## Details

Learn how to build a modern application in this guided workshop. Join Nicki and AM for a brief introduction to the concepts of modern app development, step-by-step instructions to build a sample app, nerdy witty banter, and probably some side commentary around anything .NET. We invite all developers/programmers to attend – no AWS is experience necessary. Familiarity with Python will be helpful, but is not required.  
  
Modern applications are built as independent microservices that communicate over well-defined APIs. They are run by small, independent teams using code packaged so it can be run anywhere. This makes applications faster to develop and easier to scale.  
  
The workshop will walk you through the steps to create the sample web application. You will learn to host the app on a front-end web server and connect it to a backend database. You’ll also learn to set up user authentication and will be able to collect and analyze user behavior.  
  
By the end of the workshop, you will build a sample website called Mythical Mysfits that enables visitors to adopt a fantasy creature (mysfit) as pet. You can see a working sample of this website at: [www.mythicalmysfits.com](http://www.mythicalmysfits.com/)  
  
Please BRING YOUR LAPTOP!  
Be prepared and make sure you have the following applications installed on your computer:  
1) Sign up for an AWS account <https://portal.aws.amazon.com/billing/signup>  
2) Get administrator-level access to you AWS account <https://docs.aws.amazon.com/IAM/latest/UserGuide/getting-started_create-admin-group.html>  
3) Bring your laptop  
  
SCHEDULE  
Come join us for an amazing evening!  
• 5:00 – 5:30pm: Intros & Happy Hour  
• 5:30 – 5:45pm: Overview of Workshop & Modern Apps  
• 5:45 – 8:30pm: Workshop  
  
WHAT YOU WILL LEARN  
This workshop will help you understand how developers are able to build modern applications by interacting with features and services through the development tools that AWS provides. By the end of the tutorial, you will use Amazon Simple Storage Service (S3), AWS Fargate, AWS API Gateway, Amazon DynamoDB, AWS Lambda, Amazon Kinesis Firehose and more. You will be creating and deploying changes to the sample application completely programmatically. You will use the AWS Command Line Interface to execute commands that create the required infrastructure components, which includes a fully managed CI/CD stack utilizing AWS CodeCommit, CodeBuild, and CodePipeline. Finally, you will complete the development tasks required all within your own browser using the cloud-based IDE, AWS Cloud9.  
  
Additional support will be available during the tutorial to troubleshoot and answer questions. This is a great event to meet fellow developers, get hands-on experience with AWS services, and hang out with like-minded technologists who are interested in building modern web apps!

Speakers:

AM

Nicki Klein

Technology Evangelist

Micro-service

Containers

Dockers

Development language in Python.

Also available in .NET

Soon available in Java and JavaScript

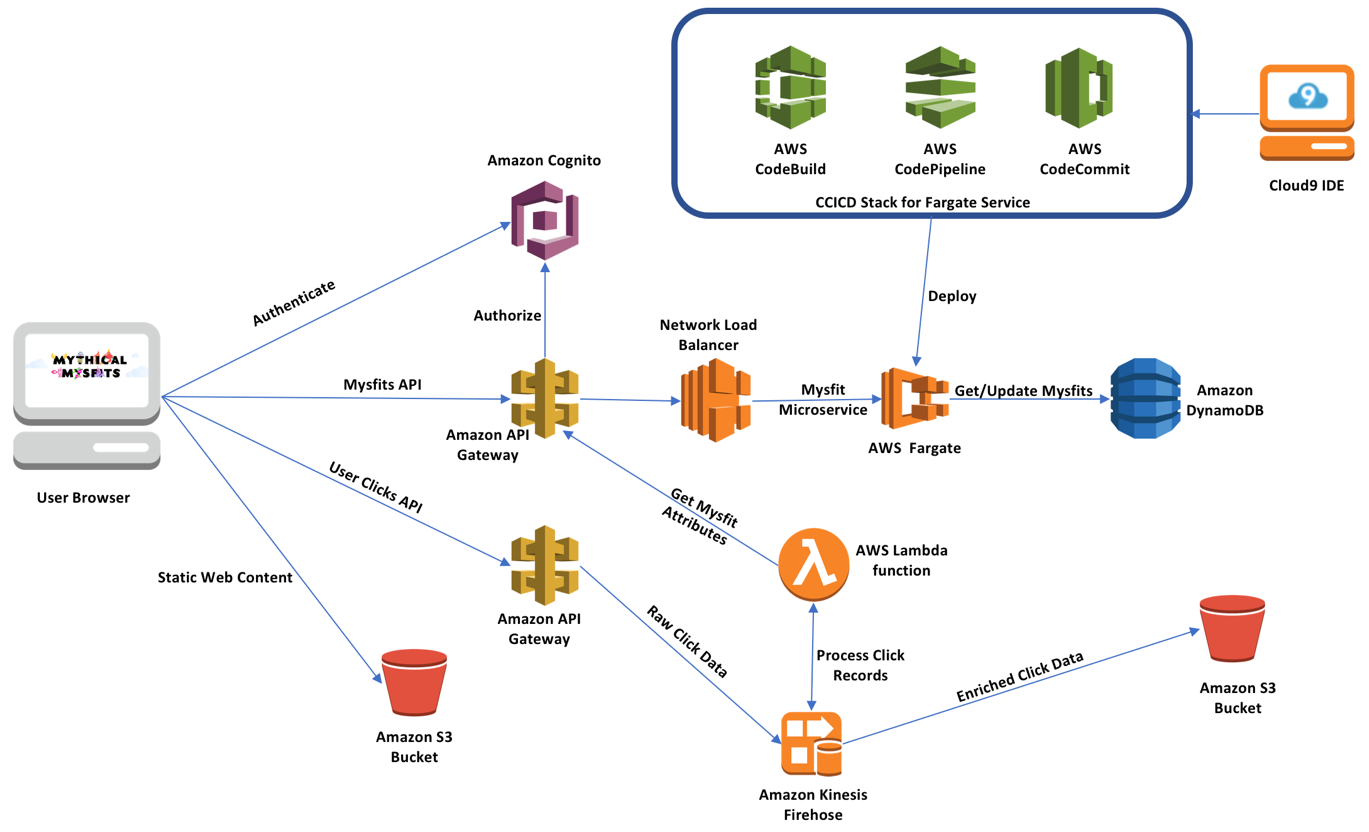
Justin perole?

dev.aws

redirects to <https://aws.amazon.com/developer/>

Tutorial

<https://aws.amazon.com/getting-started/projects/build-modern-app-fargate-lambda-dynamodb-python/>



What is a container?

* Apps on container
* Runs multiple containers in 1 server
* Consistency across the board

Container analogies

* Traditional server is more like a pet, feed it water it, death = funeral
* Containers are treated like cattle, when 1 cattle die, it will be replaced (constantly)
* 1980s coffee, if it broke, I would be sad,
* Styrofoam, you throw it away and replace

“Works on my machine” – dependency issue

Docker

* Engine that makes container easily work
* Writes the instructions of the layers
* Houses the environment/process
* Enclosed operating system
* Immediate benefit, access on production code.
* Without Docker, Debugging in production is terrifying.
* Standardization process
* More resiliency

Then push it to Elastic Container – Amazon ACR

Then create a cluster in Fargate

Container orchestration

* Managing multiple stateless containers
* Micro-services are containerized, i.e. talking to a CRUD container
* Kubernetes is an example
* Amazon ECS is an example

Orchestration can be on/off but not if scaled.

Fargate

* Abstract things for you
* Provision clusters
* Do less
* Customizable – deployment to EC2

Amazon DynamoDB

* NoSQL DB
* Python files will now be changing, instead of pulling in JSON, it will pull in Dynamo

CI/CD pipeline

* Automate my process

Amazon Cognito

* Authorization

AWS Lamda function

* Process records

Amazon Kinesis Firehose

Amazon S3 bucket

Amazon API Gateway

* Layered on top of services

Tutorial would say why (pros and cons)

Traditionally, apps were run in 1 environment (monolithic architecture)

Now, services were broken into pieces = micro-services

It added productivity. Easy to push features. Feature A doesn’t affect Feature B.

