:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#configure-env] = Configure environments to use with the {aws} CDK :info\_titleabbrev: Configure environments :keywords: {aws} CDK, {aws} environments, environments, {aws} account, {aws} Region, env

== [abstract]

## You can configure {aws} environments in multiple ways to use with the {aws} Cloud Development Kit ({aws} CDK). The best method of managing {aws} environments will vary, based on your specific needs.

// Content start

You can configure {aws} environments in multiple ways to use with the {aws} Cloud Development Kit ({aws} CDK). The best method of managing {aws} environments will vary, based on your specific needs.

Each CDK stack in your application must eventually be associated with an environment to determine where the stack gets deployed to.

For an introduction to {aws} environments, see xref:environments[Environments for the {aws} CDK].

[#configure-env-where] == Where you can specify environments from

You can specify environments in credentials and configuration files, or by using the +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html#env[env]+ property of the Stack construct from the {aws} Construct Library.

[#configure-env-where-files] === Credentials and configuration files

You can use the {aws} Command Line Interface ({aws} CLI) to create credentials and config files that store, organize, and manage your {aws} environment information. To learn more about these files, see https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html[Configuration and credential file settings] in the *{aws} Command Line Interface User Guide*.

Values stored in these files are organized by *profiles*. How you name your profiles and the key-value pairs in these files will vary based on your method of configuring programmatic access. To learn more about the different methods, see xref:configure-access[Configure security credentials for the {aws} CDK CLI].

In general, the {aws} CDK resolves {aws} account information from your credentials file and {aws} Region information from your config file.

Once you have your credentials and config files configured, you can specify the environment to use with the {aws} CDK CLI and through environment variables.

[#configure-env-where-env] === env property of the Stack construct

You can specify the environment for each stack by using the +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html#env[env]+ property of the Stack construct. This property defines an account and Region to use. You can pass hard-coded values to this property or pass environment variables that are offered by the CDK.

To pass environment variables, use the AWS\_DEFAULT\_ACCOUNT and AWS\_DEFAULT\_REGION environment variables. These environment variables can pass values from your credentials and config files. You can also use logic within your CDK code to determine the values of these environment variables.

[#configure-env-precedence] == Environment precedence with the {aws} CDK

If you use multiple methods of specifying environments, the {aws} CDK adheres to the following precedence:

. Hard-coded values specified with the env property of the Stack construct. . AWS\_DEFAULT\_ACCOUNT and AWS\_DEFAULT\_REGION environment variables specified with the env property of the Stack construct. . Environment information associated with the profile from your credentials and config files and passed to the CDK CLI using the --profile option. . The default profile from your credentials and config files.

[#configure-env-when] == When to specify environments

When you develop with the CDK, you start by defining CDK stacks, which contain constructs that represent {aws} resources. Next, you synthesize each CDK stack into an {aws} CloudFormation template. You then deploy the CloudFormation template to your environment. How you specify environments determines when your environment information gets applied and can affect CDK behavior and outcomes.

[#configure-env-when-synth] === Specify environments at template synthesis

When you specify environment information using the env property of the Stack construct, your environment information is applied at template synthesis. Running cdk synth or cdk deploy produces an environment-specific CloudFormation template.

If you use environment variables within the env property, you must use the --profile option with CDK CLI commands to pass in the profile containing your environment information from your credentials and configuration files. This information will then be applied at template synthesis to produce an environment-specific template.

Environment information within the CloudFormation template takes precedence over other methods. For example, if you provide a different environment with cdk deploy --profile <profile>, the profile will be ignored.

When you provide environment information in this way, you can use environment-dependent code and logic within your CDK app. This also means that the synthesized template could be different, based on the machine, user, or session that it’s synthesized under. This approach is often acceptable or desirable during development, but is not recommended for production use.

