MozDef Event Framework

Release 0.0.1

Dec 13, 2019

Contents

CHAPTER 1

Overview

1.1 Purpose

It's easiest to describe The MozDef Event Framework as a set of micro-services you can use to integrate event sources with the Mozilla Enterprise Defense platform (MozDef).

1.2 Provides

Many resources are not wholly contained within the datacenter, there are SAAS and IAAS providers which may provide event logging that is external to the network where your SIEM resides. Exposing your security infrastructure doesn't ensure a secure pipeline. In order to solve for this we decided on pulling that information as our method of choice, as opposed to pushing. Utilizing cloud based microservices can reduce exposure, ensure efficiency and scalability, and leads to less maintenance.

We created a framework using AWS Cloudformation and the Serverless framework to build a pipeline that will allow placement of lambda scripts specific for an event source and provide continuous deployment and integration. This takes the guesswork out of having to write serverless and cloudformation code every time you want to deploy a pipeline for a new event source and creates a standard method to be used to obtain those events.

Webhook sources utilize an AWS API gateway that they can post events to. As we scale this out the capabilities will increase. The hope is to have a scalable framework that can be deployed for both REST and webhook based sources.

1.3 Flow

There are three cloudformation templates to choose from in the templates directory:

- Github repo used as source: templates/codepipeline-cf-template-github-source.yml
- CodeCommit repo used as the config source: templates/codepipeline-cf-template-codecommit-source.yml

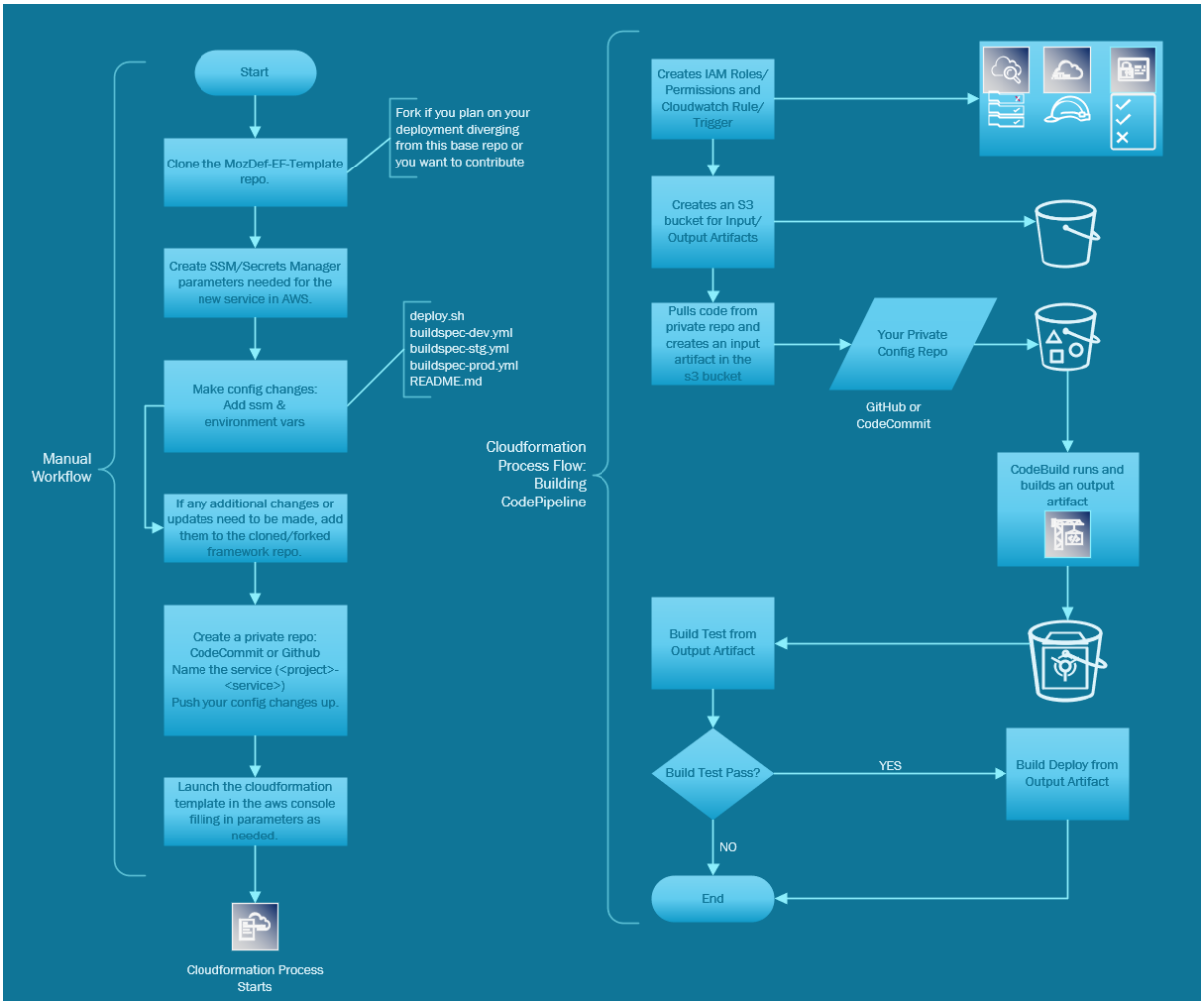
- Github and CodeCommit used as a multiple source that can be merged: templates/multi_source_template/codepipeline-cf-template-with-merge.yml

We've added toggles to view the basic workflow diagrams below, feel free to choose what works best for you.