**Start tutorial (should delete after, to not get charged)**

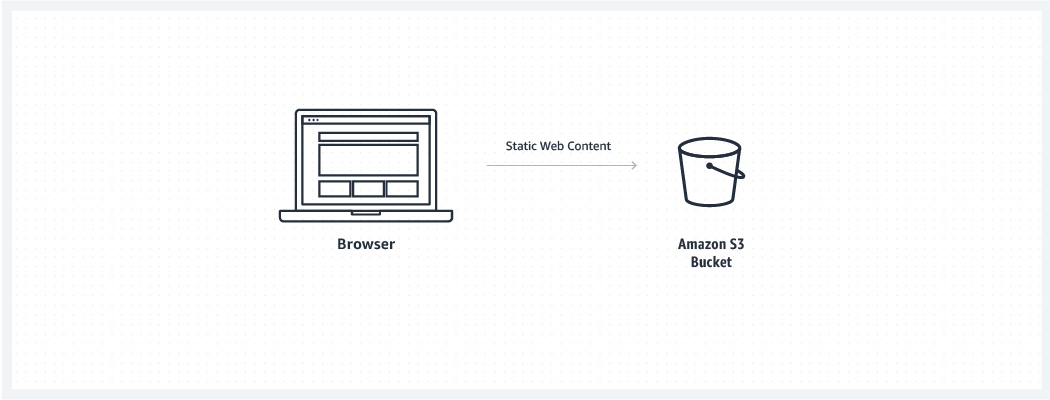
Module 1: Build a Static Website

## Overview

In this module, we'll host the static content (html, js, css, media content, etc.) of our Mythical Mysfit website on Amazon S3 (Simple Storage Service). S3 is a highly durable, highly available, and inexpensive object storage service that can serve stored objects directly via HTTP. This makes it wonderfully useful for serving static web content directly to web browsers for sites on the Internet.

Before we start storing our *mysfits* in S3, let's set up AWS Cloud9 for you. Cloud9 is a cloud-based integrated development environment (IDE) that lets you write, run, and debug your code with just a browser.

## Architecture Diagram



#### **Time to Complete**

20 minutes

#### **Services Used**

[AWS Cloud9](https://aws.amazon.com/cloud9/)  
[Amazon Simple Storage Service (S3)](https://aws.amazon.com/s3/)

AWS Cloud9

* Dev environment
* <https://us-east-2.console.aws.amazon.com/cloud9/home/product>
* Create a dev environment

In the AWS Cloud9 terminal

Clone the branch of the repository

* git clone –b python --single-branch https://github.com/aws-samples/aws-modern-application-workshop.git

Change directory

* cd aws-modern-application-workshop

Create S3 bucket

* aws s3 mb s3://mythical-mysfits-meetup-clark
* mb (make bucket)

Create website hosting

* aws s3 website s3://mythical-mysfits-meetup-clark --index-document index.html

In AWS Cloud9



"Resource": "arn:aws:s3:::REPLACE\_ME\_BUCKET\_NAME/\*"

Update S3 Bucket policy

* aws s3api put-bucket-policy --bucket mythical-mysfits-meetup-clark --policy file://~/environment/aws-modern-application-workshop/module-1/aws-cli/website-bucket-policy.json

Copy files to bucket

* aws s3 cp ~/environment/aws-modern-application-workshop/module-1/web/index.html s3://REPLACE\_ME\_BUCKET\_NAME/index.html
* cp (copy/paste)
* aws s3 cp ~/environment/aws-modern-application-workshop/module-1/web/index.html s3://mythical-mysfits-meetup-clark/index.html

For us-east-1 (N. Virginia), us-west-2 (Oregon), eu-west-1 (Ireland) use:

http://REPLACE\_ME\_BUCKET\_NAME.s3-website-REPLACE\_ME\_YOUR\_REGION.amazonaws.com

For us-east-2 (Ohio) use:

http://REPLACE\_ME\_BUCKET\_NAME.s3-website.REPLACE\_ME\_YOUR\_REGION.amazonaws.com

Working URL:

<http://mythical-mysfits-meetup-clark.s3-website.us-east-2.amazonaws.com/>

**End of module 1**

**Module 2** Host Your Application On A Web Server

In this module you will create a new microservice hosted with AWS Fargate.

* Container
* Push it
* Create a fargate
* End, CI/CD pipeline

Load-balancer

* Route the traffic of requests evenly to the containers running

### **Module 2A: Setup Core Infrastructure**

Code formation

* Infrastructure as Code
* Json file/ Yaml file will create all we need

Directory

cd ~/environment/aws-modern-application-workshop/

AWS/CLI

aws cloudformation create-stack --stack-name MythicalMysfitsCoreStack --capabilities CAPABILITY\_NAMED\_IAM --template-body file://~/environment/aws-modern-application-workshop/module-2/cfn/core.yml

Change directory

cd ~/environment/aws-modern-application-workshop/module-2/app

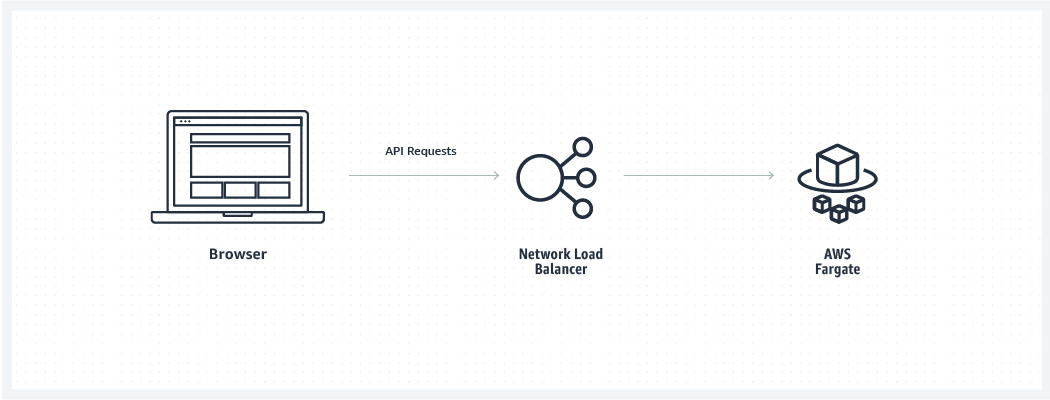
aws sts get-caller-identity

* {
* "Account": "795901041651",
* "UserId": "795901041651",
* "Arn": "arn:aws:iam::795901041651:root"

}

### **Module 2B: Deploy A Service With AWS Fargate**

#### **Architecture Diagram**



#### **Step 1: Create a Flask Service**

**Create a Flask Service**

##### A: Build A Docker Image

docker build . -t REPLACE\_ME\_AWS\_ACCOUNT\_ID.dkr.ecr.REPLACE\_ME\_REGION.amazonaws.com/mythicalmysfits/service:latest

docker build . -t 795901041651.dkr.ecr.us-east-2.amazonaws.com/mythicalmysfits/service:latest

this is your tag:

795901041651.dkr.ecr.us-east-2.amazonaws.com/mythicalmysfits/service:latest

##### B: Test The Service Locally

docker run -p 8080:8080 REPLACE\_ME\_WITH\_DOCKER\_IMAGE\_TAG

docker run -p 8080:8080 -t 795901041651.dkr.ecr.us-east-2.amazonaws.com/mythicalmysfits/service:latest

inside AWS Cloud9 window

<https://fa28d59b2c964f67bdc1eaf93a1f7d9c.vfs.cloud9.us-east-2.amazonaws.com/mysfits>

C: Push the Docker Image to Amazon ECR

Push in Amazon Elastic Container Registry

Change in directory

/environment

aws ecr create-repository --repository-name mythicalmysfits/service

$ (aws ecr get-login --no-include-email)

* include the $ sign, run command on commands
* AWS/CLI has a helper method that starts with a login, and have access to push to a repository
* Spits a command, JWT tokten

docker push 795901041651.dkr.ecr.us-east-2.amazonaws.com/mythicalmysfits/service:latest

* ec2-user:~/environment $ docker push 795901041651.dkr.ecr.us-east-2.amazonaws.com/mythicalmysfits/service:latest
* The push refers to repository [795901041651.dkr.ecr.us-east-2.amazonaws.com/mythicalmysfits/service]
* eb682fa20517: Pushed
* 275beb2f1bd4: Pushed
* d715adeddeae: Pushed
* b0af508ada7a: Pushed
* 8193e0e04b5c: Pushed
* 8d7ea83e3c62: Pushed
* 6a061ee02432: Pushed
* f73b2816c52a: Pushed
* 6267b420796f: Pushed
* a30b835850bf: Pushed

latest: digest: sha256:51a86a10ae6da58554065e0484e42d798741ebdc2bf4f46f073ebf4405a3b507 size: 2413

aws ecr describe-images --repository-name mythicalmysfits/service

* ec2-user:~/environment $ aws ecr describe-images --repository-name mythicalmysfits/service
* {
* "imageDetails": [
* {
* "imageSizeInBytes": 209092387,
* "imageDigest": "sha256:51a86a10ae6da58554065e0484e42d798741ebdc2bf4f46f073ebf4405a3b507",
* "imageTags": [
* "latest"
* ],
* "registryId": "795901041651",
* "repositoryName": "mythicalmysfits/service",
* "imagePushedAt": 1538013922.0
* }
* ]

}

##### A: Create An AWS Fargate Cluster

aws ecs create-cluster --cluster-name MythicalMysfits-Cluster

##### B: Create An AWS CloudWatch Logs Group

aws logs create-log-group --log-group-name mythicalmysfits-logs

* logging

##### C: Register An ECS Task Definition

A JSON file has been provided that will serve as the input to the CLI command.