[#configure-env-when-deploy,] === Specify environments at stack deployment

If you don’t specify an environment using the env property of the Stack construct, the CDK CLI will produce an environment-agnostic CloudFormation template at synthesis. You can then specify the environment to deploy to by using cdk deploy --profile <profile>.

If you don’t specify a profile when deploying an environment-agnostic template, the CDK CLI will attempt to use environment values from the default profile of your credentials and config files at deployment.

If environment information is not available at deployment, {aws} CloudFormation will attempt to resolve environment information at deployment through environment-related attributes such as stack.account, stack.region, and stack.availabilityZones.

For environment-agnostic stacks, constructs within the stack cannot use environment information and you cannot use logic that requires environment information. For example, you cannot write code like if (stack.region ==== 'us-east-1') or use construct methods that require environment information such as +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ec2.Vpc.html#static-fromwbrlookupscope-id-options[Vpc.fromLookup]+. To use these features, you must specify an environment with the env property.

For environment-agnostic stacks, any construct that uses Availability Zones will see two Availability Zones, allowing the stack to be deployed to any Region.

[#configure-env-how] == How to specify environments with the {aws} CDK

[#configure-env-how-hard-coded] === Specify hard-coded environments for each stack

Use the env property of the Stack construct to specify {aws} environment values for your stack. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const envEU = { account: ‘2383838383’, region: ‘eu-west-1’ }; const envUSA = { account: ‘8373873873’, region: ‘us-west-2’ };

new MyFirstStack(app, ‘first-stack-us’, { env: envUSA }); new MyFirstStack(app, ‘first-stack-eu’, { env: envEU }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const envEU = { account: ‘2383838383’, region: ‘eu-west-1’ }; const envUSA = { account: ‘8373873873’, region: ‘us-west-2’ };

new MyFirstStack(app, ‘first-stack-us’, { env: envUSA }); new MyFirstStack(app, ‘first-stack-eu’, { env: envEU }); —

Python:: + [source,python,subs=“verbatim,attributes”] — env\_EU = cdk.Environment(account=“8373873873”, region=“eu-west-1”) env\_USA = cdk.Environment(account=“2383838383”, region=“us-west-2”)

MyFirstStack(app, “first-stack-us”, env=env\_USA) MyFirstStack(app, “first-stack-eu”, env=env\_EU) —

Java:: + [source,java,subs=“verbatim,attributes”] — public class MyApp {

…. // Helper method to build an environment static Environment makeEnv(String account, String region) { return Environment.builder() .account(account) .region(region) .build(); }

public static void main(final String argv[]) { App app = new App();

Environment envEU = makeEnv("8373873873", "eu-west-1");  
Environment envUSA = makeEnv("2383838383", "us-west-2");  
  
new MyFirstStack(app, "first-stack-us", StackProps.builder()  
 .env(envUSA).build());  
new MyFirstStack(app, "first-stack-eu", StackProps.builder()  
 .env(envEU).build());  
  
app.synth();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — Amazon.CDK.Environment makeEnv(string account, string region) { return new Amazon.CDK.Environment { Account = account, Region = region }; }

var envEU = makeEnv(account: “8373873873”, region: “eu-west-1”); var envUSA = makeEnv(account: “2383838383”, region: “us-west-2”);

new MyFirstStack(app, “first-stack-us”, new StackProps { Env=envUSA }); new MyFirstStack(app, “first-stack-eu”, new StackProps { Env=envEU }); —

Go:: + [source,go,subs=“verbatim,attributes”] — env\_EU := awscdk.Environment{ Account: jsii.String(“8373873873”), Region: jsii.String(“eu-west-1”), }

env\_USA := awscdk.Environment{ Account: jsii.String(“2383838383”), Region: jsii.String(“us-west-2”), }

MyFirstStack(app, “first-stack-us”, &awscdk.StackProps{ Env: &env\_USA, })

MyFirstStack(app, “first-stack-eu”, &awscdk.StackProps{ Env: &env\_EU, }) — ====

We recommend this approach for production environments. By explicitly specifying the environment in this way, you can ensure that the stack is always deployed to the specific environment.