1.3.1 Workflow Diagrams

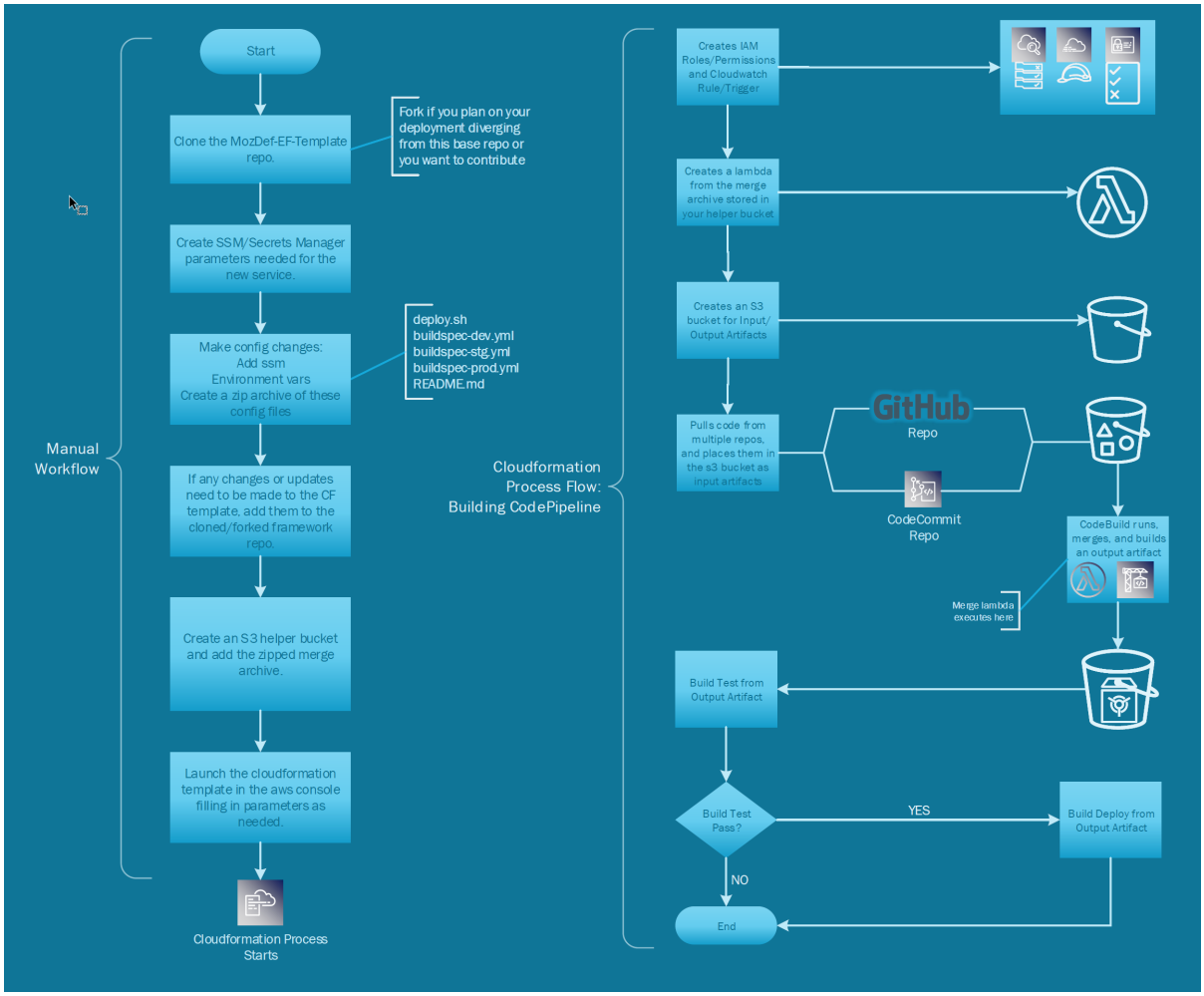
1. Workflow Using a Single Source Repo:

Workflow using a Single Source Repo Cloudformation Template



2. Workflow Using Multiple Source Repos:

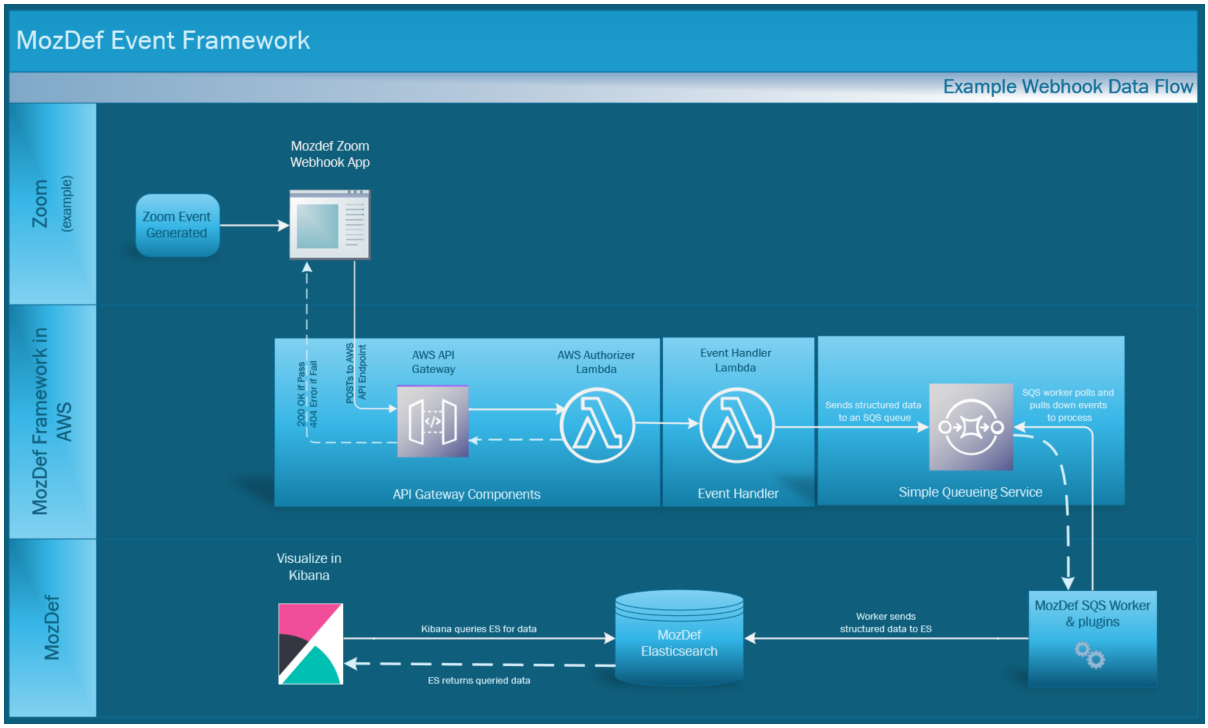
Workflow using the Mutliple Source Repos w/ Merge Cloudformation Template



1.3.2 Dataflow Diagram

Example Data Flow Using Zoom's Webhook:

Example Dataflow using Zoom's Webhook



1.4 Components

The following components make up this framework:

- AWS API Gateway
- AWS Cloudformation
- AWS CloudWatch
- AWS CodePipeline
- AWS CodeCommit
- AWS CodeBuild
- AWS Lambda
- AWS S3
- AWS SQS
- AWS XRay
- GitHub
- Serverless framework

1.5 Status

The MozDef event framework is under development at this time.

