the same initial idea, but making sure the value appears as kebab case, while the entered `#artifactId` is using camel case:

```
include: ['**/*.md', '**/*.xml', '**/*.yaml']
chain:
  - type: ReplaceText
    substitutions:
      - text: "hello-world-app"
        with: "#camel2Kebab(#artifactId)"
```

Replacing the hardcoded string `"REPLACE-ME"` with the contents of file named after the value of the `#platform` option in `README.md`:

```
include: ['README.md']
chain:
  - type: ReplaceText
    substitutions:
      - text: "REPLACE-ME"
        with: "#files.contentsOf('snippets/install-' + #platform + '.md')"
```

See also

- UseEncoding transform

RewritePath transform

The `RewritePath` transform allows you to change the name and path of files without affecting their content.

Syntax reference

```
type: RewritePath
regex: <string>
rewriteTo: <SpEL expression>
matchOrFail: <boolean>
```

For each input file, `RewritePath` attempts to match its `path` by using the regular expression (regex) defined by the `regex` property. If the regex matches, `RewritePath` changes the `path` of the file to the evaluation result of `rewriteTo`.

`rewriteTo` is an expression that has access to the overall engine model and to variables defined by capturing groups of the regular expression. Both *named capturing groups* `(?<example>[a-z]*)` and regular *index-based* capturing groups are supported. `g0` contains the whole match, `g1` contains the first capturing group, and so on.

If the regex doesn't match, the behavior depends on the `matchOrFail` property:

- If set to `false`, which is the default, the file is left untouched.
- If set to `true`, an error occurs. This prevents misconfiguration if you expect all files coming in to match the regex. For more information about typical interactions between `RewritePath` and `Chain + Include`, see the following section, Interaction with Chain and Include.

The default value for `regex` is the following regular expression, which provides convenient access to some named capturing groups:

```
^(?<folder>.*/)?(?<filename>([^/]+?|)(?=(?<ext>\.[^/.]*)?)$)
```

Using `some/deep/nested/file.xml` as an example, the preceding regular expression captures:

- **folder:** The full folder path the file is in. In this example, `some/deep/nested/`.
- **filename:** The full name of the file, including extension *if present*. In this example, `file.xml`.
- **ext:** The last dot and extension in the filename, *if present*. In this example, `.xml`.

The default value for `rewriteTo` is the expression `#folder + #filename`, which doesn't actually rewrite paths.

Examples

The following moves all files from `src/main/java` to `sub-module/src/main/java`:

```
type: RewritePath
regex: src/main/java/(.*)
rewriteTo: "'sub-module/src/main/java' + #g1"   # 'sub-module/' + #g0 works too
```

The following flattens all files found inside the `sub-path` directory and its subdirectories, and puts them into the `flattened` folder:

```
type: RewritePath
regex: sub-path/(.*/)*(?<filename>[^/]+)
rewriteTo: "'flattened' + #filename"   # 'flattened' + #g2 would work too
```

The following turns all paths into lowercase:

```
type: RewritePath
rewriteTo: "#g0.toLowerCase()"
```

Interaction with Chain and Include

It's common to define pipelines that perform a `Chain` of transformations on a subset of files, typically selected by `Include/Exclude`:

```
- include: "**/*.java"
- chain:
    - # do something here
    - # and then here
```

If one of the transformations in the chain is a `RewritePath` operation, chances are you want the rewrite to apply to *all* files matched by the `Include`. For those typical configurations, you can set the `matchOrFail` guard to `true` to make sure the `regex` you provide indeed matches all files coming in.

See also

- UniquePath transform can be used to ensure rewritten paths don't clash with other files, or to decide which path to select if they do.

OpenRewriteRecipe transform

The `OpenRewriteRecipe` transform allows you to apply any Open Rewrite **Recipe** to a set of files and gather the results.

> **Note:** Currently, only Java related recipes are supported. The engine leverages version `7.0.0` of Open Rewrite and parses Java files using the grammar for Java 11.