[#configure-env-how-env] === Specify environments using environment variables

The {aws} CDK provides two environment variables that you can use within your CDK code: CDK\_DEFAULT\_ACCOUNT and CDK\_DEFAULT\_REGION. When you use these environment variables within the env property of your stack instance, you can pass environment information from your credentials and configuration files using the CDK CLI --profile option.

The following is an example of how to specify these environment variables:

==== [role=“tablist”] TypeScript:: Access environment variables via Node’s process object. + [NOTE] – You need the DefinitelyTyped module to use process in TypeScript. cdk init installs this module for you. However, you should install this module manually if you are working with a project created before it was added, or if you didn’t set up your project using cdk init.

== [source,none,subs=“verbatim,attributes”]

## npm install @types/node

|  |
| --- |
| + |
| == [source,javascript,subs=“verbatim,attributes”] |
| new MyDevStack(app, ‘dev’, { env: { account: process.env.CDK\_DEFAULT\_ACCOUNT, region: process.env.CDK\_DEFAULT\_REGION }}); |

JavaScript:: Access environment variables via Node’s process object. + [source,javascript,subs=“verbatim,attributes”] — new MyDevStack(app, ‘dev’, { env: { account: process.env.CDK\_DEFAULT\_ACCOUNT, region: process.env.CDK\_DEFAULT\_REGION }}); —

Python:: Use the os module’s environ dictionary to access environment variables. + [source,python,subs=“verbatim,attributes”] — import os MyDevStack(app, “dev”, env=cdk.Environment( account=os.environ[“CDK\_DEFAULT\_ACCOUNT”], region=os.environ[“CDK\_DEFAULT\_REGION”])) —

Java:: Use System.getenv() to get the value of an environment variable. + [source,java,subs=“verbatim,attributes”] — public class MyApp {

…. // Helper method to build an environment static Environment makeEnv(String account, String region) { account = (account == null) ? System.getenv(“CDK\_DEFAULT\_ACCOUNT”) : account; region = (region == null) ? System.getenv(“CDK\_DEFAULT\_REGION”) : region;

return Environment.builder()  
 .account(account)  
 .region(region)  
 .build();

}

public static void main(final String argv[]) { App app = new App();

Environment envEU = makeEnv(null, null);  
Environment envUSA = makeEnv(null, null);  
  
new MyDevStack(app, "first-stack-us", StackProps.builder()  
 .env(envUSA).build());  
new MyDevStack(app, "first-stack-eu", StackProps.builder()  
 .env(envEU).build());  
  
app.synth();

} } —- ….

C#:: Use System.Environment.GetEnvironmentVariable() to get the value of an environment variable. + [source,csharp,subs=“verbatim,attributes”] — Amazon.CDK.Environment makeEnv(string account=null, string region=null) { return new Amazon.CDK.Environment { Account = account ?? System.Environment.GetEnvironmentVariable(“CDK\_DEFAULT\_ACCOUNT”), Region = region ?? System.Environment.GetEnvironmentVariable(“CDK\_DEFAULT\_REGION”) }; }

== new MyDevStack(app, “dev”, new StackProps { Env = makeEnv() });

Go:: + [source,go,subs=“verbatim,attributes”] — import “os”

MyDevStack(app, “dev”, &awscdk.StackProps{ Env: &awscdk.Environment{ Account: jsii.String(os.Getenv(“CDK\_DEFAULT\_ACCOUNT”)), Region: jsii.String(os.Getenv(“CDK\_DEFAULT\_REGION”)), }, }) — ====

By specifying environments using environment variables, you can have the same CDK stack synthesize to {aws} CloudFormation templates for different environments. This means that you can deploy the same CDK stack to different {aws} environments without having to modify your CDK code. You only have to specify the profile to use when running cdk synth.