1.6 Goals

1.6.1 High level

- Provide a platform for use by security infrastructure engineers to rapidly deploy a pipeline to enable ingestion of events into MozDef.
- Facilitate continuous integration and development.
- Facilitate repeatable, predictable processes for adding new event sources.
- Provide a means with which to reprocess any events that do not meet the requirements you set.

1.6.2 Technical

- Offer micro services that enable rapid consumption of various event sources as needed.
- Scalable, should be able to handle thousands of events per second, provide validation, and a means to reprocess events that fail validation by utilizing the CI/CD pipeline this framework builds.

1.7 Roadmap

1.7.1 Done

- Allows the use of Webhook API connectivity
- Can pull configuration from one or more sources (Github/CodeCommit) during build by selecting the appropriate CF template
- Utilizes SSM and Secrets manager parameters to prevent exposure of secrets through code
- CI/CD pipeline is implemented using AWS Codepipeline

1.7.2 ToDo

- Implement Dead Letter Queue (DLQ) reprocessing functionality
- Implement schema validation on incoming events.
- Implement function library to allow choice between webhook API or REST API connectivity
- Implement monitoring of the entire stack that is created by this framework

1.8 Inspiration

The following resources inspired us and were used to build out this project:

- https://github.com/tooltwist/codepipeline-artifact-munge
- https://github.com/getcft/aws-serverless-code-pipeline-cf-template

CHAPTER 2

Architecture

2.1 Cloudformation

Cloudformation will create the CI/CD components utilizing:

- **CodePipeline**
- **CodeCommit**
- **CodeBuild**
- **S3**
- **CloudWatch**
- **Cloudtrail**
- **GitHub**

There are multiple Cloudformation templates to choose from depending on the management style you prefer. The template can be deployed once for each event source you wish to integrate. Each event source will have it's own configuration repository that the Cloudformation template will pull from to build the framework. This allows us to maintain a generic framework that is event source agnostic and relieved of sensitive data while still maintaining configuration revision history specific to each event source.

2.2 Serverless

The Serverless framework deploys the application components that will handle the event source data. This framework creates the following resources:

- A **custom authorizer** for webhook based APIs that use an Authorization header.
- An **API Gateway** for receiving post events using the authorizer.

- Custom configuration variables that are populated by the buildspec.yml located in each event source configuration repository.[1]
- A **lambda** event handler that will validate to some degree the event to be processed by MozDef. At some point we may extend functionality here to fully process the event by another lambda.
- An **SQS queue** to receive the events that have been handled by our lambda handler.
- An **SQS Dead Letter Queue (DLQ)** to receive events that fail to be handled, so that they can be reprocessed once a fix is introduced. This is future functionality that has not been implemented yet.

[1] This repository can be in **GitHub** or **CodeCommit** depending on the Cloudformation template used.

CHAPTER 3

Preparation

To deploy this framework you will need the following established:

1. **AWS Account:**

   An AWS Account with the ability to use all of the AWS services previously mentioned.

2. **Environment Variables:**

   Decide on the various parameter and environment variable values, as these are referenced throughout the CF templates and serverless framework configuration. These include (example values in parenthesis):

   - Project (e.g., MozDef-EF)
   - Service (e.g., zoom)
   - Environment or Stage (e.g. dev)
   - Stack name (e.g., <project>-<service>, MozDef-EF-zoom)
   - Token Arn (e.g., arn:aws:ssm:us-west-2:<ACCOUNT_ID>:parameter/<project/service/environment/auth_token>, arn:aws:ssm:us-west-2:<ACCOUNT_ID>:parameter/MozDef-EF/zoom/dev/auth_token>)

3. **Event Handler Lambda Code:**

   The event handler lambda code that currently lives in the public github repo under functions/handler.py should be viewed as a template to get you going with your own service. If you are implementing a simple webhook pipeline and the response returns a body and a payload, then it should function just fine. However, if it does not contain either of those parameters, then you will need to modify the handler.py to evaluate the webhook content for what you expect.

4. **SSM/KMS/Secrets Manager:**

   You'll need to design your parameter store pathing, and add your tokens or other data to be used with the framework. This allows us to keep track of the various event sources, what environment they are used in, and keep the same variables across all event sources that will contain different values. We used the following structure:

```
/<project>/<service>/<environment>/auth_token
```

Note: If you would like use Github in your workflow to host your code and/or configuration, you will need to generate a Personal Access Token in GitHub, and store the token in AWS Secrets Manager. We recommend creating a personal access token per repo / event source, and naming it accordingly (as per below) in AWS Secrets Manager:

```
"<project>/<service>/<environment>/codepipeline/github"
```

A single token can be used across multiple stacks as long as it has necessary scopes, however if the token needs to be revoked/rotated for some reason, this would affect multiple stacks.

See *Deployment* section for more details.

5. **A CodeCommit or separate Github repository:**

   This is where your configuration files will be stored in addition to the buildspec and deploy scripts. If you parameterize all your sensitive data, there shouldn't be any risk of sensitive data disclosure.

   Note: The "env:" variables must be populated, these can essentially be whatever you want them to be provided they fall in line with the naming conventions the cloudformation and serverless scripts expect.

6. **GitHub Framework Repo:**

   This repo contains all the templates you'll need to build out the pipeline. The cloudformation template will automatically build your CI/CD pipeline which will deploy the serverless application. You should not need to make any changes to it unless you plan to modify it for your own purposes, which is entirely OK with us!

7. **Pick a Cloudformation template:**

   If you choose to use the multiple source cloudformation template, you'll need:

   - A private S3 bucket to hold the merge lambda code (merge.zip)
   - Add an archive of your zipped configuration files (templates are in the framework Github repo you clone) to the bucket you use for the merge lambda.