Syntax reference

```
type: OpenRewriteRecipe
recipe: <string>                  # Full qualified classname of the recipe
options:
  <string>: <SpEL expression>      # Keys and values depend on the class of the recipe
  <string>: <SpEL expression>      # Refer to the documentation of said recipe
  ...
```

Example

The following example applies the ChangePackage Recipe to a set of Java files in the `com.acme` package and moves them to the value of `#companyPkg`. This is more powerful than using RewritePath transform and ReplaceText transform, as it reads the syntax of files and correctly deals with imports, fully vs. non-fully qualified names, and so on.

```
chain:
  - include: ["**/*.java"]
  - type: OpenRewriteRecipe
    recipe: org.openrewrite.java.ChangePackage
    options:
      oldFullyQualifiedPackageName: "'com.acme'"
      newFullyQualifiedPackageName: "#companyPkg"
      recursive: true
```

YTT transform

The `YTT` transform invokes the YTT template engine as an external process.

Syntax reference

```
type: YTT
extraArgs: # optional
  - <SPEL-EXPRESSION-1>
  - <SPEL-EXPRESSION-2>
  - ...
```

The `YTT` transforms yaml notation does not require any parameters. When invoked without parameters (the typical use case), then YTT transforms input is determined entirely by two things only:

1 The input files fed into the transform.

2 The current values for options and derived symbols.

Execution

YTT is invoked as an external process with the following command line:

```
ytt -f <input-folder> \
    --data-values-file <symbols.json> \
    --output-files <output-folder> \
    <extra-args>
```

The `<input-folder>` is a temporary directory into which the input files are 'materialized' (i.e. the set of files that is passed to the YTT transform as input, is written out into this directory to allow the YTT process to read them).

The `<symbols.json>` file is a temporary JSON file which the current option values and derived symbols are materialized in the form of a JSON map. This allows YTT templates in the `<input-folder>` to make use of these symbols during processing.

The `<output-folder>` is a fresh temporary directory which is empty at the time of invocation. In a typical scenario, upon completion, the output directory contains files generated by YTT.

The `<extra-args>` are additional command line arguments obtained by evaluating the SPEL expressions from the `extraArgs` attribute.

When the `ytt` process completes with a 0 exit code, then this is considered as a 'successful' execution and the contents of the output directory is then taken to be the result of the YTT Transform.

When the `ytt` process completes with a non 0 exit code; the execution of the `YTT` transform is considered to have failed and an exception is raised.

Examples

Basic invocation

When you want to execute `ytt` on the contents of the entire accelerator repository use the 'YTT' transform as your only transform in the engine declaration.

```
accelerator:
  ...
engine:
  type: YTT
```

> **Note:** To do anything beyond calling YTT then you will need to compose YTT into your accelerator flow using merge or chain combinators. This is exactly the same as composing any other type of transform.

For example, when you want to define some derived symbols as well as merge the results from YTT with results from other parts of your accelerator transform, you may reference this example:

```
engine:
  let: # Define derived symbols visible to all transforms (including YTT)
  - name: theAnswer
    expression: "41 + 1"
  merge:
  - include: ["deploy/**.yml"] # select some yaml files to process with YTT
    chain: # Chain selected yaml files to YTT
      type: YTT
  - ... include/generate other stuff to be merged alongside yaml generated by YTT...
```

The preceding example uses a combination of Chain transform and Merge transform. You can use either `Merge` or `Chain` or both to compose YTT into your accelerator flow. Which one you choose depends on how you want to use YTT as part of your larger accelerator.

Using `extraArgs`

The `extraArgs` is used to pass additional command line arguments to YTT. This is used to add file marks.

For example, the following runs YTT and renames `foo/demo.yml` file in its output to `bar/demo.yml`.

```
engine:
  type: YTT
  extraArgs: ["'--file-mark'",  "'foo/demo.yml:path=bar/demo.yml'"]
```

> **Note:** The `extraArgs` attribute expects SPEL expressions. Take care to use proper escaping of literal strings using double and single quotes (that is, `"'LITERAL-STRING'").

