Uploading Files to S3

In this lesson, you will learn how to upload files to S3 by modifying the template file.

WE'LL COVER THE FOLLOWING

- Using custom resources to extend CloudFormation
- Redeploying custom resources
- Uploading files without the SAR component

You added a `web-site` directory to your project, with static web assets. Before you can deploy the new version of the application, you need to send those assets to S3 and replace the placeholder for the API URL in the index file.

At the time this was written, CloudFormation didn't have any built-in support for sending web asset files to S3. However, there is a component in the AWS Serverless Application Repository that can handle simple uploads to S3, which is just what you need to deploy everything at the same time. `DeployToS3`, created by Aleksandar Simović, can upload files to S3 and optionally replace patterns in those files with CloudFormation references. That component will make it very easy to coordinate API Gateway updates and static website assets. Import the component into the application template by adding it to the `Resources` section. The code from the following listing will be added and indented so it aligns with existing resources (for example `ConvertFileFunction`).

```
DeployToS3:
  Type: AWS::Serverless::Application
  Properties:
    Location:
      ApplicationId: arn:aws:serverlessrepo:us-east-1:375983427419:applications/deploy-to-s3
      SemanticVersion: 1.0.0
```

Line 136 to Line 141 of code/ch11/template.yaml

CloudFormation and SAM can only package files from a local directory for Lambda functions. `DeployToS3` abuses the Lambda packaging process slightly to trick CloudFormation into collecting files from a local directory with web assets and uploading them to S3.

In order for CloudFormation to package the `web-site` directory files, you need to make it think it's collecting the source code for a Lambda function. For that, you just need to create another `AWS::Serverless:Function` resource and point it to the local directory. That function, of course, won't be able to run, because it does not contain executable code or a Lambda handler. Also, the files will be, in that case, deployed to a Lambda container, not to S3. That's where the `DeployToS3` component comes in. It contains a Lambda layer with a Python function handler. When you attach the layer to the fake Lambda function, it will make the function executable. Once the handler from the layer runs, it uploads the current function source code (which will be the contents of the `web-site` directory) to a specified S3 location.

The `Arn` output of the `DeployToS3` component will contain the layer reference, so you'll need to add that to the fake function. You'll also need to ensure that the function can write to the web assets bucket, so that the handler from the layer can upload the files. The block from the following listing is added to the `Resources` section of your template, aligned with the `DeployToS3` resource.

```
SiteSource:
  Type: AWS::Serverless::Function
  Properties:
    Layers:
      - !GetAtt DeployToS3.Outputs.Arn
    CodeUri: web-site/
    AutoPublishAlias: production
    Runtime: python3.6
    Handler: deployer.resource_handler
    Timeout: 600
    Policies:
      - S3FullAccessPolicy:
          BucketName: !Ref WebAssetsS3Bucket
```

Line 142 to Line 154 of code/ch11/template.yaml

The layer works as a Python application, and lines 8 and 9 configure the function so that the layer can take over the execution. This allows you to deploy web assets both without developing any new code and by packaging the Lambda function just with the website assets.

Line 8 from the previous listing is very important. The handler in the `DeployToS3` layer is written in Python and supports Python 3.6 and 3.7 Lambda environments. SAM will try to build all local functions when you run `sam build`, including the fake function, so change this configuration line to match your local Python version. If you don't do that, SAM will complain about a Python binary mismatch.

> At the time this was written, `sam build` for Python applications failed unless the function directory contained a package manifest file, even if it had no dependencies. Create an empty file called `requirements.txt` in the `web-site` directory to trick it into packaging up the assets.

Using custom resources to extend CloudFormation

CloudFormation will now package the `web-site` directory files with your template. It will also create the `SiteSource` Lambda function containing those files. You now just need to make the Lambda function execute during deployments instead of waiting for someone to manually trigger it later. CloudFormation can do that with a feature called *custom resource*.

Custom resources are extensions to CloudFormation, handling application-specific workflows that are not supported by standard components, such as uploading files to S3. When declaring a custom resource, you can tell CloudFormation to execute a Lambda function, which is ideal for the `SiteSource` fake function. In fact, the layer created by `DeployToS3` is intended to work this way, so it supports the custom resource workflow out of the box.

To declare a custom resource, you just add another block to the `Resources` section of the template, using the type `AWS::CloudFormation::CustomResource`. Custom resources require one property, named `ServiceToken`, pointing to a Lambda function that CloudFormation needs to execute. You can add any other properties required for the resource to function.

To run the `SiteSource` function during deployment, you need to create a custom resource with the function ARN as the `ServiceToken`. You will also

need to configure a few other properties required by the `DeployToS3` component:

- `TargetBucket` should point to an S3 bucket for file uploads.
- `Acl` is an optional property defining the access control for uploaded files. You want to serve public files but not let anyone change them, so you'll need to specify `public-read`.
- `CacheControlMaxAge` sets the optional browser caching period in seconds. For static files, you can safely turn on caching.
- `Substitutions` is an optional property containing two fields: `FilePattern` is a regular expression that tells `DeployToS3` which files to search for placeholders and `Values` maps placeholder names to replacements. You changed `index.html` to include a placeholder for the `API_URL`, so you can now define the appropriate substitution.