This approach is great for development environments when deploying the same stack to different environments. However, we do not recommend this approach for production environments since the same CDK code can synthesize different templates, depending on the machine, user, or session that it’s synthesized under.

[#configure-env-how-files] === Specify environments from your credentials and configuration files with the CDK CLI

When deploying an environment-agnostic template, use the --profile option with any CDK CLI command to specify the profile to use. The following is an example that deploys a CDK stack named myStack using the prod profile that is defined in the credentials and config files:

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

$ cdk deploy ++++++–profile ++++++—-++++++++++++

For more information on the --profile option, along with other CDK CLI commands and options, see xref:ref-cli-cmd[{aws} CDK CLI command reference].

[#configure-env-considerations] == Considerations when configuring environments with the {aws} CDK

Services that you define by using constructs within your stacks must support the Region that you are deploying to. For a list of supported {aws} services per region, see https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/[{aws} Services by Region].

You must have valid {aws} Identity and Access Management (IAM) credentials to perform stack deployments with the {aws} CDK into your specified environments.

[#configure-env-examples] == Examples

[#configure-env-examples-agnostic] === Synthesize an environment–agnostic CloudFormation template from a CDK stack

In this example, we create an environment-agnostic CloudFormation template from our CDK stack. We can then deploy this template to any environment.

The following is our example CDK stack. This stack defines an Amazon S3 bucket and a CloudFormation stack output for the bucket’s Region. For this example, env is not defined:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — export class CdkAppStack extends cdk.Stack { constructor(scope: Construct, id: string, props?: cdk.StackProps) { super(scope, id, props);

…. // Create the S3 bucket const bucket = new s3.Bucket(this, ‘amzn-s3-demo-bucket’, { removalPolicy: cdk.RemovalPolicy.DESTROY, });

// Create an output for the bucket’s Region new cdk.CfnOutput(this, ‘BucketRegion’, { value: bucket.env.region, }); } } —- ….

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — class CdkAppStack extends cdk.Stack { constructor(scope, id, props) { super(scope, id, props);

…. // Create the S3 bucket const bucket = new s3.Bucket(this, ‘amzn-s3-demo-bucket’, { removalPolicy: cdk.RemovalPolicy.DESTROY, });

// Create an output for the bucket’s Region new cdk.CfnOutput(this, ‘BucketRegion’, { value: bucket.env.region, }); } } —- ….

Python:: + [source,python,subs=“verbatim,attributes”] — class CdkAppStack(cdk.Stack):

…. def **init**(self, scope: cdk.Construct, id: str, \*\*kwargs) -> None: super().\_\_init\_\_(scope, id, \*\*kwargs)

# Create the S3 bucket  
bucket = s3.Bucket(self, 'amzn-s3-demo-bucket',  
 removal\_policy=cdk.RemovalPolicy.DESTROY  
)  
  
# Create an output for the bucket's Region  
cdk.CfnOutput(self, 'BucketRegion',  
 value=bucket.env.region  
) ----

….

Java:: + [source,java,subs=“verbatim,attributes”] — public class CdkAppStack extends Stack {

…. public CdkAppStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props);

// Create the S3 bucket  
Bucket bucket = Bucket.Builder.create(this, "amzn-s3-demo-bucket")  
 .removalPolicy(RemovalPolicy.DESTROY)  
 .build();  
  
// Create an output for the bucket's Region  
CfnOutput.Builder.create(this, "BucketRegion")  
 .value(this.getRegion())  
 .build();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — namespace MyCdkApp { public class CdkAppStack : Stack { public CdkAppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Create the S3 bucket var bucket = new Bucket(this, “amzn-s3-demo-bucket”, new BucketProps { RemovalPolicy = RemovalPolicy.DESTROY });

// Create an output for the bucket's Region  
 new CfnOutput(this, "BucketRegion", new CfnOutputProps  
 {  
 Value = this.Region  
 });  
 }