UseEncoding transform

When considering files in textual form (for example when doing ReplaceText transform), the engine needs to decide which encoding to use.

By default, `UTF-8` is assumed. If any files need to be handled differently, use the `UseEncoding` transform to annotate them with an explicit encoding.

> **Note:** `UseEncoding` returns an error if you apply encoding to files that have already been explicitly configured with a particular encoding.

Syntax reference

```
type: UseEncoding
encoding: <encoding>    # As recognized by the java java.nio.charset.Charset class
condition: <SpEL expression>
```

Supported encoding names include, for example, `UTF-8`, `US-ASCII`, and `ISO-8859-1`.

Example usage

`UseEncoding` is typically used as an upfront transform to, for example, ReplaceText transform in a chain:

```
type: Chain   # Or using "Combo"
transformations:
  - type: UseEncoding
    encoding: ISO-8859-1
  - type: ReplaceText
    substitutions:
      - text: "hello"
        with: "#howToSayHello"
```

See also

- ReplaceText transform

UniquePath transform

You can use the `UniquePath` transform to ensure there are no `path` conflicts between files transformed. Frequently, you'll use this at the tail of a Chain transform.

Syntax reference

```
type: UniquePath
strategy: <conflict resolution>
condition: <SpEL expression>
```

Examples

The following example concatenates the file that was originally named `DEPLOYMENT.md` to the file `README.md`:

```
chain:
  - merge:
      - include: ['README.md']
      - include: ['DEPLOYMENT.md']
```

```
        chain:
          - type: RewritePath
            rewriteTo: "'README.md'"
  - type: UniquePath
    strategy: Append
```

See also

- `UniquePath` uses a Conflict resolution strategy to decide what to do when several input files use the same `path`.
- Combo transform implicitly embeds a `UniquePath` after the Merge transform defined by its `merge` property.

Conflict resolution

This topic describes how to resolve conflicts that transforms might produce.

For instance, if you're using Merge transform (or Combo transform's `merge` syntax) or RewritePath transform, a transform can produce several files at the same `path`. The engine then needs to take an action: Should it keep the last file? Report an error? Concatenate the files together?

Such conflicts can arise for a number of reasons. You can avoid or resolve them by configuring transforms with a *conflict resolution*. For example:

- Combo transform uses UseLast by default, but you can configure it to do otherwise.
- You can explicitly end a transform Chain transform with a UniquePath transform, which by default uses Fail. This is customizable.

Syntax reference

```
type: Combo      # often omitted
merge:
  - <transform>
chain:
  - <transform>
  - ...
onConflict: <conflict resolution>  # defaults to 'UseLast'
```

```
type: Chain      # or implicitly using Combo
transformations:
  - <transform>
  - <transform>
  - type: UniquePath
    strategy: <conflict resolution>  # defaults to 'Fail'
```

Available strategies

The following values and behaviors are available:

- `Fail`: Stop processing on the first file that exhibits `path` conflicts.

- `UseFirst`: For each conflicting file, the file produced first (typically by a transform appearing earlier in the YAML definition) is retained.
- `UseLast`: For each conflicting file, the file produced last (typically by a transform appearing later in the YAML definition) is retained.
- `Append`: The conflicting versions of files are concatenated (as if using `cat file1 file2 ...`), with files produced first appearing first.

See also

- Combo transform
- UniquePath transform

SpEL Samples

This document shows some common Spring Expression Language (SpEL) use cases for the `accelerator.yaml` file.

Variables

All the values added as options in the `accelerator` section from the yaml file can be referenced as variables in the `engine` section, you can access the value using the syntax `#<option name>` *i.e.*

```
options:
  - name: foo
    dataType: string
    inputType: text
...
engine:
  - include: ["some/file.txt"]
    chain:
    - type: ReplaceText
      substitutions:
      - text: bar
        with: "#foo"
```

This sample will replace every occurrence of the text `bar` in the file `some/file.txt` for the contents of the `foo` option.

