include::attributes.txt[]

// Attributes

[.topic] [#toolkit-library-configure-messages] = Configuring CDK Toolkit messages and interactions :info\_titleabbrev: Configuring messages and interactions :keywords: CDK Toolkit Library, Programmatic access, {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), deploy, {aws} CloudFormation, Infrastructure as Code, io host, IIoHost, messages, interactions

== [abstract]

## The {aws} CDK Toolkit Library provides the IIoHost interface to customize how messages and interactions are handled during CDK operations, enabling you to control the display of deployment progress, error messages, and user prompts to better integrate with your application’s user experience.

// Content start

The {aws} CDK Toolkit Library provides the +link:https://docs.aws.amazon.com/cdk/api/toolkit-lib/Package/toolkit-lib/Interface/IIoHost/[IIoHost]+ interface to customize how messages and interactions are handled during CDK operations, enabling you to control the display of deployment progress, error messages, and user prompts to better integrate with your application’s user experience.

Before performing operations like deployment or synthesis, you need to understand how the CDK Toolkit communicates with users. The IIoHost interface serves as the communication channel between the CDK Toolkit and your application, handling both outgoing messages and incoming user responses.

When the CDK Toolkit runs operations, it communicates through two primary mechanisms:

* *Messages*: Informational outputs that notify you about operation progress (like “Starting deployment” or “Resource created”).
* *Requests*: Decision points that require input or confirmation (like “Do you want to deploy these changes?”), giving you an opportunity to provide information that wasn’t known to be needed beforehand.

[#toolkit-library-configure-messages-iiohost] == Using the IIoHost interface

The IIoHost interface consists of two primary methods:

. notify: Handles one-way informational messages. . requestResponse: Handles interactive requests that require a response.

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

import { IoMessage, IoRequest } from ‘@aws-cdk/toolkit-lib’;

interface IIoHost { // Handle informational messages notify(message: IoMessage): Promise++++++;++++++

// Handle requests that need responses requestResponse(request: IoRequest): Promise++++++; } —-++++++

[#toolkit-library-configure-messages-levels] == Message levels and request types

The CDK Toolkit generates several types of messages and requests:

=== Message levels

* *Debug*: Detailed messages for troubleshooting.
* *Error*: Error messages that may affect operation.
* *Info*: General informational messages.
* *Result* - Primary message of an operation.
* *Trace*: Very detailed execution flow information.
* WARNING: Warning messages that don’t prevent operation.

For a complete list, see link:https://docs.aws.amazon.com/cdk/api/toolkit-lib/message-registry/[IoMessages Registry] in the *{aws} CDK Toolkit Library API Reference*.

=== Request types

The CDK Toolkit sends requests when it needs input or confirmation from the user. These are special messages that allow for a response. If no response is provided, the Toolkit will use a default response when available.

[#toolkit-library-configure-messages-basic] == Basic IIoHost implementation

Here’s a simple example of implementing a custom io host:

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

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

// Create a toolkit with custom message handling const toolkit = new Toolkit({ ioHost: { // Implementing the IIoHost interface // Handle informational messages notify: async function (msg) { // Example: Handle different message levels appropriately switch (msg.level) { case ‘error’: console.error([${msg.time}] ERROR: ${msg.message}); break; case ‘warning’: console.warn([${msg.time}] WARNING: ${msg.message}); break; case ‘info’: console.info([${msg.time}] INFO: ${msg.message}); break; case ‘debug’: console.debug([${msg.time}] DEBUG: ${msg.message}); break; case ‘trace’: console.debug([${msg.time}] TRACE: ${msg.message}); break; default: console.log([${msg.time}] ${msg.level}: ${msg.message}); } },

// Handle requests that need responses requestResponse: async function (msg) { // Example: Log the request and use default response console.log(Request: ${msg.message}, using default: ${msg.defaultResponse}); return msg.defaultResponse;

// Or implement custom logic to provide responses // if (msg.type === ‘deploy’) { // return promptUserForDeployment(msg); // } } } as IIoHost // Explicitly cast to IIoHost interface }); —-

The CDK Toolkit awaits the completion of each call, allowing you to perform asynchronous operations like HTTP requests or user prompts when handling messages.

[#toolkit-library-configure-messages-iiohost-default] == Default IIoHost behavior

If you don’t provide a custom io host, the CDK Toolkit Library uses a default non-interactive implementation:

* Console output for messages (using appropriate colors for different message types).
* Fully non-interactive with no prompts for user input.
* Automatically uses default responses where possible (equivalent to answering “yes” to prompts).
* Fails when input is required but no default response is available.

This default behavior is suitable for unattended operations, but not for interactive command-line applications. For command-line applications that require user interaction, you’ll need to implement a custom io host. Custom implementations are also useful for integrating with logging systems, UIs, or other specialized environments.

[#toolkit-library-configure-messages-advanced] == Advanced io host implementation

For more complex scenarios, we recommend extending the NonInteractiveIoHost class as a starting point. This approach allows you to leverage the existing non-interactive implementation while customizing only the specific behaviors you need to change.