} } —-

Go:: + [source,go,subs=“verbatim,attributes”] — func NewCdkAppStack(scope constructs.Construct, id string, props \*CdkAppStackProps) awscdk.Stack { stack := awscdk.NewStack(scope, &id, &props.StackProps)

…. // Create the S3 bucket bucket := awss3.NewBucket(stack, jsii.String(“amzn-s3-demo-bucket”), &awss3.BucketProps{ RemovalPolicy: awscdk.RemovalPolicy\_DESTROY, })

// Create an output for the bucket’s Region awscdk.NewCfnOutput(stack, jsii.String(“BucketRegion”), &awscdk.CfnOutputProps{ Value: stack.Region(), })

return stack } —- ==== ….

When we run cdk synth, the CDK CLI produces a CloudFormation template with the pseudo parameter +{aws}::Region+ as the output value for the bucket’s Region. This parameter will be resolved at deployment:

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

Outputs: BucketRegion: Value: Ref: {aws}::Region —

To deploy this stack to an environment that is specified in the dev profile of our credentials and configuration files, we run the following:

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

## $ cdk deploy CdkAppStack –profile dev

If we don’t specify a profile, the CDK CLI will attempt to use environment information from the default profile in our credentials and configuration files.

[#configure-env-example-logic] === Use logic to determine environment information at template synthesis

In this example, we configure the env property of our stack instance to use a valid expression. We specify two additional environment variables, CDK\_DEPLOY\_ACCOUNT and CDK\_DEPLOY\_REGION. These environment variables can override defaults at synthesis time if they exist:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new MyDevStack(app, ‘dev’, { env: { account: process.env.CDK\_DEPLOY\_ACCOUNT || process.env.CDK\_DEFAULT\_ACCOUNT, region: process.env.CDK\_DEPLOY\_REGION || process.env.CDK\_DEFAULT\_REGION }}); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new MyDevStack(app, ‘dev’, { env: { account: process.env.CDK\_DEPLOY\_ACCOUNT || process.env.CDK\_DEFAULT\_ACCOUNT, region: process.env.CDK\_DEPLOY\_REGION || process.env.CDK\_DEFAULT\_REGION }}); —

Python:: + [source,python,subs=“verbatim,attributes”] — MyDevStack(app, “dev”, env=cdk.Environment( account=os.environ.get(“CDK\_DEPLOY\_ACCOUNT”, os.environ[“CDK\_DEFAULT\_ACCOUNT”]), region=os.environ.get(“CDK\_DEPLOY\_REGION”, os.environ[“CDK\_DEFAULT\_REGION”]) ) ) —

Java:: + [source,java,subs=“verbatim,attributes”] — public class MyApp {

…. // Helper method to build an environment static Environment makeEnv(String account, String region) { account = (account == null) ? System.getenv(“CDK\_DEPLOY\_ACCOUNT”) : account; region = (region == null) ? System.getenv(“CDK\_DEPLOY\_REGION”) : region; account = (account == null) ? System.getenv(“CDK\_DEFAULT\_ACCOUNT”) : account; region = (region == null) ? System.getenv(“CDK\_DEFAULT\_REGION”) : region;

return Environment.builder()  
 .account(account)  
 .region(region)  
 .build();

}

public static void main(final String argv[]) { App app = new App();

Environment envEU = makeEnv(null, null);  
Environment envUSA = makeEnv(null, null);  
  
new MyDevStack(app, "first-stack-us", StackProps.builder()  
 .env(envUSA).build());  
new MyDevStack(app, "first-stack-eu", StackProps.builder()  
 .env(envEU).build());  
  
app.synth();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — Amazon.CDK.Environment makeEnv(string account=null, string region=null) { return new Amazon.CDK.Environment { Account = account ?? System.Environment.GetEnvironmentVariable(“CDK\_DEPLOY\_ACCOUNT”) ?? System.Environment.GetEnvironmentVariable(“CDK\_DEFAULT\_ACCOUNT”), Region = region ?? System.Environment.GetEnvironmentVariable(“CDK\_DEPLOY\_REGION”) ?? System.Environment.GetEnvironmentVariable(“CDK\_DEFAULT\_REGION”) }; }