Implicit Variables

Some variables are automatically made available to the model by the engine. At the time of writing, these are the following:

- `artifactId` is a built-in value which is derived from the `projectName` passed in from the UI with spaces replaced by '_'. If that value is empty it will be set to `app`.

- `files` is a helper object which currently exposes the `contentsOf(<path>)` method. You can learn more about it in ReplaceText transform.
- `camel2Kebab` and other variations of the form `xxx2Yyyy` are a series of helper functions for dealing with changing case of words. You can learn more about them ReplaceText transform.

Conditionals

You can use boolean options for conditional in your transformations.

```
options:
  - name: numbers
      inputType: select
      choices:
        first: First Option
        second: Second Option
      defaultValue: first
...
engine:
  - include: ["some/file.txt"]
    condition: "#numbers == 'first'"
    chain:
    - type: ReplaceText
      substitutions:
      - text: bar
        with: "#foo"
```

This will replace the text only if the selected option is the first one.

Rewrite Path Concatenation

```
options:
  - name: renameTo
    dataType: string
    inputType: text
...
engine:
  - include: ["some/file.txt"]
    chain:
    - type: RewritePath
      rewriteTo: "'somewhere/' + #renameTo + '.txt'"
```

Regular Expressions

Regular expressions let you use patterns to use as a matcher for strings, here is a small example of what you can do with them.

```
options:
  - name: foo
    dataType: string
    inputType: text
    defaultValue: abcZ123
...
engine:
  - include: ["some/file.txt"]
    condition: "#foo matches '[a-z]+Z\\d+'"
    chain:
    - type: ReplaceText
      substitutions:
      - text: bar
        with: "#foo"
```

This example uses RegEx to match a string of letters that ends with a capital z and ends with any number of digits. If this condition is fulfilled, the text wil be replaced in the file `file.txt`.

Accelerator Custom Resource Definition

The `Accelerator` custom resource definition (CRD) defines any accelerator resources that should be made available to the Application Accelerator for VMware Tanzu system. It is a namespaced CRD, meaning that any resources created will belong to a namespace and in order for the resource to be available to the Application Accelerator system it must be created in the namespace that the Application Acccelerator UI Server is configured to watch.

API definitions

The `Accelerator` CRD is defined with the following properties:

Property	Value
Name	Accelerator
Group	accelerator.apps.tanzu.vmware.com
Version	v1alpha1
ShortName	acc

The `Accelerator` CRD *spec* defined in the `AcceleratorSpec` type has the following fields:

Field	Description	Required/ Optional
displayName	A short descriptive name used for an Accelerator.	Optional (*)
description	A longer description of an Accelerator.	Optional (*)
iconUrl	A URL for an image to represent the Accelerator in a UI.	Optional (*)
tags	An array of strings defining attributes of the Accelerator that can be used in a search.	Optional (*)
git	Defines the accelerator source Git repository.	Optional (***)

Field	Description	Required/ Optional
git.url	The repository URL, can be a HTTP/S or SSH address.	Optional (***)
git.ignore	Overrides the set of excluded patterns in the .sourceignore format (which is the same as .gitignore). If not provided, a default of `.git/` will be used.	Optional (**)
git.interval	The interval at which to check for repository updates. If not provided it defaults to 10 min. **Note:** There is an additional refresh interval (currently 10s) involved before accelerators may appear in the UI. There could be a 10s delay before changes are reflected in the UI.*	Optional (**)
git.ref	Git reference to checkout and monitor for changes, defaults to master branch.	Optional (**)
git.ref.branch	The Git branch to checkout, defaults to master.	Optional (**)
git.ref.commit	The Git commit SHA to checkout, if specified tag filters will be ignored.	Optional (**)
git.ref.semver	The Git tag semver expression, takes precedence over tag.	Optional (**)
git.ref.tag	The Git tag to checkout, takes precedence over branch.	Optional (**)
git.secretRef	The secret name containing the Git credentials. For HTTPS repositories the secret must contain username and password fields. For SSH repositories the secret must contain identity, identity.pub and known_hosts fields.	Optional (**)
source	Defines the source image repository.	Optional (***)
source.image	Image is a reference to an image in a remote registry.	Optional (***)
source.imagePullSecrets	ImagePullSecrets contains the names of the Kubernetes Secrets containing registry login information to resolve image metadata.	Optional
source.interval	The interval at which to check for repository updates.	Optional
source.serviceAccountName	ServiceAccountName is the name of the Kubernetes ServiceAccount used to authenticate the image pull if the service account has attached pull secrets.	Optional