Open ~/environment/aws-modern-application-workshop/module-2/aws-cli/task-definition.json in the IDE.

Replace the indicated values with the appropriate ones from your created resources.

"executionRoleArn": "REPLACE\_ME\_ECS\_SERVICE\_ROLE\_ARN",

"taskRoleArn": "REPLACE\_ME\_ECS\_TASK\_ROLE\_ARN",

To check

aws cloudformation describe-stacks --stack-name MythicalMysfitsCoreStack

To query specific output

aws cloudformation describe-stacks --stack-name MythicalMysfitsCoreStack --query Stacks[].Outputs

* "Description": "REPLACE\_ME\_ECS\_SERVICE\_ROLE\_ARN",
* "ExportName": "MythicalMysfitsCoreStack:EcsServiceRole",
* "OutputKey": "EcsServiceRole",
* "OutputValue": "arn:aws:iam::795901041651:role/MythicalMysfitsCoreStack-EcsServiceRole-9VDFBDHSWY0I"

},

* "Description": "REPLACE\_ME\_ECS\_TASK\_ROLE\_ARN",
* "ExportName": "MythicalMysfitsCoreStack:ECSTaskRole",
* "OutputKey": "ECSTaskRole",

"OutputValue": "arn:aws:iam::795901041651:role/MythicalMysfitsCoreStack-ECSTaskRole-12SEJQWBVEFAV"

COPY THE OUTPUT VALUES

Save

Then run

aws ecs register-task-definition --cli-input-json file://~/environment/aws-modern-application-workshop/module-2/aws-cli/task-definition.json

**Step 3: Enable A Load Balance Fargate Service**

##### A: Create A Network Load Balancer

aws elbv2 create-load-balancer --name mysfits-nlb --scheme internet-facing --type network --subnets REPLACE\_ME\_PUBLIC\_SUBNET\_ONE REPLACE\_ME\_PUBLIC\_SUBNET\_TWO

aws elbv2 create-load-balancer --name mysfits-nlb --scheme internet-facing --type network --subnets subnet-0d1cbb283986f57c4 subnet-04d4db0439e468a57

Copy the values at "DNSName":, "VpcId":, & "LoadBalancerArn": or save the full response provided by this command

"DNSName": "mysfits-nlb-30d50cb4673bfe36.elb.us-east-2.amazonaws.com",

"VpcId": "vpc-047179b4f2460166d",

"LoadBalancerArn": "arn:aws:elasticloadbalancing:us-east-2:795901041651:loadbalancer/net/mysfits-nlb/30d50cb4673bfe36",

##### B: Create A Load Balancer Target Group

Next, use the CLI to create an NLB \*\*target group\*\*.

aws elbv2 create-target-group --name MythicalMysfits-TargetGroup --port 8080 --protocol TCP --target-type ip --vpc-id REPLACE\_ME --health-check-interval-seconds 10 --health-check-path / --health-check-protocol HTTP --healthy-threshold-count 3 --unhealthy-threshold-count 3

aws elbv2 create-target-group --name MythicalMysfits-TargetGroup --port 8080 --protocol TCP --target-type ip --vpc-id vpc-047179b4f2460166d --health-check-interval-seconds 10 --health-check-path / --health-check-protocol HTTP --healthy-threshold-count 3 --unhealthy-threshold-count 3

Copy the value at ```"TargetGroupArn":``` or save the \*\*full response\*\* from the above command as well, which contains the Target Group ARN to be used in the next step.

"TargetGroupArn": "arn:aws:elasticloadbalancing:us-east-2:795901041651:targetgroup/MythicalMysfits-TargetGroup/1f61202d31cd87b7",

##### C: Create A Load Balancer Listener

Next, use the CLI to create a load balancing listener for the NLB.

aws elbv2 create-listener --default-actions TargetGroupArn=REPLACE\_ME,Type=forward --load-balancer-arn REPLACE\_ME --port 80 --protocol TCP

aws elbv2 create-listener --default-actions TargetGroupArn=arn:aws:elasticloadbalancing:us-east-2:795901041651:targetgroup/MythicalMysfits-TargetGroup/1f61202d31cd87b7,Type=forward --load-balancer-arn arn:aws:elasticloadbalancing:us-east-2:795901041651:loadbalancer/net/mysfits-nlb/30d50cb4673bfe36 --port 80 --protocol TCP

**Step 4: Create A Service With Fargate**

##### A: Create A Service Linked Role For ECS

Without creating this role, the ECS service would not be granted permissions to perform the actions required. To create the role, execute the following command in the terminal:

aws iam create-service-linked-role --aws-service-name ecs.amazonaws.com

##### B: Create The Service

ith the NLB created and configured, and the ECS service granted appropriate permissions, we're ready to create the actual ECS \*\*service\*\* where our containers will run and register themselves to the load balancer to receive traffic.

Open ~/environment/aws-modern-application-workshop/module-2/aws-cli/service-definition.json in the IDE and replace the indicated values of REPLACE\_ME. Save it, then run the following command to create the service:

aws ecs create-service --cli-input-json file://~/environment/aws-modern-application-workshop/module-2/aws-cli/service-definition.json

##### C: Test The Service

http://mysfits-nlb-123456789-abc123456.elb.us-east-1.amazonaws.com/mysfits

In the Chrome browser,

http://mysfits-nlb-30d50cb4673bfe36.elb.us-east-2.amazonaws.com/mysfits

**Step 5: Update Mythical Mysfits To Call The NLB**

##### A: Replace the API Endpoint

Next, we need to integrate our website with your new API backend instead of using the hard coded data that we previously uploaded to S3. You'll need to update the following file to use the same NLB URL for API calls (do not include the /mysfits path): /module-2/web/index.html

Open the file in Cloud9 and replace the highlighted area below between the quotes with the NLB URL:

var mysfitsApiEndpoint = 'REPLACE\_ME'; // example: 'http://mythi-publi-abcd12345-01234567890123.elb.us-east-1.amazonaws.com'

var mysfitsApiEndpoint = http://mysfits-nlb-30d50cb4673bfe36.elb.us-east-2.amazonaws.com/mysfits';

##### B: Upload To S3

To upload this file to your S3 hosted website, use the bucket name again that was created during Module 1, and run the following command:

aws s3 cp ~/environment/aws-modern-application-workshop/module-2/web/index.html s3://INSERT-YOUR-BUCKET-NAME/index.html

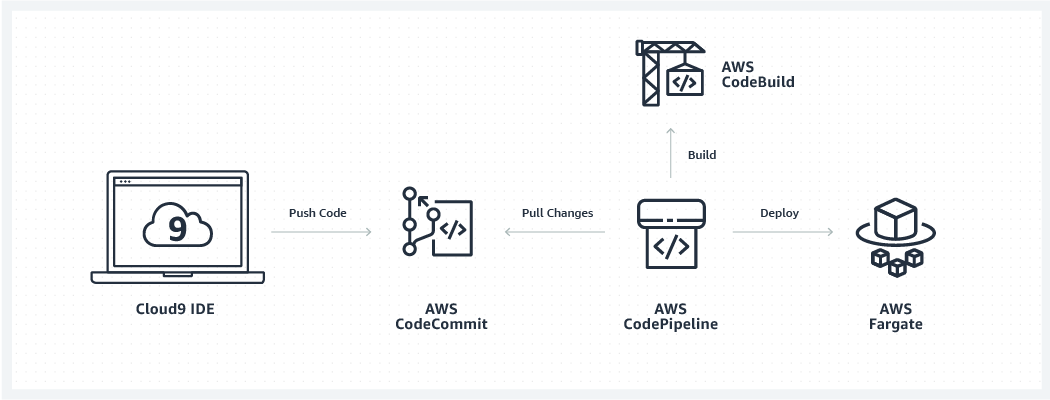
aws s3 cp ~/environment/aws-modern-application-workshop/module-2/web/index.html s3://mythical-mysfits-meetup-clark/index.html

Module 2C: Automate Deployments using AWS Code Services

Now that you have a service up and running, you may think of code changes that you'd like to make to your Flask service. It would be a bottleneck for your development speed if you had to go through all of the same steps above every time you wanted to deploy a new feature to your service. That's where Continuous Integration and Continuous Delivery or CI/CD come in!

In this module, you will create a fully managed CI/CD stack that will automatically deliver all of the code changes that you make to your code base to the service you created during the last module.