CHAPTER 4

Deployment

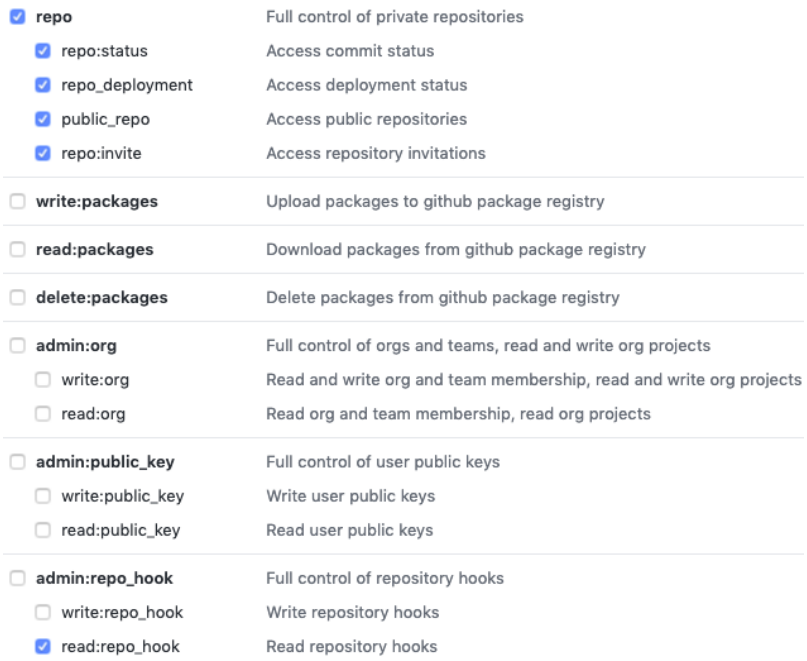
This guide uses Zoom as our service, your service may be different, be sure to name it something appropriately. The environment used in our examples is set to dev as the default.

4.1 Getting Started

1. Clone the master branch of the repository at "https://github.com/mozilla/mozdef-event-framework" to the local file system.
2. Before any action, decide on the various parameter and environment variable values, as these are referenced throughout the CF templates and serverless framework configuration. The following are the environment variables with example values:
   - PROJECT: MozDef-EF
   - SERVICE: zoom
   - STAGE: dev
   - API_PATH: events
   - Stack name: zoom2mozdef

   Note: The rest of these steps are not needed if you will not use Github in your workflow.

3. If you would like to use Github as a repository (either as the single repo to host everything, or in the multi-source scenario), create:
   - A Github repository,
   - A Personal Access Token (Your account settings -> Developer Settings -> Personal access token)
   - It is sufficient for the generated token to have the following scopes:



4. Decide on how you would like store your GitHub token using AWS Secrets Manager. You will have to specify this as a stack parameter during deployment. You should determine the name of the secret so CloudFormation can refer to it. We recommend this to be specific for your stack, for instance:

   "<project>/<service>/<environment>/codepipeline/github"

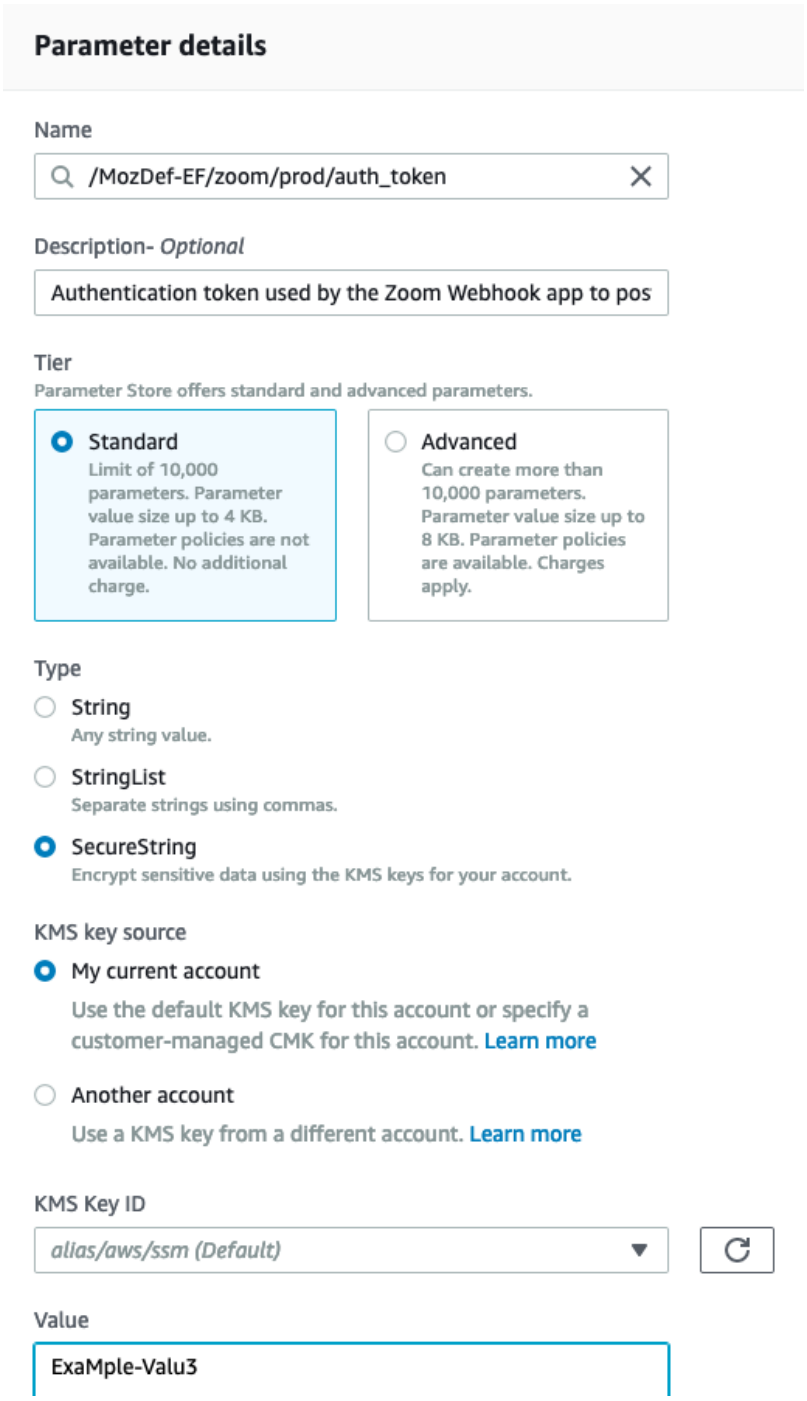
4.2 Steps in the AWS Console

You'll need to log into the AWS console or you can alternatively use the aws-cli to create the parameters needed for your webhook to properly authenticate:

