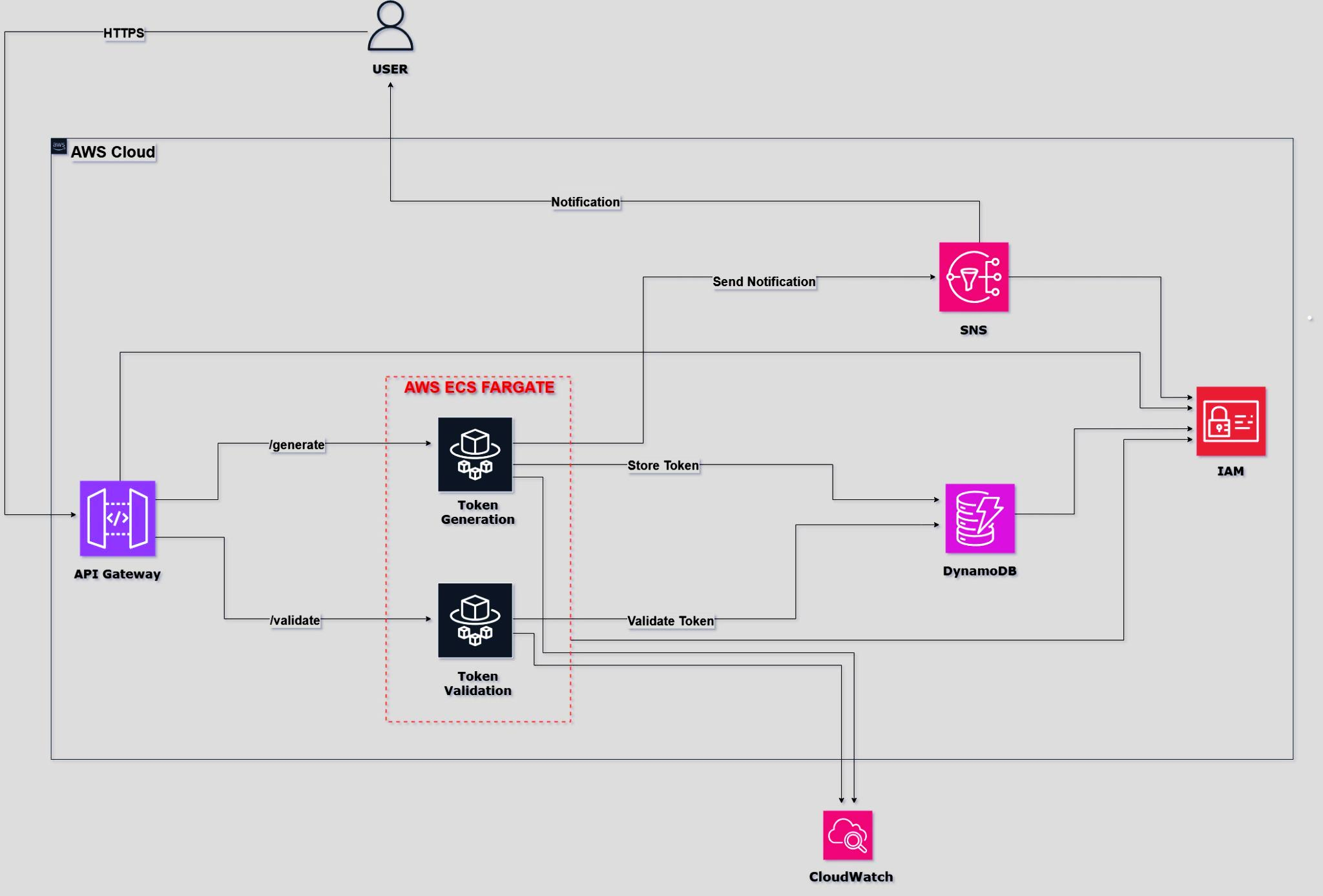
**2FA Token Solution on AWS**

**2FA Token Architecture Overview:**

* AWS ECS Fatgate for generating and validating tokens.
* Amazon DynamoDB for storing tokens.
* Amazon SNS sends tokens to users via Notification.
* Amazon API Gateway provides RESTful endpoints for users to request or validate tokens.
* AWS IAM manages access and ensures system security.