Here’s an example of a custom io host that extends the base implementation:

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

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

class MyCustomIoHost extends NonInteractiveIoHost { // Override only the methods you need to customize

// Example: Custom implementation for notify public async notify(msg: IoMessage++++++): Promise++++++{ // Add custom notification handling logic if (msg.level === ‘error’) { console.error(ERROR: ${msg.message}); // Optionally log to a service or notify a monitoring system await this.logToMonitoringService(msg); } else { await super.notify(msg); } }++++++++++++

// Example: Custom implementation for requestResponse public async requestResponse<T, U>(request: IoRequest<T, U>): Promise++++++{ // Implement custom request handling console.log(Received request: ${request.message}); return request.defaultResponse; }++++++

private async logToMonitoringService(msg: IoMessage++++++): Promise++++++{ // Implementation for monitoring service integration console.log(Logging to monitoring service: ${msg.level} - ${msg.message}); } } —-++++++++++++

This approach is more maintainable than implementing the entire IIoHost interface from scratch, as you only need to override the specific methods that require custom behavior.

[#toolkit-library-configure-messages-integration] == Integrating with different environments

=== Web application integration

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

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

// Example for integrating with a web application const toolkit = new Toolkit({ ioHost: { notify: async function (msg) { // Send message to frontend via WebSocket webSocketServer.send(JSON.stringify({ type: ‘cdk-notification’, messageLevel: msg.level, message: msg.message, time: msg.time })); },

requestResponse: async function (msg) { // Create a promise that will be resolved when the user responds return new Promise((resolve) => { const requestId = generateUniqueId();

// Store the resolver function  
 pendingRequests[requestId] = resolve;  
  
 // Send request to frontend  
 webSocketServer.send(JSON.stringify({  
 type: 'cdk-request',  
 requestId: requestId,  
 requestType: msg.type,  
 message: msg.message,  
 defaultResponse: msg.defaultResponse  
 }));  
  
 // Frontend would call an API endpoint with the response,  
 // which would then call pendingRequests[requestId](response)

}); } } as IIoHost // Explicitly cast to IIoHost interface }); —-

=== CI/CD environment integration

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

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

// Example for CI/CD environments (non-interactive) const toolkit = new Toolkit({ ioHost: { notify: async function (msg) { // Log all messages with appropriate level switch (msg.level) { case ‘error’: console.error(msg.message); break; case ‘warning’: console.warn(msg.message); break; default: console.log(msg.message); } },

requestResponse: async function (msg) { // In CI/CD, always use default responses or predefined answers console.log(Auto-responding to request: ${msg.message} with ${msg.defaultResponse}); return msg.defaultResponse; } } as IIoHost // Explicitly cast to IIoHost interface }); —-

[#toolkit-library-configure-messages-best-practices] == Best practices for io host implementation

When implementing a custom io host, consider these best practices:

* *Error handling*: Implement robust error handling in your io host methods to prevent failures from affecting CDK operations.
* *Timeouts*: Consider implementing timeouts for user interactions to prevent indefinite waiting.
* *Logging*: Store important messages in logs for troubleshooting, especially in non-interactive environments.
* *Default responses*: Provide sensible default responses for automated environments where user interaction isn’t possible.
* *Progress indication*: For long-running operations, provide clear progress indicators to improve user experience.

The following example demonstrates these best practices by implementing a custom io host with error handling, timeouts for user interactions, and proper logging. This implementation is suitable for interactive applications where you need to balance user responsiveness with reliable operation:

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

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

// Example with error handling and timeouts const toolkit = new Toolkit({ ioHost: { notify: async function (msg) { try { console.log([${msg.time}] ${msg.level}: ${msg.message}); // Additional logging or UI updates } catch (error) { // Ensure errors in notification handling don’t break the CDK operation console.error(“Error handling notification:”, error); } },

requestResponse: async function (msg) { try { // Implement timeout for user response const response = await Promise.race([ getUserResponse(msg), new Promise(resolve => setTimeout(() => resolve(msg.defaultResponse), 60000)) ]); return response; } catch (error) { console.error(“Error handling request:”, error); return msg.defaultResponse; } } } as IIoHost // Explicitly cast to IIoHost interface }); —-

[#toolkit-library-configure-messages-troubleshooting] == Troubleshooting

Considerations when implementing a custom io host:

* *Unhandled promise rejections*: Ensure all async operations are properly handled with try/catch blocks.
* *Infinite waiting*: Implement timeouts for all user interactions to prevent the application from hanging.
* *Missing message levels*: Be prepared to handle new message levels that might be added in future CDK versions.
* *Inconsistent responses*: Ensure your requestResponse implementation returns values in the expected format.

The following example demonstrates a robust approach to handling message levels, including graceful handling of unknown message types that might be introduced in future CDK versions. This implementation ensures your io host remains compatible with CDK updates while maintaining proper logging:

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

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

// Example of handling unknown message levels const toolkit = new Toolkit({ ioHost: { notify: async function (msg) { // Handle known levels const knownLevels = [‘info’, ‘warning’, ‘error’, ‘debug’, ‘trace’, ‘status’];

…. if (knownLevels.includes(msg.level)) { // Handle known level handleKnownMessageLevel(msg); } else { // Handle unknown level as info console.log(Unknown message level "${msg.level}": ${msg.message}); } },

requestResponse: async function (msg) { // Default implementation return msg.defaultResponse; } } as IIoHost // Explicitly cast to IIoHost interface }); —- ….