1. Login to AWS Console using your account credentials. This framework has been tested to work with federated logins where the user assumes a role.
2. Navigate to "Services" and under "Management and Governance" select "Systems Manager"
3. Once the page loads, under "Application Management, choose "Parameter Store"
4. Create a parameter there in the form of:

   "/<project>/<service>/<environment>/auth_token"

   Example: For our Zoom example, this would be something like:



5. In the description field, add "Authentication token used by the Webhook app to post events".
6. For type, select "SecureString".
7. For value, paste the value from the app's configuration from your webhook's configured authentication token.
8. Add a Tag with key: Project and value: <project>-<environment>, then create the parameter.

Note: The rest of these steps are not needed if you will not use Github in your workflow.

---

9. If you are using Github as a repository, you need to store the personal access token value in AWS Secrets Manager. Navigate to "Services" and select "Secrets Manager".
10. Store a new secret of type "Other type of secrets".
11. Specify the key/value pair as "PersonalAccessToken" (without quotes) and the value of the token and click next.
12. For the secret name, enter the name you determined in step 4 of *Getting Started*.
13. Add a description and a tag using these "Project" as key and <project>-<environment> as the value. Click next.
14. Configure if you would like to automatic rotation of this secret. Click Next.
15. Review the details and click store when ready.

Note: The next step is *only* required if you would like to use the multi-source deployment option.

---

16. If you would like to keep the source code for the framework and its configuration separately, you will need to use a merger function (as lambda). We have provided this code as a ZIP archive in the folder "templates/multi_source_template/merge_function". For this to be used as a part of the pipeline, you need to create an S3 bucket and upload this ZIP file to that bucket.

    Once created, take a note of the name of the S3 bucket, as you will have to provide this as a stack parameter during deployment. We used a default name of "mozdef-ef-helper-bucket".

4.3 Fill in the Framework Config Template

The following should be done in your local copy of the framework you cloned or forked:

1. Open the downloaded repository in an IDE to edit locally.
2. Under config directory, edit "buildspec-dev.yml" file to contain:

```
version: 0.2

env:
  variables:
    STAGE: dev
    SERVICE: zoom
    PROJECT: MozDef-EF
    API_PATH: zoom
    TOKEN_ARN: arn:aws:ssm:$AWS_REGION:<ACCOUNT_ID>:parameter/
↪<parameter-name>

phases:
  install:
    runtime-versions:
      python: 3.7
      nodejs: 10
    commands:
      # Install dependencies here
      - pip3 install --upgrade awscli -q
      - pip3 install --upgrade pytest -q
      - pip3 install --upgrade moto -q
```

(continues on next page)

(continued from previous page)

```
      - pip3 install --upgrade aws-xray-sdk -q
      - npm install -g --silent --progress=false serverless
      - npm install --silent --save-dev serverless-pseudo-parameters
      - npm install --silent --save-dev serverless-prune-plugin
      # Remove or comment out the next line if you are not using
      # "serverless-python-requirements" plugin to manage 3rd party␣
↪Python libraries
      - npm install --silent --save-dev serverless-python-
↪requirements
  pre_build:
    commands:
      # Perform pre-build actions here
      - chmod +x $CODEBUILD_SRC_DIR/config/deploy.sh
      - $CODEBUILD_SRC_DIR/config/deploy.sh unit-test
  build:
    commands:
      # Invoke the deploy script here
      - $CODEBUILD_SRC_DIR/config/deploy.sh deploy $STAGE $AWS_REGION
```

3. The important part here is the filling in of the "env" section at the top of the file. These environment variables will be used by the "serverless.yml" file when deployed by the serverless framework. For each service deployed for a source (such as zoom), the service name and API path will be different.
4. Save the file.
5. Make any other desired changes on the local copy. For webhook based services, like zoom, there should not be any additional changes needed.

4.4 Deploy Your Framework

This is where we take everything we've done up to this point and start the deployment.

1. Using a Single Source Repo (CodeCommit)

1. Go back to AWS Console "Services -> CodeCommit" and create a repository with the name "<project>-<service>", in this case "mozdef-ef-zoom". Add a description and a tag with key: Project and value: <project>-<environment>.
2. Using the connection settings, setup Git access with the git credential helper over HTTPS (ensure you can pull and push to the newly created repo)
3. Pull the empty repository to a local directory, then add/move all the cloned and updated framework code to this repository. Add and commit all changes, then push.

Note: Make sure to not override the `.git` directory while moving the framework code from Github to your newly created custom CodeCommit repository.

Note: Currently, the Serverless app uses serverless-python-requirements plugin to manage 3rd party libraries. Do not forget to update the `requirements.txt` with your dependent libraries/modules.

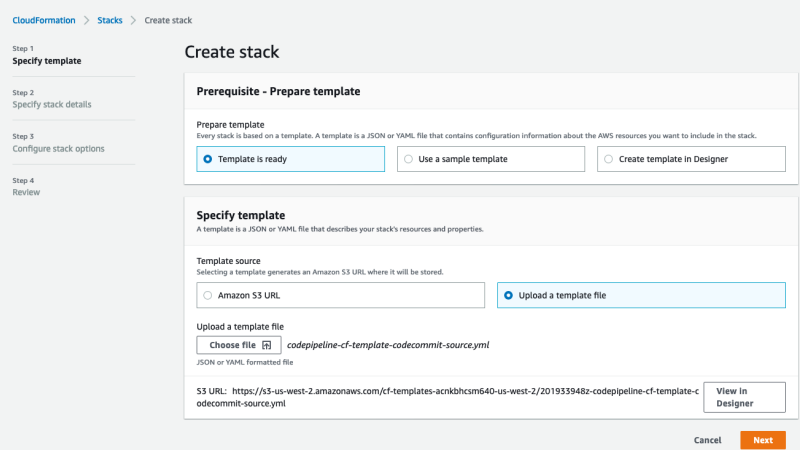
Also, if you would like to manage the requirements in another way (without this plugin), do not forget to update the `serverless.yml` file accordingly.