**Github:** <https://github.com/Mabdullahofficial/KR91-2FA-Cybernetic-Controls-Limited-Asseessment.git>

**2FA Token Architecture Diagram**



**Detail of the Components**

1. **API Gateway:**
   * Exposes two endpoints: /generate-token and /validate-token
   * Integrated with ECS Fargate
2. **AWS ECS Fargate:** a. Token Generation Service:
   * Generates a token
   * Stores token in DynamoDB with expiration time
   * Sends token via SNS

b. Token Validation Service:

* + Receives token from user
  + Checks token validity in DynamoDB
  + Deletes token if valid (single-use)

1. **DynamoDB:**
   * Store validation token in table.
2. **SNS:**
   * Used to send SNS Notifications with tokens
3. **IAM** 
   * Manages permissions for secure access.
4. **CloudWatch**
   * Monitors the application and logs critical events like token generation, validation requests, and service health.

**Code Implementation Concept**

#### **Token Generation Service on ECS Fargate**

* **Token-generation.py (Token Generation Service)**

| **from** flask **import** Flask, request, jsonify  **import** boto3  **import** jwt  **import** os  **import** datetime   app = Flask(\_\_name\_\_)  dynamodb = boto3.resource('dynamodb')  table = dynamodb.Table('Tokens')   **@app.route('/generate\_token', methods=['POST'])**  **def** **generate\_token**():   *# Check if request JSON has required keys*  **if** **not** request.json **or** 'user\_id' **not** **in** request.json **or** 'phone\_number' **not** **in** request.json:  **return** jsonify({'status': 'error', 'message': 'Missing user\_id or phone\_number'}), 400   user\_id = request.json['user\_id']   phone\_number = request.json['phone\_number']    *# Generate token*  expiration\_time = datetime.datetime.utcnow() + datetime.timedelta(minutes=5)  token = jwt.encode({'user\_id': user\_id, 'exp': expiration\_time}, os.environ['SECRET\_KEY'], algorithm='HS256')    *# Store token in DynamoDB*  **try**:  table.put\_item(Item={'token': token, 'user\_id': user\_id, 'status': 'unused', 'expiration': expiration\_time.timestamp()})  **except** Exception **as** e:  **return** jsonify({'status': 'error', 'message': str(e)}), 500   *# Send token via SNS*  sns = boto3.client('sns')   **try**:  sns.publish(PhoneNumber=phone\_number, Message=f'Your token is: {token}')   **except** Exception **as** e:  **return** jsonify({'status': 'error', 'message': str(e)}), 500   **return** jsonify({'status': 'success', 'message': 'Token sent'})   **if** \_\_name\_\_ == '\_\_main\_\_':   app.run(host='0.0.0.0', port=5000) |
| --- |

**Dockerfile of Token Generation:**

| *# Use a specific version of Python for consistency* **FROM** python:3.9.12-slim   *# Set the working directory* **WORKDIR** /app   *# Copy the requirements file first for better caching* **COPY** requirements.txt .   *# Install dependencies* **RUN** pip install --no-cache-dir -r requirements.txt   *# Copy the rest of your application code* **COPY** . .   *# Expose the port the app runs on* **EXPOSE** 5000  *# Set environment variables (optional, can also be set at runtime)* *# ENV SECRET\_KEY=your\_secret\_key*  *# Command to run the application* **CMD** ["python", "app.py"] |
| --- |