#### **Architecture Diagram**



**Step 1: Create The CI/CD Pipeline**

##### A: Create A S3 Bucket For Pipelie Artifacts

We will need to create another S3 bucket that will be used to store the temporary artifacts that are created in the middle of our CI/CD pipeline executions. Choose a new bucket name for these artifacts and create one using the following CLI command:

aws s3 mb s3://mythical-mysfits-artifacts-bucket-name

aws s3 mb s3://mythical-mysfits-artifacts-meetup-clark

Next, this bucket needs a bucket policy to define permissions for the data stored within it. But unlike our website bucket that allowed access to anyone, only our CI/CD pipeline should have access to this bucket. We have provided the JSON file needed for this policy at *~/environment/aws-modern-application-workshop/module-2/aws-cli/artifacts-bucket-policy.json*.

Open this file, and inside you will need to replace several strings to include the ARNs that were created as part of the MythicalMysfitsCoreStack earlier, as well as your newly chosen bucket name for your CI/CD artifacts.

Once you've modified and saved this file, execute the following command to grant access to this bucket to your CI/CD pipeline:

Then run

aws s3api put-bucket-policy --bucket mythical-mysfits-artifacts-bucket-name --policy file://~/environment/aws-modern-application-workshop/module-2/aws-cli/artifacts-bucket-policy.json

aws s3api put-bucket-policy --bucket mythical-mysfits-artifacts-meetup-clark --policy file://~/environment/aws-modern-application-workshop/module-2/aws-cli/artifacts-bucket-policy.json

##### B: Create a CodeCommit Repository

You'll need a place to push and store your code in. Create an \*\*AWS CodeCommit Repository\*\* using the CLI for this purpose:

##### C: Create a CodeBuild Project

With a repository to store our code in, and an S3 bucket that will be used for our CI/CD artifacts, lets add to the CI/CD stack with a way for a service build to occur. This will be accomplished by creating an AWS CodeBuild Project. Any time a build execution is triggered, AWS CodeBuild will automatically provision a build server to our configuration and execute the steps required to build our Docker image and push a new version of it to the ECR repository we created (and then spin the server down when the build is completed).

The steps for our build (which package our Python code and build/push the Docker container) are included in the *~/environment/aws-modern-application-workshop/module-2/app/buildspec.yml* file. The buildspec.yml file is what you create to instruct CodeBuild what steps are required for a build execution within a CodeBuild project.

To create the CodeBuild project, another CLI input file is required to be updated with parameters specific to your resources. It is located at *~/environment/aws-modern-application-workshop/module-2/aws-cli/code-build-project.json*. Similarly replace the values within this file as you have done before from the MythicalMysfitsCoreStackOutput. Once saved, execute the following with the CLI to create the project:

aws codebuild create-project --cli-input-json file://~/environment/aws-modern-application-workshop/module-2/aws-cli/code-build-project.json

##### D: Create a CodePipeline Pipeline

All of these steps are defined in a JSON file provided that you will use as the input into the AWS CLI to create the pipeline. This file is located at *~/environment/aws-modern-application-workshop/module-2/aws-cli/code-pipeline.json*, open it and replace the required attributes within, and save the file.

Once saved, create a pipeline in CodePipeline with the following command:

aws codepipeline create-pipeline --cli-input-json file://~/environment/aws-modern-application-workshop/module-2/aws-cli/code-pipeline.json

##### E: Enable Automated Access to ECR Image Repository

We have one final step before our CI/CD pipeline can execute end-to-end successfully. With a CI/CD pipeline in place, you won't be manually pushing container images into ECR anymore. CodeBuild will be pushing new images now.

We need to give CodeBuild permission to perform actions on your image repository with an ECR repository policy\*. The policy document needs to be updated with the specific ARN for the CodeBuild role created by the MythicalMysfitsCoreStack, and the policy document is located at *~/environment/aws-modern-application-workshop/module-2/aws-cli/ecr-policy.json*.

Update and save this file and then run the following command to create the policy:

aws ecr set-repository-policy --repository-name mythicalmysfits/service --policy-text file://~/environment/aws-modern-application-workshop/module-2/aws-cli/ecr-policy.json

**Step 2: Test the CI/CD Pipeline**

##### A: Using Git with AWS CodeCommit

git config --global user.name "REPLACE\_ME\_WITH\_YOUR\_NAME"

git config --global user.email REPLACE\_ME\_WITH\_YOUR\_EMAIL@example.com

git config --global credential.helper '!aws codecommit credential-helper $@'

git config --global credential.UseHttpPath true

Next change directories in your IDE to the environment directory using the terminal:

git clone https://git-codecommit.REPLACE\_REGION.amazonaws.com/v1/repos/MythicalMysfitsService-Repository

This will tell us that our repository is empty! Let's fix that by copying the application files into our repository directory using the following command:

cp -r ~/environment/aws-modern-application-workshop/module-2/app/\* ~/environment/MythicalMysfitsService-Repository/

##### B: Pushing A Code Change

Now the completed service code that we used to create our Fargate service in Module 2 is stored in the local repository that we just cloned from AWS CodeCommit. Let's make a change to the Flask service before committing our changes, to demonstrate that the CI/CD pipeline we've created is working. In Cloud9, open the file stored at *~/environment/MythicalMysfitsService-Repository/service/mysfits-response.json* and change the age of one of the mysfits to another value and save the file.

After saving the file, change directories to the new repository directory:

cd ~/environment/MythicalMysfitsService-Repository/

Then, run the following git commands to push in your code changes.

git add .

git commit -m "I changed the age of one of the mysfits."

git push

After they are pushed in to the repository, you can open the CodePipeline service in the AWS Console to view your changes as they progress through the CI/CD pipeline.

After committing your code change, it will take about 5 to 10 minutes for the changes to be deployed to your live service running in Fargate. Refresh your Mythical Mysfits website in the browser to see that the changes have taken effect.

This concludes Module 2.

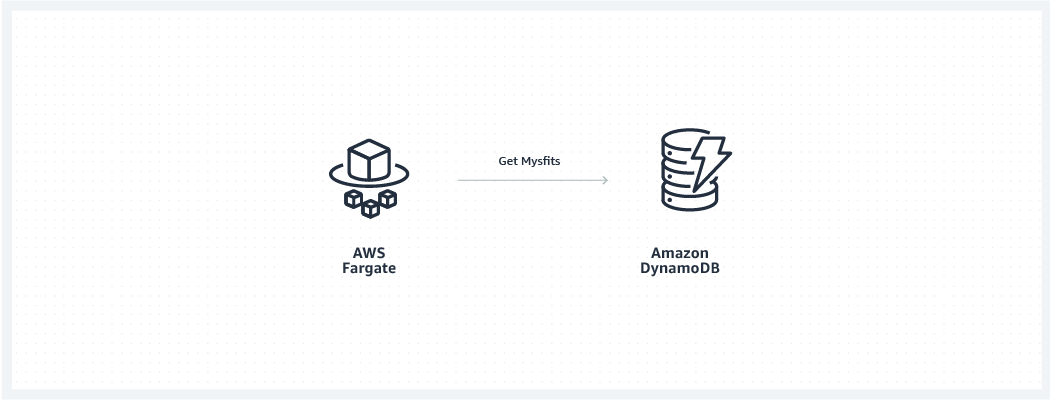
## Module 3: Store Mysfit Information

In this module you will set up Amazon DynamoDB to store the mysfit information in a central database table.

## Overview

Now that you have a service deployed and a working CI/CD pipeline to deliver changes to that service automatically whenever you update your code repository, you can quickly move new application features from conception to available for your Mythical Mysfits customers. With this increased agility, let's add another foundational piece of functionality to the Mythical Mysfits website architecture, a data tier.

## Architecture Diagram



Step 1: Adding A NoSQL Database To Mythical Mysfits

##### A: Create A DynamoDB Table

To add a DynamoDB table to the architecture, we have included another JSON CLI input file that defines a table called MysfitsTable. This table will have a primary index defined by a hash key attribute called MysfitId, and two more secondary indexes. The first secondary index will have the hash key of Species and a range key of MysfitId, and the second secondary index will have the hash key of Alignment and a range key of MysfitId.

These two secondary indexes will allow us to execute queries against the table to retrieve all of the mysfits that match a given Species or Alignment to enable the filter functionality you may have noticed isn't yet working on the website. You can view this file at *~/environment/aws-modern-application-workshop/module-3/aws-cli/dynamodb-table.json*. No changes need to be made to this file and it is ready to execute.