---

4. Go to “Services -> CloudFormation” on the AWS Console.
5. On top right, click “Create stack (with new resources)”
6. Select “template is ready” on the first option. In “specify template” menu, select “upload a template file”

   Create stack from Template:

   Example screenshot for creating a stack from the template



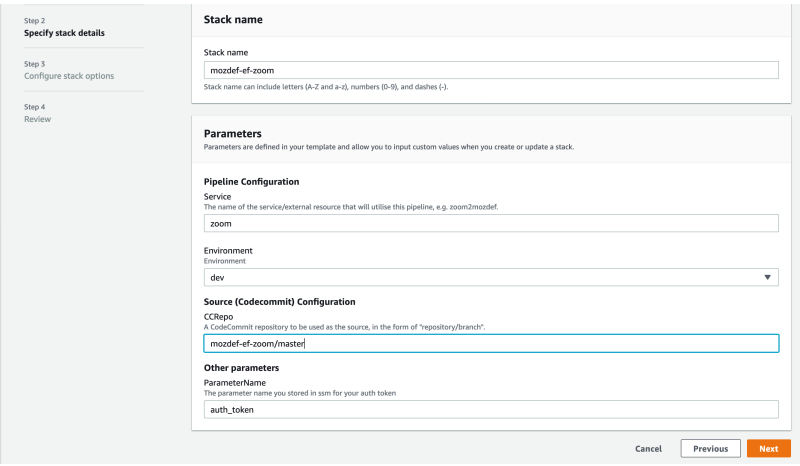
7. Browse the filesystem, and select the “codepipeline-cf-template-codecommit-source.yml” CloudFormation template under the “templates” directory of the cloned and updated framework code. Assuming no syntax errors, click next.
8. For the stack name, enter something descriptive, like: <project>-<service> (e.g., mozdef-ef-zoom, see the example image below for steps 8 through 12).
9. For stack parameters, enter the values decided in *Getting Started* Step 2.

   - For service, enter your <service> name that you determined earlier.
   - For environment, choose “dev”, “staging”, or “prod” according to the environment you are working out of.
   - In the SSMParamName field you’ll need to enter the name of the SSM parameter used to store the auth token to correctly map the IAM permissions for this resource.

10. An S3 utility bucket will be created for AWS CodePipeline to store artifacts. The bucket name will match the parameters you created for your stack name in step 8 and the environment in step 11 (e.g., <stackname>-<environment>-utility)
11. For source configuration, enter the name of the codecommit repo created in step 1, and the branch to monitor for changes and trigger rebuilds of the deployment. For our example we used zoom, “mozdef-ef-zoom/master”.

    Stack Details:

    Example screenshot for steps 8 through 12



12. Under stack options, add a tag with key: "Project" and value: <project>-<environment>. Click Next
13. On the review step, check the box under "Capabilities" saying "I acknowledge that AWS CloudFormation might create IAM resources with custom names.".
14. Click Create Stack. On the Cloudformation page, check the stack creation status. It should deploy the pipeline stack successfully.
15. Once the API Gateway has been created, copy the URL into your webhook application's configuration as the endpoint to post events to begin sending events to the AWS infra that was deployed using this framework.

Note: There is a stack output called `ServiceEndpoint` which is generated by the Serverless Framework after deployment. If you specified a custom `API_PATH` in your buildspec in *Getting Started*, this will *not* be your endpoint URL. Instead, your correct endpoint URL would be the value of `<ServiceEndpoint>/<API_PATH>`

---

2. Using a Single Source Repo (GitHub)

1. Pull the empty Github repository created earlier in section *Getting Started* to a local directory, then add/move all the cloned and updated framework code to this repository. Add and commit all changes, then push.
   - Do not forget to modify the `deploy.sh` configuration file to remove the reference to `$CODEBUILD_SRC_DIR` environment variable (as it is specific to CodeCommit build image).

Note: Make sure to not override the `.git` directory while moving the framework code from Github to your newly created custom GitHub repository.

Note: Currently, the Serverless app uses serverless-python-requirements plugin to manage 3rd party libraries. Do not forget to update the `requirements.txt` with your dependent libraries/modules.

Also, if you would like to manage the requirements in another way (without this plugin), do not forget to update the `serverless.yml` file accordingly.

---

2. Go to "Services -> CloudFormation" on the AWS Console.

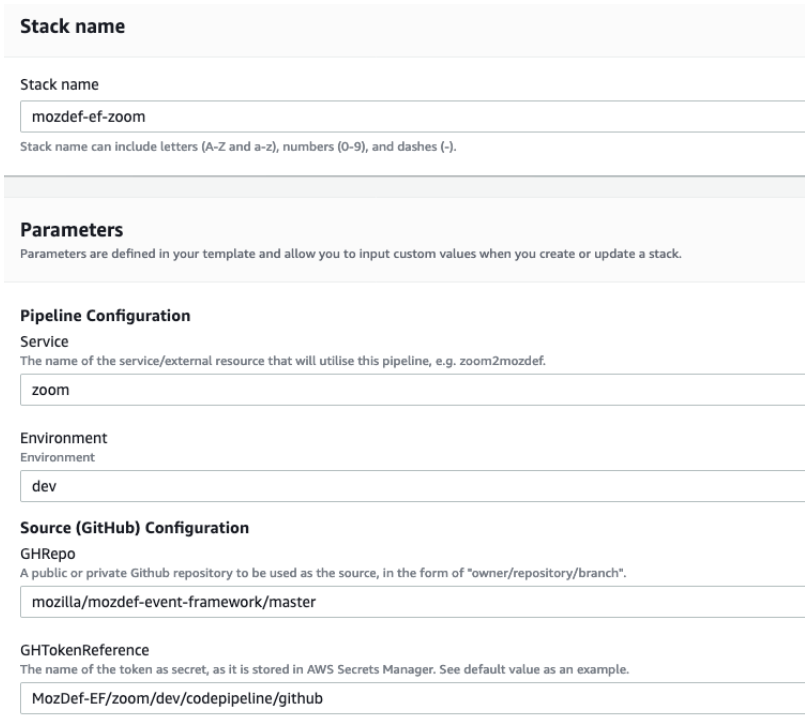
3. On top right, click "Create stack (with new resources)"
4. Select "template is ready" on the first option. In "specify template" menu, select "upload a template file.
5. Browse the filesystem, and select the "codepipeline-cf-template-github-source.yml" CloudFormation template under the "templates" directory of the cloned and updated framework code. Assuming no syntax errors, click next.
6. For the stack name, enter something descriptive, like: <project>-<service> (e.g., mozdef-ef-zoom, see the example image below for steps 6 through 10).
7. For stack parameters, enter the values decided in *Getting Started* Step 2.