* Any optional fields marked with an asterisk (*) will be populated from a field of the same name in the `accelerator` definition in the `accelerator.yaml` file if that is present in the Git repository for the accelerator.

** Any fields marked with a double asterisk (**) are part of the Flux GitRepository CRD that is documented in the Flux Source Controller Git Repositories documentation.

*** Any fields marked with a triple asterisk (***) are optional but one of `git` or `source` is required to specify the repository to use. If `git` is specified the `git.url` is required and if `source` is specified then `source.image` is required.

Excluding files

The `git.ignore` field defaults to `.git/` which is different from the defaults provided by the Flux Source Controller GitRepository implementation. You can override this, and provide your own exclusions. See the fluxcd/source-controller Excluding files documentation for additional details.

Non-public repositories

For Git repositories that aren't accessible anonymously we need to provide credentials in a Secret. For HTTPS repositories the secret must contain username and password fields. For SSH repositories the secret must contain identity, identity.pub and known_hosts fields. See the fluxcd/source-controller HTTPS authentication and fluxcd/source-controller SSH authentication documentation for additional details.

For Image repositories that aren't publicly available an image pull secret can be provided. See Using imagePullSecrets.

Examples

A minimal example could look like the following manifest:

hello-fun.yaml

```
apiVersion: accelerator.apps.tanzu.vmware.com/v1alpha1
kind: Accelerator
metadata:
  name: hello-fun
spec:
  git:
    url: https://github.com/sample-accelerators/hello-fun
    ref:
      branch: main
```

This minimal example creates an accelerator named `hello-fun`. The `displayName`, `description`, `iconUrl`, and `tags` fields will be populated based on the content under the `accelerator` key in the `accelerator.yaml` file that is found in the `main` branch of the Git repo at https://github.com/sample-accelerators/hello-fun. That file could have this content:

accelerator.yaml

```
accelerator:
  displayName: Hello Fun
  description: A simple Spring Cloud Function serverless app
  iconUrl: https://raw.githubusercontent.com/simple-starters/icons/master/icon-cloud.png
  tags:
  - java
  - spring
  - cloud
  - function
  - serverless

...
```

We can also explicitly specify the `displayName`, `description`, `iconUrl`, and `tags` fields and this would override any values provided in the accelerator's Git repository. The following example explicitly sets thos fields plus the `ignore` field as well:

my-hello-fun.yaml

```
apiVersion: accelerator.apps.tanzu.vmware.com/v1alpha1
kind: Accelerator
metadata:
  name: my-hello-fun
spec:
  displayName: My Hello Fun
  description: My own Spring Cloud Function serverless app
  iconUrl: https://github.com/simple-starters/icons/raw/master/icon-cloud.png
  tags:
    - spring
    - cloud
    - function
    - serverless
  git:
    ignore: ".git/, bin/"
    url: https://github.com/sample-accelerators/hello-fun
    ref:
      branch: test
```

Example for a private Git repo

To create an accelerator by using a private Git repo, first create a secret by using HTTP credentials or SSH credentials.