To create the table using the AWS CLI, execute the following command in the Cloud9 terminal:

aws dynamodb create-table --cli-input-json file://~/environment/aws-modern-application-workshop/module-3/aws-cli/dynamodb-table.json

After the command runs, you can view the details of your newly created table by executing the following AWS CLI command in the terminal:

aws dynamodb describe-table --table-name MysfitsTable

If we execute the following command to retrieve all of the items stored in the table, you'll see that the table is empty:

aws dynamodb scan --table-name MysfitsTable

{

"Count": 0,

"Items": [],

"ScannedCount": 0,

"ConsumedCapacity": null

}

* ec2-user:~/environment $ aws dynamodb describe-table --table-name MysfitsTable
* {
* "Table": {
* "TableArn": "arn:aws:dynamodb:us-east-2:795901041651:table/MysfitsTable",
* "AttributeDefinitions": [
* {
* "AttributeName": "GoodEvil",
* "AttributeType": "S"
* },
* {
* "AttributeName": "LawChaos",
* "AttributeType": "S"
* },
* {
* "AttributeName": "MysfitId",
* "AttributeType": "S"
* }
* ],
* "GlobalSecondaryIndexes": [
* {
* "IndexSizeBytes": 0,
* "IndexName": "LawChaosIndex",
* "Projection": {
* "ProjectionType": "ALL"
* },
* "ProvisionedThroughput": {
* "NumberOfDecreasesToday": 0,
* "WriteCapacityUnits": 5,
* "ReadCapacityUnits": 5
* },
* "IndexStatus": "CREATING",
* "KeySchema": [
* {
* "KeyType": "HASH",
* "AttributeName": "LawChaos"
* },
* {
* "KeyType": "RANGE",
* "AttributeName": "MysfitId"
* }
* ],
* "IndexArn": "arn:aws:dynamodb:us-east-2:795901041651:table/MysfitsTable/index/LawChaosIndex",
* "ItemCount": 0
* },
* {
* "IndexSizeBytes": 0,
* "IndexName": "GoodEvilIndex",
* "Projection": {
* "ProjectionType": "ALL"
* },
* "ProvisionedThroughput": {
* "NumberOfDecreasesToday": 0,
* "WriteCapacityUnits": 5,
* "ReadCapacityUnits": 5
* },
* "IndexStatus": "CREATING",
* "KeySchema": [
* {
* "KeyType": "HASH",
* "AttributeName": "GoodEvil"
* },
* {
* "KeyType": "RANGE",
* "AttributeName": "MysfitId"
* }
* ],
* "IndexArn": "arn:aws:dynamodb:us-east-2:795901041651:table/MysfitsTable/index/GoodEvilIndex",
* "ItemCount": 0
* }
* ],
* "ProvisionedThroughput": {
* "NumberOfDecreasesToday": 0,
* "WriteCapacityUnits": 5,
* "ReadCapacityUnits": 5
* },
* "TableSizeBytes": 0,
* "TableName": "MysfitsTable",
* "TableStatus": "CREATING",
* "TableId": "8ad2837b-0175-4ada-807c-67d6e2d775f2",
* "KeySchema": [
* {
* "KeyType": "HASH",
* "AttributeName": "MysfitId"
* }
* ],
* "ItemCount": 0,
* "CreationDateTime": 1538017092.664
* }

}

* ec2-user:~/environment $ aws dynamodb scan --table-name MysfitsTable
* {
* "Count": 0,
* "Items": [],
* "ScannedCount": 0,
* "ConsumedCapacity": null

}

##### B: Add Items To The DynamoDB Table

Also provided is a JSON file that can be used to batch insert a number of Mysfit items into this table. This will be accomplished through the DynamoDB API BatchWriteItem.

To call this API using the provided JSON file, execute the following terminal command (the response from the service should report that there are no items that went unprocessed):

aws dynamodb batch-write-item --request-items file://~/environment/aws-modern-application-workshop/module-3/aws-cli/populate-dynamodb.json

Now, if you run the same command to scan all of the table contents, you'll find the items have been loaded into the table:

aws dynamodb scan --table-name MysfitsTable

**Step 2: Commit Your First Real Code Change**

##### A: Copy The Updated Flask Service Code

Now that we have our data included in the table, let's modify our application code to read from this table instead of returning the static JSON file that was used in Module 2. We have included a new set of Python files for your Flask microservice, but now instead of reading the static JSON file will make a request to DynamoDB.

The request is formed using the AWS Python SDK called boto3. This SDK is a powerful yet simple way to interact with AWS services via Python code. It enables you to use service client definitions and functions that have great symmetry with the AWS APIs and CLI commands you've already been executing as part of this workshop. Translating those commands to working Python code is simple when using boto3.

To copy the new files into your CodeCommit repository directory, execute the following command in the terminal:

cp ~/environment/aws-modern-application-workshop/module-3/app/service/\* ~/environment/MythicalMysfitsService-Repository/service/

##### B: Copy The Updated Flask Service Code

Now, we need to check in these code changes to CodeCommit using the git command line client.

Run the following commands to check in the new code changes and kick of your CI/CD pipeline:

cd ~/environment/MythicalMysfitsService-Repository

git add .

git commit -m "Add new integration to DynamoDB."

git push

Now, in just 5-10 minutes you'll see your code changes make it through your full CI/CD pipeline in CodePipeline and out to your deployed Flask service to AWS Fargate on Amazon ECS.

Feel free to explore the AWS CodePipeline console to see the changes progress through your pipeline.

#### **Step 3: Update The Website Content in S3**

Finally, we need to publish a new index.html page to our S3 bucket so that the new API functionality using query strings to filter responses will be used. The new index.html file is located at ~/environment/aws-modern-application-workshop/module-3/web/index.html.

Open this file in your Cloud9 IDE and replace the string indicating “REPLACE\_ME” just as you did in Module 2, with the appropriate NLB endpoint. Remember do not inlcude the /mysfits path.

Refer to the file you already edited in the /module-2/ directory if you need to. After replacing the endpoint to point at your NLB, upload the new index.html file by running the following command (replacing with the name of the bucket you created in Module 1:

aws s3 cp --recursive ~/environment/aws-modern-application-workshop/module-3/web/ s3://your\_bucket\_name\_here/

Re-visit your Mythical Mysfits website to see the new population of Mysfits loading from your DynamoDB table and how the Filter functionality is working!

That concludes module 3.

aws s3 cp --recursive ~/environment/aws-modern-application-workshop/module-3/web/ s3://mythical-mysfits-meetup-clark/

## Module 4: Setup User Registration

In this module you will setup user registration on the website so you can capture user-specific information.

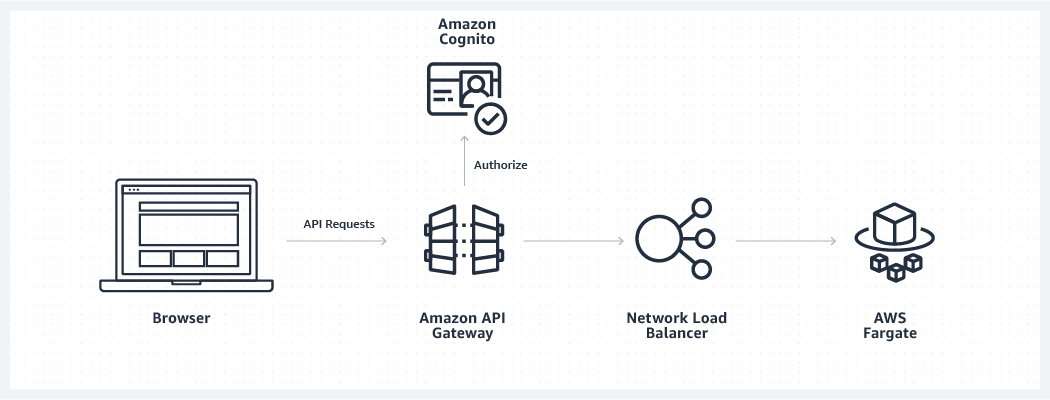
## Overview

In order to add some more critical aspects to the Mythical Mysfits website, like allowing users to vote for their favorite mysfit and adopt a mysfit, we need to first have users register on the website. To enable registration and authentication of website users, we will create a User Pool in AWS Cognito - a fully managed user identity management service.

Then, to make sure that only registered users are authorized to like or adopt mysfits on the website, we will deploy an REST API with Amazon API Gateway to sit in front of our NLB. Amazon API Gateway is also a managed service, and provides commonly required REST API capabilities out of the box like SSL termination, request authorization, throttling, API stages and versioning, and much more.

You will again use the AWS CLI to deploy the needed resources to AWS.