- For service, enter your <service> name that you determined earlier.
- For environment, choose "dev", "staging", or "prod" according to the environment you are working out of.
- In the SSMParamName field you'll need to enter the name of the SSM parameter used to store the auth token to correctly map the IAM permissions for this resource.

8. An S3 utility bucket will be created for AWS CodePipeline to store artifacts. The bucket name will match the parameters you created for your stack name in step 8 and the environment in step 11 (e.g., <stackname>-<environment>-utility)
9. For source configuration:
    - Enter the name of the Github repo housing the code, in the following format: *owner/repository/branch*.
    - For the token reference, enter the name you determined in step 4 of *Getting Started*. This way the template will be able to find the secret (Github token) stored in AWS Secrets Manager.

Stack Details:

Example screenshot for creating a stack from the template with GitHub



10. Under stack options, add a tag with key: "Project" and value: <project>-<environment>. Click Next.
11. On the review step, check the box under "Capabilities" saying "I acknowledge that AWS CloudFormation might create IAM resources with custom names.".
12. Click Create Stack. On the Cloudformation page, check the stack creation status. It should deploy the pipeline stack successfully.
13. Once the API Gateway has been created, copy the URL into your webhook application's configuration as the endpoint to post events to begin sending events to the AWS infra that was deployed using this framework.

Note: There is a stack output called `ServiceEndpoint` which is generated by the Serverless Framework after deployment. If you specified a custom `API_PATH` in your buildspec in *Getting Started*, this will *not* be your endpoint URL. Instead, your correct endpoint URL would be the value of `<ServiceEndpoint>/<API_PATH>`

---

3. Using Multiple Source Repos (CodeCommit + Github)

1. Go back to AWS Console "Services -> CodeCommit" and create a repository with the name "<project>-<service>", in this case "mozdef-ef-zoom". Add a description and a tag with key: Project and value: <project>-<environment>.
2. Using the connection settings, setup Git access with the git credential helper over HTTPS (ensure you can pull and push to the newly created repo).
3. Pull the empty repository to a local directory, then *only* add the *config* directory contents from the cloned framework code to this repository. Make relevant configuration changes (such as to the deploy script, buildspec etc.), commit all changes, then push.
4. Now, also pull the empty Github repository created earlier in section *Getting Started* to another local directory. Add/move all the cloned framework code to this repository, *except "config" directory*.

Make changes to the code if desired, commit all changes, then push.

Note: You could also move everything to this repository (including the config directory), but add "config/" to the *.gitignore* file in order to avoid having multiple config directories tracked by source control.

Also make sure to not override the `.git` directory while moving the framework code from Github to your newly created custom repositories.

Note: Currently, the Serverless app uses serverless-python-requirements plugin to manage 3rd party libraries. Do not forget to update the `requirements.txt` with your dependent libraries/modules.

Also, if you would like to manage the requirements in another way (without this plugin), do not forget to update the `serverless.yml` file accordingly.