Example using http credentials

Note: For better security, use an access token as the password.

```
kubectl create secret generic https-credentials \
    --from-literal=username=<user> \
    --from-literal=password=<password>
```

https-credentials.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: https-credentials
  namespace: default
type: Opaque
data:
  username: <BASE64>
  password: <BASE64>
```

After you have the secret file, you can create the accelerator by using the `secret-ref` property:

private-acc.yaml

```
apiVersion: accelerator.apps.tanzu.vmware.com/v1alpha1
kind: Accelerator
metadata:
  name: private-acc
spec:
  displayName: private
  description: Accelerator using private repo
  git:
    url: <repository-URL>
    ref:
      branch: main
    secret-ref:
      name: https-credentials
```

Example using SSH credentials

```
ssh-keygen -q -N "" -f ./identity
ssh-keyscan github.com > ./known_hosts
kubectl create secret generic ssh-credentials \
    --from-file=./identity \
    --from-file=./identity.pub \
    --from-file=./known_hosts
```

This example assumes you don't have a key file already created. If you do, replace the values using the following format:

`--from-file=identity=<path to your identity file>`

`--from-file=identity.pub=<path to your identity.pub file>`

`--from-file=known_hosts=<path to your know_hosts file>`

ssh-credentials.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: ssh-credentials
  namespace: default
type: Opaque
data:
  identity: <BASE64>
  identity.pub: <BASE64>
  known_hosts: <BASE64>
```

private-acc-ssh.yaml

```
apiVersion: accelerator.apps.tanzu.vmware.com/v1alpha1
kind: Accelerator
metadata:
  name: private-acc
spec:
  displayName: private
  description: Accelerator using private repo
  git:
    url: <repository-URL>
    ref:
      branch: main
    secret-ref:
      name: ssh-credentials
```

Engine Specification

5

The following specifications describe the internal contracts between the engine and other components that make up App Accelerator and may be of interest to authors and users:

- Options Specification
- Engine Specification
- HTTP Endpoint Specification

Application Accelerator CLI

6

The Application Accelerator Command Line Interface (CLI) includes commands for developers and operators to create and use accelerators.

This chapter includes the following topics:

Server API connections for operators and developers

The Application Accelerator CLI must connect to a server for all provided commands except for the `help` and `version` commands.

Operators typically use **create**, **update**, and **delete** commands for managing accelerators in a Kubernetes context. These commands require a Kubernetes context where the operator is already authenticated and is authorized to create and edit the accelerator resources. Operators can also use the **get** and **list** commands by using the same authentication. For any of these commands, the operator can specify the `--context` flag to access accelerators in a specific Kubernetes context.

Developers use the **list**, **get**, and **generate** commands for using accelerators available in an Application Accelerator server. Developers use the `--server-url` to point to the Application Accelerator server they want to use. The URL depends on the configuration settings for Application Accelerator:

- For installations configured with a **shared ingress**, use `https://accelerator.<domain>` where `domain` is provided in the values file for the accelerator configuration.
- For installations using a **LoadBalancer**, look up the External IP address by using:

  ```
  kubectl get -n accelerator-system service/acc-server
  ```

  Then use `http://<External-IP>` as the URL.
- For any other configuration, you can use port forwarding by using:

  ```
  kubectl port-forward service/acc-server -n accelerator-system 8877:80
  ```

Then use `http://localhost:8877` as the URL.

The developer can set an `ACC_SERVER_URL` environment variable to avoid having to provide the same `--server-url` flag for every command. Simply run `export ACC_SERVER_URL=<URL>` for the terminal session in use. If the developer explicitly specifies the `--server-url` flag, it overrides the `ACC_SERVER_URL` environment variable if it is set.

Installation

The Application Accelerator CLI commands are part of the Tanzu CLI Accelerator Plug-in. This is shipped as part of VMware Tanzu Application Platform and can be installed alongside other Tanzu CLI plug-ins shipped with the platform. For information about installing the Tanzu CLI and bundled plug-ins, see the Tanzu Application Platform documentation.

Commands

To view a list of commands, a description of commands, and help information, run:

```
tanzu accelerator  --help
```

Accelerator commands

Manage accelerators in a Kubernetes cluster.

Usage: `tanzu accelerator [command]`

Where `[command]` is a compatible command.