## Architecture Diagram



#### **Time to Complete**

60 minutes

#### **Services Used**

[Amazon Cognito](https://aws.amazon.com/cognito/)  
[Amazon API Gateway](https://aws.amazon.com/api-gateway/)  
[Amazon Simple Storage Service (S3)](https://aws.amazon.com/s3/)

## Implementation Instructions

#### **Step 1: Add A User Pool For Website Users**

##### A: Create The Cognito User Pool

To create the Cognito User Pool where all of the Mythical Mysfits visitors will be stored, execute the following CLI command to create a user pool named MysfitsUserPool and indicate that all users who are registered with this pool should automatically have their email address verified via confirmation email before they become confirmed users.

aws cognito-idp create-user-pool --pool-name MysfitsUserPool --auto-verified-attributes email

Copy the response from the above command, which includes the unique ID for your user pool that you will need to use in later steps. Eg: Id: us-east-1\_ab12345YZ)

##### B: Create A Cognito User Pool Client

Next, in order to integrate our frontend website with Cognito, we must create a new User Pool Client for this user pool. This generates a unique client identifier that will allow our website to be authorized to call the unauthenticated APIs in cognito where website users can sign-in and register against the Mythical Mysfits user pool. To create a new client using the AWS CLI for the above user pool, run the following command (replacing the --user-pool-id value with the one you copied above):

aws cognito-idp create-user-pool-client --user-pool-id REPLACE\_ME --client-name MysfitsUserPoolClient

aws cognito-idp create-user-pool-client --user-pool-id REPLACE\_ME --client-name MysfitsUserPoolClient

aws cognito-idp create-user-pool-client --user-pool-id us-east-2\_9LbAusDyl --client-name MysfitsUserPoolClient

**Step 2: Add A User Pool For Website Users**

##### A: Create An API Gateway VPC Link

Next, let's turn our attention to creating a new RESTful API in front of our existing Flask service, so that we can perform request authorization before our NLB receives any requests. We will do this with Amazon API Gateway, as described in the module overview.

In order for API Gateway to privately integrate with our NLB, we will configure an API Gateway VPC Link that enables API Gateway APIs to directly integrate with backend web services that are privately hosted inside a VPC. Note: For the purposes of this workshop, we created the NLB to be internet-facing so that it could be called directly in earlier modules. Because of this, even though we will be requiring Authorization tokens in our API after this module, our NLB will still actually be open to the public behind the API Gateway API.

In a real-world scenario, you should create your NLB to be internal from the beginning (or create a new internal load balancer to replace the existing one), knowing that API Gateway would be your strategy for Internet-facing API authorization. But for the sake of time, we'll use the NLB that we've already created that will stay publicly accessible.

Create the VPC Link for our upcoming REST API using the following CLI command (you will need to replace the indicated value with the Load Balancer ARN you saved when the NLB was created in module 2):

aws apigateway create-vpc-link --name MysfitsApiVpcLink --target-arns REPLACE\_ME\_NLB\_ARN

aws apigateway create-vpc-link --name MysfitsApiVpcLink --target-arns arn:aws:elasticloadbalancing:us-east-2:795901041651:loadbalancer/net/mysfits-nlb/30d50cb4673bfe36

**LOOK AT AWS CONSOLE**

The response to the above will indicate that a new VPC link is being provisioned and is in PENDING state.

Copy the indicated id for future use when we create our REST API in the next step.

{

"status": "PENDING",

"targetArns": [

"YOUR\_ARN\_HERE"

],

"id": "abcdef1",

"name": "MysfitsApiVpcLink"

}

With the VPC link creating, we can move on to create the actual REST API using Amazon API Gateway.

##### B: Create The REST API Using Swagger

Your MythicalMysfits REST API is defined using \*\*Swagger\*\*, a popular open-source framework for describing APIs via JSON. This Swagger definition of the API is located at `~/environment/aws-modern-applicaiton-workshop/module-4/aws-cli/api-swagger.json`. Open this file and you'll see the REST API and all of its resources, methods, and configuration defined within.

There are several places within this JSON file that need to be updated to include parameters specific to your Cognito User Pool, as well as your Network Load Balancer.

The `securityDefinitions` object within the API definition indicates that we have setup an apiKey authorization mechanism using the Authorization header. You will notice that AWS has provided custom extensions to Swagger using the prefix `x-amazon-api-gateway-`, these extensions are where API Gateway specific functionality can be added to typical swagger files to take advantage of API Gateway-specific capabilities.

CTRL-F through the file to search for the various places `REPLACE\_ME` is located and awaiting your specific parameters. Once the edits have been made, save the file and execute the following AWS CLI command:

{

"status": "PENDING",

"targetArns": [

"arn:aws:elasticloadbalancing:us-east-2:795901041651:loadbalancer/net/mysfits-nlb/30d50cb4673bfe36"

],

"id": "k94y0s",

"name": "MysfitsApiVpcLink"

}

aws apigateway import-rest-api --parameters endpointConfigurationTypes=REGIONAL --body file://~/environment/aws-modern-application-workshop/module-4/aws-cli/api-swagger.json --fail-on-warnings

Copy the response this command returns and save the `id` value for the next step:

{

"name": "MysfitsApi",

"endpointConfiguration": {

"types": [

"REGIONAL"

]

},

"id": "abcde12345",

"createdDate": 1529613528

}

{

"apiKeySource": "HEADER",

"name": "MysfitsApi",

"createdDate": 1538018312,

"endpointConfiguration": {

"types": [

"REGIONAL"

]

},

"id": "uuyf5snxo6"

}

##### C: Deploy The API

Now, our API has been created, but it's yet to be deployed anywhere. To deploy our API, we must first create a deployment and indicate which \*\*stage\*\* the deployment is fore. A stage is a named reference to a deployment, which is a snapshot of the API.

You use a Stage to manage and optimize a particular deployment. For example, you can set up stage settings to enable caching, customize request throttling, configure logging, define stage variables or attach a canary release for testing. We will call our stage `prod`. To create a deployment for the prod stage, execute the following CLI command:

aws apigateway create-deployment --rest-api-id REPLACE\_ME\_WITH\_API\_ID --stage-name prod

aws apigateway create-deployment --rest-api-id uuyf5snxo6 --stage-name prod

{

"id": "nc8nnh",

"createdDate": 1538018639

}

With that, our REST API that's capable of user Authorization is deployed and available on the Internet... but where?! Your API is available at the following location:

https://REPLACE\_ME\_WITH\_API\_ID.execute-api.REPLACE\_ME\_WITH\_REGION.amazonaws.com/prod

https://uuyf5snxo6.execute-api.us-east-2.amazonaws.com/prod

Copy the above, replacing the appropriate values, and add `/mysfits` to the end of the URI. Entered into a browser address bar, you should once again see your Mysfits JSON response. But, we've added several capabilities like adopting and liking mysfits that our Flask service backend doesn't have implemented yet.

Let's take care of that next.

While those service updates are being automatically pushed through your CI/CD pipeline, continue on to the next step.

**Step 3: Update the Mythical Mysfits Website**

##### A: Update the Flask Service Backend

To accommodate the new functionality to view Mysfit Profiles, like, and adopt them, we have included updated Python code for your backend Flask web service.

Let's overwrite your existing codebase with these files and push them into the repository:

cd ~/environment/MythicalMysfitsService-Repository/

cp -r ~/environment/aws-modern-application-workshop/module-4/app/\* .

git add .

git commit -m "Update service code backend to enable additional website features."

git push

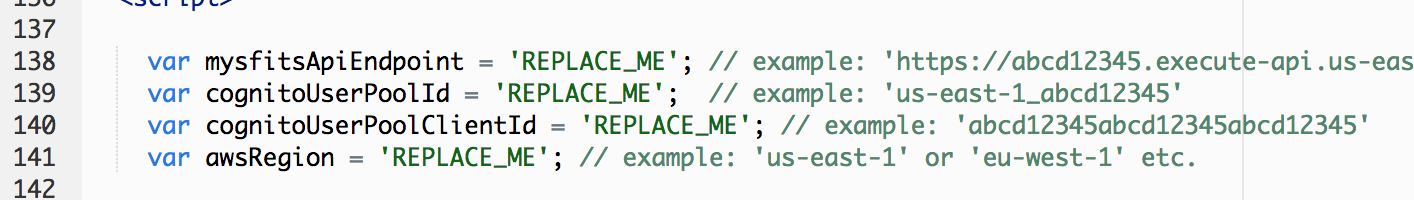
While those service updates are being automatically pushed through your CI/CD pipeline, continue on to the next step.

##### B: Update The Mythical Mysfits Website In S3

Open the new version of the Mythical Mysfits index.html file we will push to S3 shortly, it is located at: *~/environment/aws-modern-application-workshop/module-4/app/web/index.html* In this new index.html file, you'll notice additional HTML and JavaScript code that is being used to add a user registration and login experience.

This code is interacting with the AWS Cognito JavaScript SDK to help manage registration, authentication, and authorization to all of the API calls that require it.

In this file, replace the strings REPLACE\_ME inside the single quotes with the OutputValues you copied from above and save the file:



Also, for the user registration process, you have an additional two HTML files to insert these values into. register.html and confirm.html. Insert the copied values into the REPLACE\_ME strings in these files as well.

Now, lets copy these HTML files, as well as the Cognito JavaScript SDK to the S3 bucket hosting our Mythical Mysfits website content so that the new features will be published online.

aws s3 cp --recursive ~/environment/aws-modern-application-workshop/module-4/web/ s3://YOUR-S3-BUCKET/

aws s3 cp --recursive ~/environment/aws-modern-application-workshop/module-4/web/ s3://mythical-mysfits-meetup-clark/

Refresh the Mythical Mysfits website in your browser to see the new functionality in action!