The block from the following listing is added to the `Resources` section of your template, aligned with the `SiteSource` resource.

```
DeploymentResource:
  Type: AWS::CloudFormation::CustomResource
  Properties:
    ServiceToken: !GetAtt SiteSource.Arn
    Version: !Ref "SiteSource.Version"
    TargetBucket: !Ref WebAssetsS3Bucket
    Substitutions:
      FilePattern: "*.html"
      Values:
        API_URL: !Sub "https://${WebApi}.execute-api.${AWS::Region}.amazonaws.com/${AppStage}
    Acl: 'public-read'
    CacheControlMaxAge: 600
```

Line 155 to Line 166 of code/ch11/template.yaml

Redeploying custom resources

CloudFormation will only update a custom resource if its parameters change, not if the underlying Lambda function changes. This is normally OK, but not in this case. You want to execute the custom resource automatically if any files in the `web-site` directory change, and CloudFormation thinks that those files are the source code for the Lambda function.

To trick CloudFormation into running the `SiteSource` function whenever its files change, you need to add a property to the custom resource that will be different with each function update. That's why the `SiteSource` function from

the previous listing automatically publishes a Lambda alias (line 148 of

`code/ch11/template.yaml`), and why the `DeploymentResource` contains a `Version` parameter pointing to the published Lambda version (line 5 in the code snippet above). This parameter is not strictly required by the `DeployToS3` application, but using a numerically incrementing version ensures that the web files are always consistent with your API.

Notice that `SiteSource.Version` in line 5 is a reference, not an attribute, so you can use `!Ref` to read it. This is a curiosity of SAM version publishing. When you use the `AutoPublishAlias` property of a function resource, SAM automatically stores the resulting version in a new reference, which it names by appending `.Version` to the function name.

You will build, package, and deploy the application again. Remember that you can add `CAPABILITY_AUTO_EXPAND` when deploying, so CloudFormation can process nested applications. Once the application is deployed, you will get the stack outputs to discover the website endpoint URL, and you should open it in your browser.

Environment Variables

Key:	Value:
AWS_ACCESS_KEY_ID	Not Specified...
AWS_SECRET_ACCE...	Not Specified...
BUCKET_NAME	Not Specified...
AWS_REGION	Not Specified...

```
{
  "body": "{\"message\": \"hello world\"}",
  "resource": "/{proxy+}",
  "path": "/path/to/resource",
  "httpMethod": "POST",
  "isBase64Encoded": false,
  "queryStringParameters": {
    "foo": "bar"
  },
  "pathParameters": {
    "proxy": "/path/to/resource"
  },
  "stageVariables": {
    "baz": "qux"
  },
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate, sdch",
```