Aliases:
accelerator
acc

Available Commands:

Command	Description
apply	Apply accelerator
create	Create a new accelerator
delete	Delete an accelerator
generate	Generate project from accelerator
get	Get accelerator info
list	List accelerators

Command	Description
push	Push local path to source image
update	Update an accelerator

Flags:

Flag	Description
--context name	Name of the kubeconfig context to use (default is current-context defined by kubeconfig)
-h, --help	Help for accelerator
--kubeconfig file	kubeconfig file (default is $HOME/.kube/config)

Use `tanzu accelerator [command] --help` for more information about a command.

Apply

Create or update accelerator resource by using the specified accelerator manifest file.

Usage: `tanzu accelerator apply [flags]`

Where `[flags]` is one or more compatible flags.

Examples:

```
tanzu accelerator apply --filename <path-to-accelerator-manifest>
```

Flags:

Flag	Description
-f, --filename string	path of manifest file for the accelerator
-h, --help	help for apply
-n, --namespace string	namespace for accelerators; default is "accelerator-system"

Global Flags:

Flag	Description
--context name	name of the kubeconfig context to use; default is current-context defined by kubeconfig
--kubeconfig file	kubeconfig file; default is $HOME/.kube/config

Create

Create a new accelerator resource with specified configuration.

Accelerator configuration options include:

- Git repository URL and branch/tag where accelerator code and metadata is defined
- Metadata like description, display-name, tags and icon-url

The Git repository option is required. Metadata options are optional and will override any values for the same options specified in the accelerator metadata retrieved from the Git repository.

Usage: `tanzu accelerator create [flags]`

Where `[flags]` is one or more compatible flags.

Examples:

```
tanzu accelerator create <accelerator-name> --git-repository <URL> --git-branch <branch>
```

Flags:

Flag	Description
--description string	Description of this accelerator
--display-name string	Display name for the accelerator
--git-branch string	Git repository branch to be used
--git-repository string	Git repository URL for the accelerator
--git-tag string	Git repository tag to be used
--git-branch string	Git repository branch to be used
-h, --help	Help for create
--icon-url string	URL for icon to use with the accelerator
--interval string	Interval for checking for updates to Git or image repository
--local-path string	Path to the directory containing the source for the accelerator
-n, --namespace name	Namespace for accelerators (default "accelerator-system")
--secret-ref string	Name of secret containing credentials for private Git or image repository
--source-image string	Name of the source image for the accelerator
--tags strings	Tags that can be used to search for accelerators

Global Flags:

Flag	Description
--context name	Name of the kubeconfig context to use (default is current-context defined by kubeconfig)
--kubeconfig file	kubeconfig file (default is $HOME/.kube/config)

Delete

Delete the accelerator resource with the specified name.

Usage: `tanzu accelerator delete [flags]`

Where `[flags]` is one or more compatible flags.

Examples:

```
tanzu accelerator delete <accelerator-name>
```

Flags:

Flag	Description
-h, --help	Help for delete
-n, --namespace name	Namespace for accelerators (default "accelerator-system")

Global Flags:

Flag	Description
--context name	Name of the kubeconfig context to use (default is current-context defined by kubeconfig)
--kubeconfig file	kubeconfig file (default is $HOME/.kube/config)

Generate

Generate a project from an accelerator using provided options and download project artifacts as a ZIP file.

Generation options are provided as a JSON string and should match the metadata options that are specified for the accelerator used for the generation. The options can include "projectName" which defaults to the name of the accelerator. This "projectName" will be used as the name of the generated ZIP file.

You can see the available options by using the "tanzu accelerator list " command.

Here is an example of an options JSON string that specifies the "projectName" and an "includeKubernetes" boolean flag:

`--options '{"projectName":"test", "includeKubernetes": true}'`

You can also provide a file that specifies the JSON string using the `--options-file` flag.

The generate command needs access to the Application Accelerator server. You can specify the `--server-url` flag or set an `ACC_SERVER_URL` environment variable. If you specify the `--server-url` flag it will override the `ACC_SERVER_URL` environmnet variable if it is set.