#### **Token Validation Service on ECS Fargate**

* **Token-validation.py (Token Validation Service)**

| **from** flask **import** Flask, request, jsonify  **import** boto3  **import** jwt  **import** os  **import** logging   *# Set up logging* logging.basicConfig(level=logging.INFO)  app = Flask(\_\_name\_\_)  dynamodb = boto3.resource('dynamodb')  table = dynamodb.Table('Tokens')   **@app.route('/validate\_token', methods=['POST'])**  **def** **validate\_token**():   *# Check if the request has a JSON body and contains the token*  **if** **not** request.json **or** 'token' **not** **in** request.json:  logging.error('Missing token in request')  **return** jsonify({'status': 'fail', 'message': 'Missing token'}), 400   token = request.json['token']    **try**:   *# Decode and validate the token*   decoded = jwt.decode(token, os.environ['SECRET\_KEY'], algorithms=['HS256'])     *# Check token status in DynamoDB*   response = table.get\_item(Key={'token': token})   item = response.get('Item')     **if** item **and** item['status'] == 'unused':   *# Mark token as used*   table.update\_item(   Key={'token': token},   UpdateExpression="set #status = :s",   ExpressionAttributeNames={'#status': 'status'},   ExpressionAttributeValues={':s': 'used'}   )   **return** jsonify({'status': 'success', 'message': 'Token is valid'}), 200     **return** jsonify({'status': 'fail', 'message': 'Token is invalid or already used'}), 401     **except** jwt.ExpiredSignatureError:   logging.error('Token has expired')  **return** jsonify({'status': 'fail', 'message': 'Token has expired'}), 401   **except** jwt.InvalidTokenError:   logging.error('Invalid token')  **return** jsonify({'status': 'fail', 'message': 'Token is invalid'}), 401   **except** Exception **as** e:   logging.error(f'An error occurred: {e}')  **return** jsonify({'status': 'fail', 'message': 'An error occurred during validation'}), 500   **if** \_\_name\_\_ == '\_\_main\_\_':   app.run(host='0.0.0.0', port=5001) |
| --- |

* **Dockerfile of Token Validation**

| *# Use a specific version of Python for consistency* **FROM** python:3.9.12-slim   *# Set the working directory* **WORKDIR** /app   *# Copy the requirements file first to leverage Docker's caching mechanism* **COPY** requirements.txt .   *# Install dependencies* **RUN** pip install --no-cache-dir -r requirements.txt   *# Copy the rest of your application code* **COPY** . .   *# Expose the port the app runs on* **EXPOSE** 5001   *# Command to run the application* **CMD** ["python", "app.py"] |
| --- |

### **Deployment Steps**

### **ECS Setup**

I defined two ECS Fargate services in the cluster, one for token generation and one for token validation.

### **Task Definitions**

I created task definitions for each service, specifying Docker images and resource requirements (CPU, memory).

### **Deploy and Test**

I deployed the services and ensured they could communicate with necessary AWS resources like DynamoDB and SNS, testing their functionality.