== new MyDevStack(app, “dev”, new StackProps { Env = makeEnv() });

Go:: + [source,go,subs=“verbatim,attributes”] — var account, region string var b bool

if account, b = os.LookupEnv(“CDK\_DEPLOY\_ACCOUNT”); !b || len(account) == 0 { account = os.Getenv(“CDK\_DEFAULT\_ACCOUNT”) } if region, b = os.LookupEnv(“CDK\_DEPLOY\_REGION”); !b || len(region) == 0 { region = os.Getenv(“CDK\_DEFAULT\_REGION”) }

MyDevStack(app, “dev”, &awscdk.StackProps{ Env: &awscdk.Environment{ Account: &account, Region: &region, }, }) — ====

With our stack's environment declared this way, we can then write a short script or batch file and set variables from command line arguments, then call ```cdk deploy```. The following is an example. Any arguments beyond the first two are passed through tocdk deploy` to specify command line options or arguments:

==== [role=“tablist”] macOS/Linux:: + [source,bash,subs=“verbatim,attributes”] — #!/usr/bin/env bash if [[ $# -ge 2 ]]; then export CDK\_DEPLOY\_ACCOUNT=$1 export CDK\_DEPLOY\_REGION=$2 shift; shift npx cdk deploy "$@” exit $? else echo 1>&2 “Provide account and region as first two args.” echo 1>&2 “Additional args are passed through to cdk deploy.” exit 1 fi — + Save the script as cdk-deploy-to.sh, then execute chmod +x cdk-deploy-to.sh to make it executable.

Windows:: + [source,bat,subs=“verbatim,attributes”] — @findstr /B /V @ %~dpnx0 > %~dpn0.ps1 && powershell -ExecutionPolicy Bypass %~dpn0.ps1 %\* @exit /B %ERRORLEVEL% if ($args.length -ge 2) { $env:CDK\_DEPLOY\_ACCOUNT, $args = $args $env:CDK\_DEPLOY\_REGION, $args = $args npx cdk deploy $args exit $lastExitCode } else { [console]::error.writeline(“Provide account and region as first two args.”) [console]::error.writeline(“Additional args are passed through to cdk deploy.”) exit 1 } — + The Windows version of the script uses PowerShell to provide the same functionality as the macOS/Linux version. It also contains instructions to allow it to be run as a batch file so it can be easily invoked from a command line. It should be saved as cdk-deploy-to.bat. The file cdk-deploy-to.ps1 will be created when the batch file is invoked.

====

We can then write additional scripts that use the cdk-deploy-to script to deploy to specific environments. The following is an example:

==== [role=“tablist”] macOS/Linux:: + [source,bash,subs=“verbatim,attributes”] — #!/usr/bin/env bash

= cdk-deploy-to-test.sh

## ./cdk-deploy-to.sh 123457689 us-east-1 “$@”

Windows:: + [source,bat,subs=“verbatim,attributes”] — @echo off rem cdk-deploy-to-test.bat cdk-deploy-to 135792469 us-east-1 %\* — ====

The following is an example that uses the cdk-deploy-to script to deploy to multiple environments. If the first deployment fails, the process stops:

==== [role=“tablist”] macOS/Linux:: + [source,bash,subs=“verbatim,attributes”] — #!/usr/bin/env bash

= cdk-deploy-to-prod.sh

./cdk-deploy-to.sh 135792468 us-west-1 “$@" || exit ./cdk-deploy-to.sh 246813579 eu-west-1 "$@” —

Windows:: + [source,bat,subs=“verbatim,attributes”] — @echo off rem cdk-deploy-to-prod.bat cdk-deploy-to 135792469 us-west-1 %\* || exit /B cdk-deploy-to 245813579 eu-west-1 %\* — ====