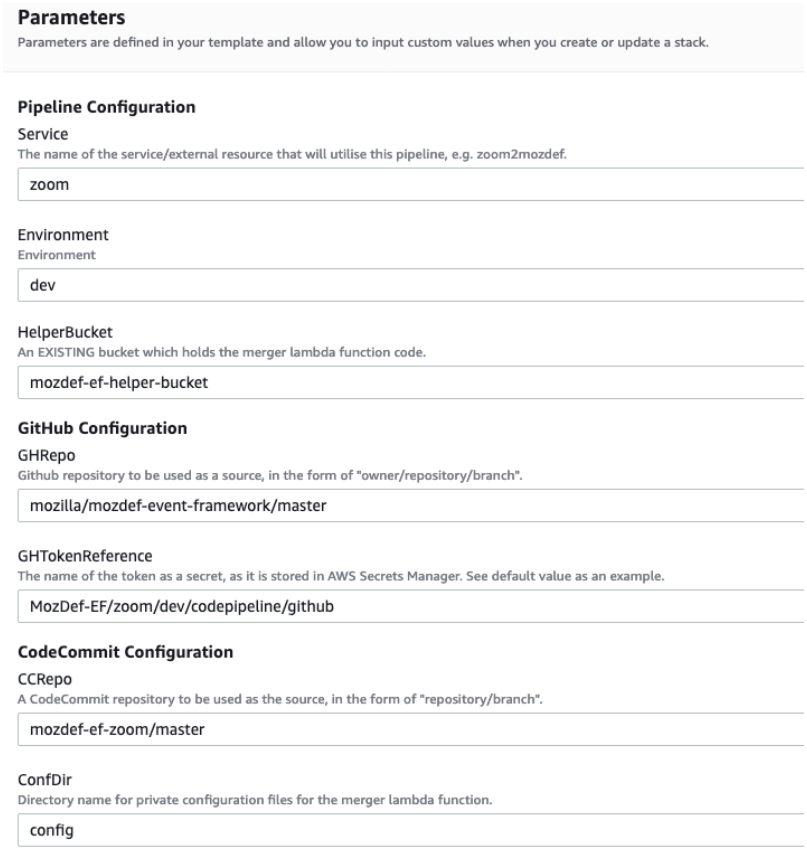
---

5. Go to "Services -> CloudFormation" on the AWS Console.
6. On top right, click "Create stack (with new resources)"
7. Select "template is ready" on the first option. In "specify template" menu, select "upload a template file".
8. Browse the filesystem, and select the "codepipeline-cf-template-with-merge.yml" CloudFormation template under the "templates" directory of the cloned framework code. Assuming no syntax errors, click next.
9. For the stack name, enter something descriptive, like: <project>-<service> (e.g., mozdef-ef-zoom, see the example image below for steps 9 through 14).
10. For stack parameters, enter the values decided in *Getting Started* Step 2.
    - For service, enter your <service> name that you determined earlier.
    - For environment, choose "dev", "staging", or "prod" according to the environment you are working out of.
    - For helper bucket, enter the name of the S3 bucket created previously (created in the last step of *Steps in the AWS Console* section) that houses the merge lambda code.
    - In the SSMParamName field you'll need to enter the name of the SSM parameter used to store the auth token to correctly map the IAM permissions for this resource.
11. An S3 utility bucket will be created for AWS CodePipeline to store artifacts. The bucket name will match the parameters you created for your stack name in step 8 and the environment in step 11 (e.g., <stackname>-<environment>-utility)
12. For GitHub configuration:
    - Enter the name of the Github repo housing the code, in the following format: *owner/repository/branch*.
    - For the token reference, enter the name you determined in step 4 of *Getting Started*. This way the template will be able to find the secret (Github token) stored in AWS Secrets Manager.
13. For CodeCommit configuration:
    - Enter the name of the codecommit repo created in step 1, and the branch to monitor for changes and trigger rebuilds of the deployment. For our example we used zoom, "mozdef-ef-zoom/master".

- Enter the name of the directory that has configuration data for the pipeline (default: config).

Stack Details:

Example screenshot for creating a stack using multi-source template



14. Under stack options, add a tag with key: "Project" and value: <project>-<environment>. Click Next.
15. On the review step, check the box under "Capabilities" saying "I acknowledge that AWS CloudFormation might create IAM resources with custom names.".
16. Click Create Stack. On the Cloudformation page, check the stack creation status. It should deploy the pipeline stack successfully.
17. Once the API Gateway has been created, copy the URL into your webhook application's configuration as the endpoint to post events to begin sending events to the AWS infra that was deployed using this framework.

Note: There is a stack output called `ServiceEndpoint` which is generated by the Serverless Framework after deployment. If you specified a custom `API_PATH` in your buildspec in *Getting Started*, this will *not* be your endpoint URL. Instead, your correct endpoint URL would be the value of `<ServiceEndpoint>/<API_PATH>`

CHAPTER 5

Examples

We've included some sample scripts you can modify for your deployment.

You'll want to replace the [env:variables]: for 'SERVICE' and 'API_PATH' to whatever you have decided upon for your event source pipeline and API gateway path.

5.1 Buildspec Example

buildspec.yml:

```
version: 0.2

env:
  variables:
    STAGE: dev
    SERVICE: zoom
    PROJECT: MozDef-EF
    API_PATH: zoom
    TOKEN_ARN: arn:aws:ssm:$AWS_REGION:<ACCOUNT_ID>:parameter/<parameter-name>

phases:
  install:
    runtime-versions:
      python: 3.7
      nodejs: 10
    commands:
      # Install dependencies here
      - pip3 install --upgrade awscli -q
      - pip3 install --upgrade pytest -q
      - pip3 install --upgrade moto -q
      - pip3 install --upgrade aws-xray-sdk -q
      - npm install -g --silent --progress=false serverless
      - npm install --silent --save-dev serverless-pseudo-parameters
```

(continues on next page)

(continued from previous page)

```
      - npm install --silent --save-dev serverless-prune-plugin
      # Remove or comment out the next line if you are not using
      # "serverless-python-requirements" plugin to manage 3rd party Python libraries
      - npm install --silent --save-dev serverless-python-requirements
  pre_build:
    commands:
      # Perform pre-build actions here
      - chmod +x $CODEBUILD_SRC_DIR/config/deploy.sh
      - $CODEBUILD_SRC_DIR/config/deploy.sh unit-test
  build:
    commands:
      # Invoke the deploy script here
      - $CODEBUILD_SRC_DIR/config/deploy.sh deploy $STAGE $AWS_REGION
```

5.2 Deploy Script Example

deploy.sh:

```
#!/bin/bash

# This is deploy script template. Feel free modify per service/resource.

instruction()
{
  echo "-----------------------------------------------"
  echo "usage: ./deploy.sh deploy <env>"
  echo "env: eg. dev, staging, prod, ..."
  echo "for example: ./deploy.sh deploy dev"
  echo ""
  echo "to test: ./deploy.sh <int-test|unit-test>"
  echo "for example: ./deploy.sh unit-test"
}

if [ $# -eq 0 ]; then
    instruction
    exit 1
elif [ "$1" = "int-test" ] && [ $# -eq 1 ]; then
    python3 -m pytest "$CODEBUILD_SRC_DIR/tests/int-tests/"

elif [ "$1" = "unit-test" ] && [ $# -eq 1 ]; then
    python3 -m pytest "$CODEBUILD_SRC_DIR/tests/unit-tests/"

elif [ "$1" = "deploy" ] && [ $# -eq 3 ]; then
    STAGE=$2
    REGION=$3
    STATE=`aws cloudformation describe-stacks --stack-name "$SERVICE-$STAGE" \
        --query Stacks[*].StackStatus --output text | grep -E "ROLLBACK|FAIL" -c`
    # Forcefully remove the stack deployed by Serverless
    # framework ONLY IF previous build errored
    # NOTE: This is probably only a good idea for dev stage
    if [ $STATE -ne 0 ]; then
       sls remove -s $STAGE -r $REGION --force
    fi
    sleep 2
```

(continues on next page)

(continued from previous page)

```
    # Try to deploy again after stack removal
    sls deploy -s $STAGE -r $REGION --force

else
    instruction
    exit 1
fi
```

CHAPTER 6

Contributors

Here is the list of the awesome contributors helping us or that have helped us in the past:

Contributors

CHAPTER 7

Indices and tables

- genindex
- modindex
- search