This concludes Module 4.

Module 5: Capture User Behavior

In this module you will capture user behavior using AWS Lambda and other serverless services.

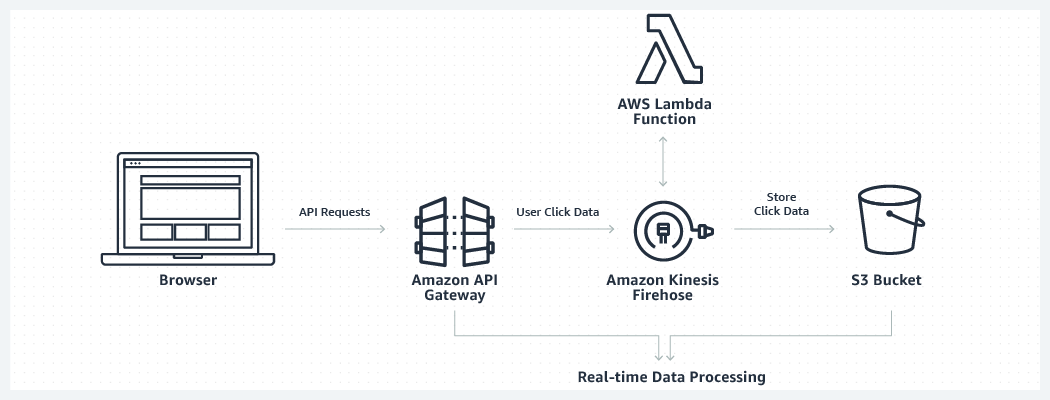
Overview

Now that your Mythical Mysfits site is up and running, let's create a way to better understand how users are interacting with the website and its Mysfits. It would be very easy for us to analyze user actions taken on the website that lead to data changes in our backend - when mysfits are adopted or liked.

But understanding the actions your users are taking on the website before a decision to like or adopt a mysfit could help you design a better user experience in the future that leads to mysfits getting adopted even faster. To help us gather these insights, we will implement the ability for the website frontend to submit a tiny request, each time a mysfit profile is clicked by a user, to a new microservice API we'll create. Those records will be processed in real-time by a serverless code function, aggregated, and stored for any future analysis that you may want to perform.

Modern application design principles prefer focused, decoupled, and modular services. So rather than add additional methods and capabilities within the existing Mysfits service that you have been working with so far, we will create a new and decoupled service for the purpose of receiving user click events from the Mysfits website. This full stack has been represented using a provided CloudFormation template.

Architecture Diagram



 Time to Complete

30 minutes

 Services Used

[AWS CloudFormation](https://aws.amazon.com/cloudformation/)  
[AWS Kinesis Firehose](https://aws.amazon.com/kinesis/)  
[Amazon S3](https://aws.amazon.com/s3/)  
[Amazon API Gateway](https://aws.amazon.com/api-gateway/)  
[AWS Lambda](https://aws.amazon.com/lambda/)  
[AWS CodeCommit](https://aws.amazon.com/codecommit/)  
[AWS Serverless Application Model (AWS SAM)](https://docs.aws.amazon.com/lambda/latest/dg/serverless_app.html)  
[AWS SAM Command Line Interface (SAM CLI)](https://docs.aws.amazon.com/lambda/latest/dg/sam-cli-requirements.html)

Why Choose AWS Lambda For This Implementation?

Lambda is great for data-driven applications that need to respond in real-time to changes in data, shifts in system state, or actions by users. These applications usually connect to data stores to access and analyze the data for batch processing, stream analytics, and machine learning inference. Lambda is well-suited for these applications because it is integrated with data stores such as Kinesis Data Streams and Data Firehose, S3, CloudWatch Events, CloudWatch Logs, and DynamoDB, with a total of 17 event sources today.

Serverless Resources Used in This Module

 AWS Kinesis Firehose delivery stream: Kinesis Firehose is a managed real-time streaming service that accepts data records and automatically ingests them into several possible storage destinations within AWS, such as Amazon S3 bucket, or Amazon Redshift data warehouse cluster. Kinesis Firehose also enables all of the records received by the stream to be automatically delivered to a serverless function created with AWS Lambda This means that code you've written can perform any additional processing or transformations of the records before they are aggregated and stored in the configured destination.

Amazon S3 bucket: A new bucket will be created in S3 where all of the processed click event records are aggregated into files and stored as objects.

AWS Lambda function: We've laready discussed AWS lambda above. Here, a Serverless code function is defined using AWS SAM. It will be deployed to AWS Lambda, written in Python, and then process and enrich the click records that are received by the delivery stream. The code we've written is very simple and the enriching it does could have been accomplished on the website frontend without any subsequent processing at all. The function retrieves additional attributes about the clicks on Mysfit to make the click record more meaningful (data that was already retrieved by the website frontend). But, for the purpose of this workshop, the code is meant to demonstrate the architectural possibilities of including a serverless code function to perform any additional processing or transformation required, in real-time, before records are stored. Once the Lambda function is created and the Kinesis Firehose delivery stream is configured as an event source for the function, the delivery stream will automatically deliver click records as events to code function we've created, receive the responses that our code returns, and deliver the updated records to the configured Amazon S3 bucket.

An Amazon API Gateway REST API: AWS Kinesis Firehose provides a service API just like other AWS services, and in this case we are using its PutRecord operation to put user click event records into the delivery stream. But, we don't want our website frontend to have to directly integrate with the Kinesis Firehose PutRecord API. Doing so would require us to manage AWS credentials within our frontend code to authorize those API requests to the PutRecord API, and it would expose to users the direct AWS API that is being depended on (which may encourage malicious site visitors to attempt to add records to the delivery stream that are malformed, or harmful to our goal of understanding real user behavior). So instead, we will use Amazon API Gateway to create an AWS Service Proxy to the PutRecord API of Kinesis Firehose. This allows us to craft our own public RESTful endpoint that does not require AWS credential management on the frontend for requests. Also, we will use a request mapping template in API Gateway as well, which will let us define our own request payload structure that will restrict requests to our expected structure and then transform those well-formed requests into the structure that the Kinesis Firehose PutRecord API requires.

IAM Roles: Kinesis Firehose requires a service role that allows it to deliver received records as events to the created Lambda function as well as the processed records to the destination S3 bucket. The Amazon API Gateway API also requires a new role that permits the API to invoke the PutRecord API within Kinesis Firehose for each received API request.

## Implementation Instructions

#### **Step 1: Create The Streaming Service Code**

##### A: Create A New CodeCommit Repository

This new stack you will deploy using CloudFormation will not only contain the infrastructure environment resources, but the application code itself that AWS Lambda will execute to process streaming events. To bundle the creation of our infrastructure and code together in one deployment, we are going to use another AWS tool that comes pre-installed in the AWS Cloud9 IDE - the AWS SAM CLI. Code for AWS Lambda functions is delivered to the service by uploading the function code in a .zip package to an Amazon S3 bucket.

The SAM CLI automates that process for us. Using it, we can create a CloudFormation template that references locally in the filesystem where all of the code for our Lambda function is stored. Then, the SAM CLI will package it into a .zip file, upload it to a configured Amazon S3 bucket, and create a new CloudFormation template that indicates the location in S3 where the created .zip package has been uploaded for deployment to AWS Lambda. We can then deploy that SAM CLI-generated CloudFormation template to AWS and watch the environment be created along with the Lambda function that uses the SAM CLI-uploaded code package.

First, let's create a new CodeCommit repository where the streaming service code will live:

aws codecommit create-repository --repository-name MythicalMysfitsStreamingService-Repository

In the response to that command, copy the value for "cloneUrlHttp". It should be of the form: https://git-codecommit.REPLACE\_ME\_REGION.amazonaws.com/v1/repos/MythicalMysfitsStreamingService-Repository

Next, let's clone that new and empty repository into our IDE:

cd ~/environment/

git clone {insert the copied cloneValueUrl from above}

{

"repositoryMetadata": {

"repositoryName": "MythicalMysfitsStreamingService-Repository",

"cloneUrlSsh": "ssh://git-codecommit.us-east-2.amazonaws.com/v1/repos/MythicalMysfitsStreamingService-Repository",

"lastModifiedDate": 1538019001.875,

"repositoryId": "c5de6bed-541c-4932-85c3-b918f4d5cf89",

"cloneUrlHttp": "https://git-codecommit.us-east-2.amazonaws.com/v1/repos/MythicalMysfitsStreamingService-Repository",

"creationDate": 1538019001.875,

"Arn": "arn:aws:codecommit:us-east-2:795901041651:MythicalMysfitsStreamingService-Repository",

"accountId": "795901041651"

}

}

##### B: Copy The Streaming Service Code Base

Now, let's move our working directory into this new repository:

cd ~/environment/MythicalMysfitsStreamingService-Repository/

Then, copy the module-5 application components into this new repository directory:

cp -r ~/environment/aws-modern-application-workshop/module-5/app/streaming/\* .