Usage: `tanzu accelerator generate [flags]`

Where `[flags]` is one or more compatible flags.

Examples:

```
tanzu accelerator generate <accelerator-name> --options '{"projectName":"test"}'
```

Flags:

Flag	Description
-h, --help	Help for generate
--options string	Options JSON string
--options-file string	Path to file containing options JSON string
--output-dir string	Directory that the zip file will be written to
--server-url string	The URL for the Application Accelerator server

Global Flags:

Flag	Description
--context name	Name of the kubeconfig context to use (default is current-context defined by kubeconfig)
--kubeconfig file	kubeconfig file (default is $HOME/.kube/config)

Get

Get accelerator info.

You can choose to get the accelerator from the Application Accelerator server using `--server-url` flag or from a Kubernetes context using `--from-context` flag. The default is to get accelerators from the Kubernetes context. To override this, you can set the `ACC_SERVER_URL` environment variable with the URL for the Application Accelerator server you want to access.

Usage: `tanzu accelerator get [flags]`

Where `[flags]` is one or more compatible flags.

Examples:

```
tanzu accelerator get <accelerator-name> --from-context
```

Flags:

Flag	Description
--from-context	Retrieve resources from current context defined in kubeconfig
-h, --help	Help for get
-n, --namespace name	Namespace for accelerators (default "accelerator-system")
--server-url string	The URL for the Application Accelerator server

Global Flags:

Flag	Description
--context name	Name of the kubeconfig context to use (default is current-context defined by kubeconfig)
--kubeconfig file	kubeconfig file (default is $HOME/.kube/config)

List

List all accelerators.

You can choose to list the accelerators from the Application Accelerator server using `--server-url` flag or from a Kubernetes context using `--from-context` flag. The default is to get accelerators from the Kubernetes context. To override this, you can set the `ACC_SERVER_URL` environment variable with the URL for the Application Acceleratior server you want to access.

Usage: `tanzu accelerator list [flags]`

Where `[flags]` is one or more compatible flags.

Examples:

```
tanzu accelerator list
```

Flags:

Flag	Description
--from-context	Retrieve resources from current context defined in kubeconfig
-h, --help	Help for list
-n, --namespace name	Namespace for accelerators (default "accelerator-system")
--server-url string	The URL for the Application Accelerator server

Global Flags:

Flag	Description
--context name	Name of the kubeconfig context to use (default is current-context defined by kubeconfig)
--kubeconfig file	kubeconfig file (default is $HOME/.kube/config)

Update

Udate an accelerator resource with the specified name using the specified configuration.

Accelerator configuration options include:

- Git repository URL and branch/tag where accelerator code and metadata is defined
- Metadata like description, display-name, tags and icon-url

The update command also provides a `--reoncile` flag that will force the accelerator to be refreshed with any changes made to the associated Git repository.

Usage: `tanzu accelerator update [flags]`

Where `[flags]` is one or more compatible flags.

Examples:

```
tanzu accelerator update <accelerator-name> --description "Lorem Ipsum"
```

Flags:

Flag	Description
--description string	Description of this accelerator
--display-name string	Display name for the accelerator
--git-branch string	Git repository branch to be used
--git-repository string	Git repository URL for the accelerator
--git-tag string	Git repository tag to be used
-h, --help	Help for update
--icon-url string	URL for icon to use with the accelerator
--interval string	Interval for checking for updates to Git or image repository
-n, --namespace name	Namespace for accelerators (default "accelerator-system")
--reconcile	Trigger a reconciliation including the associated GitRepository resource
--secret-ref string	Name of secret containing credentials for private Git or image repository
--source-image string	Name of the source image for the accelerator
--tags strings	Tags that can be used to search for accelerators

Global Flags:

Flag	Description
--context name	Name of the kubeconfig context to use (default is current-context defined by kubeconfig)
--kubeconfig file	kubeconfig file (default is $HOME/.kube/config)