**CloudFormation Template**

| AWSTemplateFormatVersion: '2010-09-09' Description: CloudFormation Template for 2FA Token Solution on AWS  Parameters:  VPC:  Type: AWS::EC2::VPC::Id  Description: Select the VPC for ECS services  Subnets:  Type: List<AWS::EC2::Subnet::Id>  Description: Select the subnets for ECS tasks  SecretKey:  Type: String  NoEcho: true  Description: Secret key for JWT token generation  Resources:  *# IAM Roles*  ECSFargateExecutionRole:  Type: AWS::IAM::Role  Properties:  AssumeRolePolicyDocument:  Statement:  - Effect: Allow  Principal:  Service: ecs-tasks.amazonaws.com  Action: sts:AssumeRole  Policies:  - PolicyName: ECSFargateTaskPolicy  PolicyDocument:  Statement:  - Effect: Allow  Action:  - dynamodb:\*  - sns:Publish  - logs:CreateLogGroup  - logs:CreateLogStream  - logs:PutLogEvents  Resource: "\*"   *# DynamoDB Table*  TokenTable:  Type: AWS::DynamoDB::Table  Properties:  TableName: Tokens  AttributeDefinitions:  - AttributeName: token  AttributeType: S  KeySchema:  - AttributeName: token  KeyType: HASH  ProvisionedThroughput:  ReadCapacityUnits: 5  WriteCapacityUnits: 5   *# SNS Topic*  SNSTopic:  Type: AWS::SNS::Topic  Properties:  DisplayName: 2FA Token Notifications   *# ECS Cluster*  ECSCluster:  Type: AWS::ECS::Cluster  Properties:  ClusterName: 2FATokenCluster   *# ECS Task Definitions*  TokenGenerationTask:  Type: AWS::ECS::TaskDefinition  Properties:  Family: TokenGenerationService  NetworkMode: awsvpc  RequiresCompatibilities:  - FARGATE  Cpu: 256  Memory: 512  ExecutionRoleArn: !GetAtt ECSFargateExecutionRole.Arn  ContainerDefinitions:  - Name: TokenGenerationService  Image: your-docker-image-url/token-generation-service:latest  PortMappings:  - ContainerPort: 5000  Environment:  - Name: SECRET\_KEY  Value: !Ref SecretKey  - Name: DYNAMODB\_TABLE  Value: Tokens  - Name: SNS\_TOPIC  Value: !Ref SNSTopic   TokenValidationTask:  Type: AWS::ECS::TaskDefinition  Properties:  Family: TokenValidationService  NetworkMode: awsvpc  RequiresCompatibilities:  - FARGATE  Cpu: 256  Memory: 512  ExecutionRoleArn: !GetAtt ECSFargateExecutionRole.Arn  ContainerDefinitions:  - Name: TokenValidationService  Image: your-docker-image-url/token-validation-service:latest  PortMappings:  - ContainerPort: 5001  Environment:  - Name: SECRET\_KEY  Value: !Ref SecretKey  - Name: DYNAMODB\_TABLE  Value: Tokens   *# ECS Services*  TokenGenerationService:  Type: AWS::ECS::Service  Properties:  Cluster: !Ref ECSCluster  DesiredCount: 1  LaunchType: FARGATE  TaskDefinition: !Ref TokenGenerationTask  NetworkConfiguration:  AwsvpcConfiguration:  Subnets: !Ref Subnets  AssignPublicIp: ENABLED   TokenValidationService:  Type: AWS::ECS::Service  Properties:  Cluster: !Ref ECSCluster  DesiredCount: 1  LaunchType: FARGATE  TaskDefinition: !Ref TokenValidationTask  NetworkConfiguration:  AwsvpcConfiguration:  Subnets: !Ref Subnets  AssignPublicIp: ENABLED   *# API Gateway*  APIGateway:  Type: AWS::ApiGateway::RestApi  Properties:  Name: 2FA-API   GenerateTokenResource:  Type: AWS::ApiGateway::Resource  Properties:  RestApiId: !Ref APIGateway  ParentId: !GetAtt APIGateway.RootResourceId  PathPart: generate-token   ValidateTokenResource:  Type: AWS::ApiGateway::Resource  Properties:  RestApiId: !Ref APIGateway  ParentId: !GetAtt APIGateway.RootResourceId  PathPart: validate-token   GenerateTokenMethod:  Type: AWS::ApiGateway::Method  Properties:  RestApiId: !Ref APIGateway  ResourceId: !Ref GenerateTokenResource  HttpMethod: POST  AuthorizationType: NONE  Integration:  Type: HTTP  Uri: !Sub "http://${TokenGenerationService.Arn}:5000/generate-token"  IntegrationHttpMethod: POST   ValidateTokenMethod:  Type: AWS::ApiGateway::Method  Properties:  RestApiId: !Ref APIGateway  ResourceId: !Ref ValidateTokenResource  HttpMethod: POST  AuthorizationType: NONEIntegration:  Type: HTTP  Uri: !Sub "http://${TokenValidationService.Arn}:5001/validate-token"  IntegrationHttpMethod: POST  Outputs:  APIGatewayURL:  Description: URL for the 2FA API  Value: !Sub "https://${APIGateway}.execute-api.${AWS::Region}.amazonaws.com/prod" |
| --- |

**Security Best Practices**

* Apply least privilege IAM roles for ECS Fargate.
* Enable DynamoDB encryption at rest.
* Use HTTPS for API Gateway.
* Implement API Gateway rate limiting.
* Protect against common web exploits with AWS WAF.

**Health/Performance Metrics**

1. Monitor Architecture using CloudWatch:

* ECS Fargate Task Invocation Count & Duration
* API Gateway Request Count & Latency
* DynamoDB Read/Write Capacity Utilization
* SNS Message Delivery Rate

1. AWS X-Ray for Cross-Service Request Tracing

**Resilience & Modularity**

* Deploy DynamoDB Across Multiple AZs for improved availability.
* Implement Retry Logic in ECS Fargate for enhanced reliability.
* Use DynamoDB Global Tables for multi-region resilience.
* Separate ECS Fargate Tasks for token generation and validation.

**Design Overview**

* Fully serverless, built on AWS (ECS Fargate, DynamoDB, SNS).
* Resilient with multi-AZ and multi-region support.
* Modular design with distinct ECS Fargate for generation and validation.