Tokens

PDF (awscdk.pdf#tokens)
Kindle (https://www.amazon.com/dp/B07S97HM34)
RSS (awscdk.rss)

Tokens represent values that can only be resolved at a later time in the lifecycle of an app (see App lifecycle (./apps.html#lifecycle)). For example, the name of an Amazon S3 bucket that you define in your AWS CDK app is only allocated when the AWS CloudFormation template is synthesized. If you print the `bucket.bucketName` attribute, which is a string, you see it contains something like the following.

```
${TOKEN[Bucket.Name.1234]}
```

This is how the AWS CDK encodes a token whose value is not yet known at construction time, but will become available later. The AWS CDK calls these placeholders *tokens*. In this case, it's a token encoded as a string.

You can pass this string around as if it was the name of the bucket, such as in the following example, where the bucket name is specified as an environment variable to an AWS Lambda function.

TypeScript (#typescript) JavaScript (#javascript) Python (#python) Java (#java)
C# (#csharp)

```
const bucket = new s3.Bucket(this, 'MyBucket');

const fn = new lambda.Function(stack, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName,
  }
});
```

When the AWS CloudFormation template is finally synthesized, the token is rendered as the AWS CloudFormation intrinsic `{ "Ref": "MyBucket" }`. At deployment time, AWS CloudFormation replaces this intrinsic with the actual name of the bucket that was created.

Tokens and token encodings

Tokens are objects that implement the IResolvable (https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_core.IResolvable.html) interface, which contains a single `resolve` method. The AWS CDK calls this method during synthesis to produce the final value for the AWS CloudFormation template. Tokens participate in the synthesis process to produce arbitrary values of any type.

Note

You'll hardly ever work directly with the `IResolvable` interface. You will most likely only see string-encoded versions of tokens.

Other functions typically only accept arguments of basic types, such as `string` or `number`. To use tokens in these cases, you can encode them into one of three types using static methods on the core.Token (https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_core.Token.html) class.

- `Token.asString` (https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_core.Token.html#static-as-stringvalue-options) to generate a string encoding (or call `.toString()` on the token object)
- `Token.asList` (https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_core.Token.html#static-as-listvalue-options) to generate a list encoding
- `Token.asNumber` (https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_core.Token.html#static-as-numbervalue) to generate a numeric encoding

These take an arbitrary value, which can be an `IResolvable`, and encode them into a primitive value of the indicated type.

⚠Important

Because any one of the previous types can potentially be an encoded token, be careful when you parse or try to read their contents. For example, if you attempt to parse a string to extract a value from it, and the string is an encoded token, your parsing will fail. Similarly, if you attempt to query the length of an array, or perform math operations with a number, you must first verify that they are not encoded tokens.

To check whether a value has an unresolved token in it, call the `Token.isUnresolved` (Python: `is_unresolved`) method.

The following example validates that a string value, which could be a token, is no more than 10 characters long.

TypeScript (#typescript) JavaScript (#javascript) Python (#python) Java (#java) C# (#csharp)

```
if (!Token.isUnresolved(name) && name.length > 10) {
  throw new Error(`Maximum length for name is 10 characters`);
}
```

If **name** is a token, validation isn't performed, and an error could still occur in a later stage in the lifecycle, such as during deployment.

ℹNote

You can use token encodings to escape the type system. For example, you could string-encode a token that produces a number value at synthesis time. If you use these functions, it's your responsibility to ensure that your template resolves to a usable state after synthesis.

String-encoded tokens

String-encoded tokens look like the following.

```
${TOKEN[Bucket.Name.1234]}
```

They can be passed around like regular strings, and can be concatenated, as shown in the following example.

TypeScript (#typescript) JavaScript (#javascript) Python (#python) Java (#java)

C# (#csharp)

```
const functionName = bucket.bucketName + 'Function';
```

You can also use string interpolation, if your language supports it, as shown in the following example.

TypeScript (#typescript) JavaScript (#javascript) Python (#python) Java (#java)

C# (#csharp)

```
const functionName = `${bucket.bucketName}Function`;
```

Avoid manipulating the string in other ways. For example, taking a substring of a string is likely to break the string token.

List-encoded tokens

List-encoded tokens look like the following

```
["#{TOKEN[Stack.NotificationArns.1234]}"]
```

The only safe thing to do with these lists is pass them directly to other constructs. Tokens in string list form cannot be concatenated, nor can an element be taken from the token. The only safe way to manipulate them is by using AWS CloudFormation intrinsic functions like Fn.select (https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-select.html) .

Number-encoded tokens

Number-encoded tokens are a set of tiny negative floating-point numbers that look like the following.

```
-1.8881545897087626e+289
```

As with list tokens, you cannot modify the number value, as doing so is likely to break the number token. The only allowed operation is to pass the value around to another construct.

Lazy values

In addition to representing deploy-time values, such as AWS CloudFormation parameters (./parameters.html) , Tokens are also commonly used to represent synthesis-time lazy values. These are values for which the final value will be determined before synthesis has completed, just not at the point where the value is constructed.

Use tokens to pass a literal string or number value to another construct, while the actual value at synthesis time may depend on some calculation that has yet to occur.

You can construct tokens representing synth-time lazy values using static methods on the `Lazy` class, such as Lazy.stringValue (https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_core.Lazy.html#static-string-valueproducer-options) (Python: `Lazy.string_value`) and Lazy.numberValue (https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_core.Lazy.html#static-number-valueproducer) (Python: `Lazy.number_value`. These methods accept an object whose `produce` property is a function that accepts a context argument and returns the final value when called.

The following example creates an Auto Scaling group whose capacity is determined after its creation.

TypeScript (#typescript) JavaScript (#javascript) Python (#python) Java (#java)

C# (#csharp)

```
let actualValue: number;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return actualValue;
    }
  })
});

// At some later point
actualValue = 10;
```

Converting to JSON

Sometimes you want to produce a JSON string of arbitrary data, and you may not know whether the data contains tokens. To properly JSON-encode any data structure, regardless of whether it contains tokens, use the method stack.toJsonString (https://docs.aws.amazon.com/cdk/api/latest/docs/@aws-cdk_core.Stack.html#to-json-stringobj-space) , as shown in the following example.

TypeScript (#typescript) JavaScript (#javascript) Python (#python) Java (#java)

C# (#csharp)

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```