And let's copy the CloudFormation template for this module as well.

cp ~/environment/aws-modern-application-workshop/module-5/cfn/\* .

**Step2: Update The Lambda Function Package And Code**

Now, we have the repository directory set with all of the provided artifacts:

A CFN template for creating the full stack.

A Python script that contains the code for our Lambda function: streamProcessor.py

This is a common approach that AWS customers take - to store their CloudFormation templates alongside their application code in a repository. That way, you have a single place where all changes to application and it's environment can be tracked together.

But, if you look at the code inside the streamProcessor.py file, you'll notice that it's using the requests Python package to make an API requset to the Mythical Mysfits service you created previously. External libraries are not automatically included in the AWS Lambda runtime environment, because different AWS customers may depend on different versions of various libraries, etc.

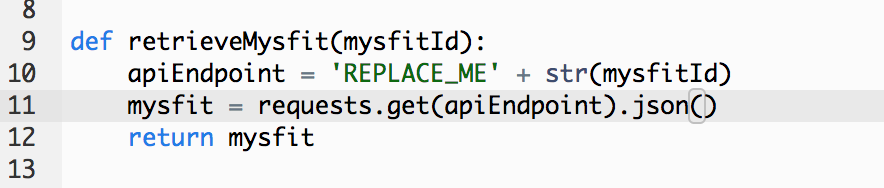
You will need to package all of your library dependencies together with your Lambda code function prior to it being uploaded to the Lambda service. We will use the Python package manager pip to accomplish this. In the Cloud9 terminal, run the following command to install the requests package and it's dependencies locally alongside your function code:

pip install requests -t .

Once this command completes, you will see several additional python package folders stored within your repository directory.

##### B: Update The Lambda Function Code

Next, we have one code change to make prior to our Lambda function code being completely ready for deployment. There is a line within the streamProcessor.py file that needs to be replaced with the ApiEndpoint for your Mysfits service API - the same service ApiEndpoint that you created in module-4 and used on the website frontend.



That service is responsible for integrating with the MysfitsTable in DynamoDB, so even though we could write a Lambda function that directly integrated with the DynamoDB table as well, doing so would intrude upon the purpose of the first microservice and leave us with multiple/separate code bases that integrated with the same table. Instead, we will integrate with that table through the existing service and have a much more decoupled and modular application architecture.

##### C: Push Your Code Into CodeCommit

Let's commit our code changes to the new repository so that they're saved in CodeCommit:

git add .

git commit -m "New stream processing service."

git push

**Step 3: Create the Streaming Service Stack**

##### A: Create An S3 Bucket For Lambda Function Code Packages

With that line changed in the Python file, and our code committed, we are ready to use the AWS SAM CLI to package all of our function code, upload it to S3, and create the deployable CloudFormation template to create our streaming stack.

First, use the AWS CLI to create a new S3 bucket where our Lambda function code packages will be uploaded to. S3 bucket names need to be globally unique among all AWS customers, so replace the end of this bucket name with a string that's unique to you:

##### B: Use The SAM CLI To Package Your Code For Lambda

With our bucket created, we are ready to use the SAM CLI to package and upload our code and transform the CloudFormation template, be sure to replace the last command parameter with the bucket name you just created above (this command also assumes your terminal is still in the repository working directory):

sam package --template-file ./real-time-streaming.yml --output-template-file ./transformed-streaming.yml --s3-bucket replace-with-your-bucket-name

sam package --template-file ./real-time-streaming.yml --output-template-file ./transformed-streaming.yml --s3-bucket mythical-mysfits-meetup-clark

If successful, you will see the newly created transformed-streaming.yml file exist within the ./cfn/ directory, if you look in its contents, you'll see that the sourceUri parameter of the serverless Lambda function has been updated with the object location where the SAM CLI has uploaded your packaged code.

##### C: Deploy The Stack Using AWS CloudFormation

Also returned by the SAM CLI command is the CloudFormation command needed to be executed to create our new full stack. But because our stack creates IAM resources, you'll need to add one additional parameter to the command. Execute the following command to deploy the streaming stack:

aws cloudformation deploy --template-file /home/ec2-user/environment/MythicalMysfitsStreamingService-Repository/cfn/transformed-streaming.yml --stack-name MythicalMysfitsStreamingStack --capabilities CAPABILITY\_IAM

Once this stack creation is complete, the full real-time processing microservice will be created.

In future scenarios where only code changes have been made to your Lambda function, and the rest of your CloudFormation stack remains unchanged, you can repeat the same AWS SAM CLI and CloudFormation commands as above. This will result in the infrastructure environment remaining unchanged, but a code deployment occurring to your Lambda function.

Step 4 Sending Mysfit Profile Clicks to Our New Microservice

##### A: Update The Website Content

With the streaming stack up and running, we now need to publish a new version of our Mythical Mysfits frontend that includes the JavaScript that sends events to our service whenever a mysfit profile is clicked by a user.

The new index.html file is included at: *~/environment/aws-modern-application-workshop/module-5/app/web/index.html*

This file contains the same placeholders as module-4 that need to be updated, as well as an additional placeholder for the new stream processing service endpoint you just created. For the previous variable values, you can refer to the previous index.html file you updated as part of module-4.

Perform the following command for the new streaming stack to retrieve the new API Gateway endpoint for your stream processing service:

aws cloudformation describe-stacks --stack-name MythicalMysfitsStreamingStack

##### B: Push The New Site Version to S3

Replace the final value within *index.html* for the streamingApiEndpoint and you are ready to publish your final Mythical Mysfits home page update:

aws s3 cp ~/environment/aws-modern-application-workshop/module-5/web/index.html s3://YOUR-S3-BUCKET/

Refresh your Mythical Mysfits website in the browser once more and you will now have a site that records and publishes each time a user clicks on a mysfits profile!

To view the records that have been processed, they will arrive in the destination S3 bucket created as part of your MythicalMysfitsStreamingStack.

Now that you have a completed modern application architecture, we encourage you now to explore the AWS Console and all the various services you've created to launch Mythical Mysfits!

Step 5: Workshop clean up

* Be sure to delete all the resources created during the workshop in order to ensure that billing for the resources does not continue for longer than you intend. We recommend that you utilize the AWS Console to explore the resources you've created and delete them when you're ready.

For the two cases where you provisioned resources using AWS CloudFormation, you can remove those resources by simply running the following CLI command for each stack:

aws cloudformation delete-stack --stack-name STACK-NAME-HERE

To remove all of the created resources, you can visit the following AWS Consoles, which contain resources you've created during the Mythical Mysfits workshop:

[AWS Kinesis](https://console.aws.amazon.com/kinesis)[AWS Lambda](https://console.aws.amazon.com/lambda/home)  
[Amazon S3](https://s3.console.aws.amazon.com/)  
[Amazon API Gateway](https://console.aws.amazon.com/apigateway)  
[Amazon Cognito](https://console.aws.amazon.com/cognito)

[AWS CodePipeline](https://console.aws.amazon.com/codepipeline)  
[AWS CodeBuild](https://console.aws.amazon.com/codebuild)  
[AWS CodeCommit](https://console.aws.amazon.com/codecommit)  
[Amazon DynamoDB](https://console.aws.amazon.com/dynamodb)  
[Amazon ECS](https://console.aws.amazon.com/ecs/)

[Amazon EC2](https://console.aws.amazon.com/ec2)  
[Amazon VPC](https://console.aws.amazon.com/vpc)  
[AWS IAM](https://console.aws.amazon.com/iam)  
[AWS CloudFormation](https://console.aws.amazon.com/cloudformation)

## Conclusion

This experience was meant to give you a taste of what it's like to be a developer designing and building modern application architectures on top of AWS. Developers on AWS are able to programmatically provision resources using the AWS CLI, reuse infrastructure definitions via AWS CloudFormation, automatically build and deploy code changes using the AWS developer tool suite of Code services, and take advantage of multiple different compute and application service capabilities that do not require you to provision or manage any servers at all!

As a great next step, to learn more about the inner workings of the Mythical Mysfits website that you've created, dive into the provided CloudFormation templates and the resources declared within them.

We hope you have enjoyed the AWS Modern Application Workshop! This tutorial is also hosted on [GitHub](https://github.com/aws-samples/aws-modern-application-workshop) so you will be able to submit an issue if you have recommendations. You’ll also be able to initialize GitHub pull request if you want to collaborate on improving the code.

If you want to learn more about developing on AWS, visit our [AWS Developer Center.](https://aws.amazon.com/developer/)