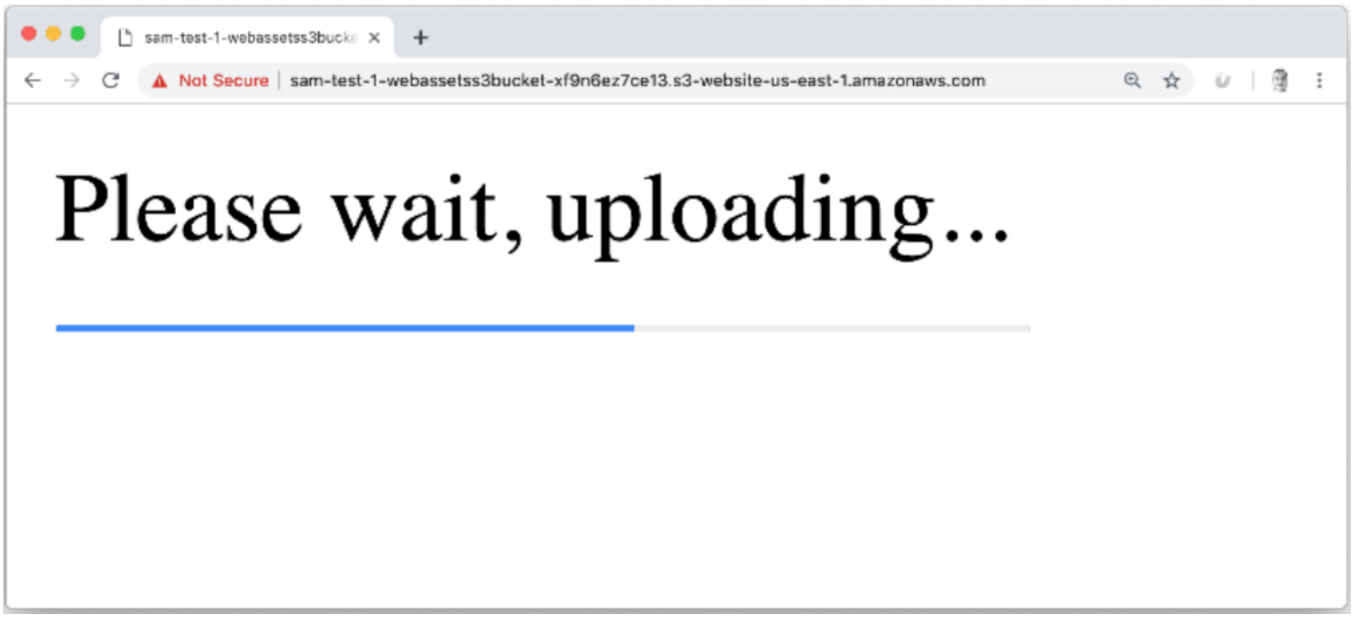
```
    "Accept-Language": "en-US,en;q=0.8",
    "Cache-Control": "max-age=0",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-Country": "US",
    "Host": "1234567890.execute-api.us-east-1.amazonaws.com",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Custom User Agent String",
    "Via": "1.1 08f323deadbeefa7af34d5feb414ce27.cloudfront.net (CloudFront)",
    "X-Amz-Cf-Id": "cDehVQoZnx43VYQb9j2-nvCh-9z396Uhbp027Y2JvkCPNLmGJHqlaA==",
    "X-Forwarded-For": "127.0.0.1, 127.0.0.2",
    "X-Forwarded-Port": "443",
    "X-Forwarded-Proto": "https"
  },
  "requestContext": {
    "accountId": "123456789012",
    "resourceId": "123456",
    "stage": "prod",
    "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
    "requestTime": "09/Apr/2015:12:34:56 +0000",
    "requestTimeEpoch": 1428582896000,
    "identity": {
      "cognitoIdentityPoolId": null,
      "accountId": null,
      "cognitoIdentityId": null,
      "caller": null,
      "accessKey": null,
      "sourceIp": "127.0.0.1",
      "cognitoAuthenticationType": null,
      "cognitoAuthenticationProvider": null,
      "userArn": null,
      "userAgent": "Custom User Agent String",
      "user": null
    },
    "path": "/prod/path/to/resource",
    "resourcePath": "/{proxy+}",
    "httpMethod": "POST",
    "apiId": "1234567890",
    "protocol": "HTTP/1.1"
  }
}
```


Check out the source code of the page and you should see that the placeholder in the index page was replaced with the full URL of the actual API as in the figure given below.

```
<html>
  <body>
    <div step="initial">
      <h1>Select a file</h1>
      <input type="hidden" id="apiurl" value="https://6mb2087x87.execute-api.us-east-1.amazonaws.com/api/" />
      <input type="file" id="picker"/>
    </div>
    <div step="uploading" style="display: none">
      Please wait, uploading...
      <br/>
      <progress id="progressbar" max="100" value="0" />
    </div>
    <div step="converting" style="display: none">
      Please wait, converting your file...
    </div>
    <div step="result" style="display: none">
      <h1>Your thumbnail is ready</h1>
      <a id="resultlink">download it</a>
    </div>
    <div step="error" style="display: none">
      <h1>There was an error creating the thumbnail</h1>
      <p id="errortext"></p>
    </div>
    <script src="user-workflow.js"></script>
  </body>
</html>
```

`DeployToS3` replaces the placeholder with the actual API URL during deployment.

Try uploading a JPG file using the new web page. You should see a progress bar while the browser is transferring your file as shown in the figure below. The client code will also wait until the result is ready to show the download link, making the user experience a lot less error prone.



Moving session workflow out of Lambda functions to the client allows us to reduce costs but also create a nicer experience, such as by displaying progress bars.

Uploading files without the SAR component

The `DeployToS3` AWS Serverless Application repository component can handle basic upload tasks easily. For more complex scenarios, such as individually

controlling caching or response headers for files, you can use the AWS

command line tools and somehow coordinate CloudFormation template deployment with uploading web assets.

The `aws s3` tool has a convenient option for synchronising a local directory with an S3 bucket. Here is the basic form for how to use it:

```
aws s3 sync <LOCAL DIR> s3://<BUCKET>/<PREFIX>
```

To make files publicly accessible, add `--acl public-read`. The command has many more options, such as for adding or removing groups of files, and setting caching headers and similar properties. Check out the S3 `sync` documentation page for more information.

AWS also has a workflow product for deploying front-end applications, called AWS Amplify Console. It might be worth investigating if you want to build and deploy complex front ends to serverless applications.

You now have a nice baseline for an application. In the next chapter, you'll make the API more robust and deal with other tasks, such as testing, that are required before your application is truly ready to accept real users.

Next up you have the 'Interesting Experiments' section!