:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#toolkit-library-actions] = Configuring CDK Toolkit programmatic actions :info\_titleabbrev: Configure programmatic actions :keywords: CDK Toolkit Library, Programmatic access, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code, synthesize

== [abstract]

## The {aws} CDK Toolkit Library provides programmatic interfaces for application lifecycle actions such as synthesis, deployment, and stack management. This guide explains how to use each action in your code.

// Content start

The {aws} CDK Toolkit Library provides programmatic interfaces for application lifecycle actions such as synthesis, deployment, and stack management. This guide explains how to use each action in your code.

[#toolkit-library-actions-synth] == Generating cloud assemblies with synth

The synth action generates a cloud assembly from your cloud assembly source. For more information about synthesis, see xref:configure-synth[Configure and perform CDK stack synthesis]. A cloud assembly contains the following deployment artifacts from your CDK app:

* {aws} CloudFormation templates that define your infrastructure.
* Assets such as Lambda function code or Docker images.
* Deployment metadata and configuration.

Here’s how to use the synth action to create a cloud assembly:

== [source,typescript,subs=“verbatim,attributes”]

// Create a toolkit instance const toolkit = new Toolkit();

// Create a cloud assembly source from a TypeScript app const cloudAssemblySource = await toolkit.fromCdkApp(“ts-node app.ts”);

// Generate a cloud assembly const cloudAssembly = await toolkit.synth(cloudAssemblySource);

// Use the cloud assembly for operations await toolkit.list(cloudAssembly); await toolkit.deploy(cloudAssembly); await toolkit.diff(cloudAssembly);

// Query information from the cloud assembly const template = cloudAssembly.getStack(“my-stack”).template; —

= [TIP]

# Using a cloud assembly can optimize performance when you need to perform multiple operations, since synthesis only happens once. For more information about managing cloud assemblies, including caching and disposal, see xref:toolkit-library-configure-ca-cache[Create and manage cloud assemblies].

[#toolkit-library-actions-list] == Viewing stack information with list

The list action retrieves information about the stacks in your CDK application, including their dependencies and current status. Use this action to inspect your infrastructure before deployment or to generate reports.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Get information about specific stacks const stackDetails = await toolkit.list(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“my-stack”], // Only include stacks matching this pattern } });

// Process the returned stack information for (const stack of stackDetails) { console.log(Stack: ${stack.id}, Dependencies: ${stack.dependencies}); } —

[#toolkit-library-actions-deploy] == Provisioning infrastructure with deploy

The deploy action provisions or updates your infrastructure in {aws} using the cloud assembly produced during synthesis. For an introduction to deploying, see xref:deploy[Deploy {aws} CDK applications]. You can control deployment options such as stack selection, parameter values, and rollback behavior.

== [source,typescript,subs=“verbatim,attributes”]

// Deploy stacks with parameter values await toolkit.deploy(cloudAssemblySource, { parameters: StackParameters.exactly({ “MyStack”: { “BucketName”: “amzn-s3-demo-bucket” } }) }); —

The deploy action supports different deployment methods to accommodate various workflows. For most scenarios, especially in production environments, we recommend using the default deployment method which uses CloudFormation change sets. For development environments where iteration speed is important, you can use alternative methods like hotswap.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Deploy using default deployment method (recommended for production) await toolkit.deploy(cloudAssemblySource, { parameters: StackParameters.exactly({ “MyStack”: { “BucketName”: “amzn-s3-demo-bucket” } }) });

// For development environments only: Deploy with hotswap for faster iterations // Note: We recommend using default deployment methods for production environments await toolkit.deploy(cloudAssemblySource, { deploymentMethod: { method: “hotswap”, fallback: true }, // Faster but introduces drift stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“dev-stack”] } }); —

[#toolkit-library-actions-rollback] == Reverting failed deployments with rollback

The rollback action returns a stack to its last stable state when a deployment fails and cannot be automatically reversed. Use this action to recover from failed deployments that require manual intervention.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Roll back stacks to their last stable state await toolkit.rollback(cloudAssemblySource, { orphanFailedResources: false, // When true, removes failed resources from CloudFormation management stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“failed-stack”] } }); —

[#toolkit-library-actions-watch] == Monitoring changes with watch

The watch action continuously monitors your CDK app for local file changes and automatically performs deployments or hotswaps. This creates a file watcher that runs until your code exits or is terminated.

= [WARNING]

# Hotswap deployments update resources directly without going through CloudFormation when possible, making updates faster during development. This is enabled by default for the watch command. While this speeds up the development cycle, it introduces drift between your CloudFormation templates and deployed resources. Therefore, we recommend that you don’t use hotswaps in production environments.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Start watching for changes const watcher = await toolkit.watch(cloudAssemblySource, { include: [“lib/\_\*/\_.ts”], // Only watch TypeScript files in the lib directory exclude: [“\_\*/\_.test.ts”], // Ignore test files deploymentMethod: { method: “hotswap” }, // This is the default, shown here for clarity stacks: { strategy: StackSelectionStrategy.ALL // Watch all stacks } });

// Later in your code, you can explicitly stop watching: // await watcher.dispose(); —

The watch function returns an IWatcher object that allows you to explicitly control when to stop watching. Call the dispose() method on this object when you want to end the watch process.

[#toolkit-library-actions-destroy] == Removing infrastructure with destroy

The destroy action removes CDK stacks and their associated resources from {aws}. Use this action to clean up resources when they’re no longer needed.

= [IMPORTANT]

# The destroy action permanently removes resources without prompting for confirmation, unlike the CLI version of this command. Make sure you have backups of any important data before destroying stacks.

== [source,typescript,subs=“verbatim,attributes”]

import { StackSelectionStrategy } from ‘@aws-cdk/toolkit-lib’;

// Remove specific stacks and their resources await toolkit.destroy(cloudAssemblySource, { stacks: { strategy: StackSelectionStrategy.PATTERN\_MUST\_MATCH, patterns: [“dev-stack”], // Only destroy stacks matching this pattern } }); —