### **Installing Terraform via CLI and Running an EC2 Instance (Part 1)**

Below is the step-by-step guide to installing Terraform on **Ubuntu** via CLI and running a script to launch an **EC2 instance**.

## **Step 1: Install Terraform**

**Update System Packages**  
sudo apt update && sudo apt install -y wget

sudo apt-get update && sudo apt-get install -y gnupg software-properties-common

**Autoremove excessive packages**

sudo apt autoremove -y

**Download and Install Terraform**

OFFICIAL DOCS: <https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli>

CHOOSE CORRECT OS from site  
wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg -y

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb\_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update && sudo apt upgrade -y

sudo apt install terraform=1.12.0 -y

**Verify Installation**  
terraform -version

You should see an output like:  
  
Terraform v1.x.x

## **Step 2: Configure AWS Credentials**

Before running Terraform, you need to set up your **AWS CLI** with credentials if necesary.

**Install AWS CLI**  
sudo apt install awscli -y

**Configure AWS Credentials**  
aws configure

Enter the following details:

* + **AWS Access Key ID**: your-access-key
  + **AWS Secret Access Key**: your-secret-key
  + **Default region**: us-east-1 (or your preferred AWS region)
  + **Default output format**: json (or leave blank)

## **Step 3: Write Terraform Script to Launch EC2**

Now, let's create a Terraform script to launch an EC2 instance.

**Create a new directory for Terraform files**  
mkdir terraform-ec2 && cd terraform-ec2

**Create main.tf file**  
nano main.tf

Add the following configuration:

To grab an AMI number, from EC2 console go to Images > Ami Catalog to grab a valid AMI id. Snapshot follows.

A close up of a logo

AI-generated content may be incorrect.  
  
provider "aws" {

region = "us-east-1" # Change this if needed

}

resource "aws\_instance" "example" {

ami = "ami-053a45fff0a704a43" # Replace with a valid AMI ID for your region

instance\_type = "t2.micro"

tags = {

Name = "Terraform-Instance"

}

}

## **Step 4: Run Terraform Commands**

**Initialize Terraform**  
terraform init

This downloads necessary provider plugins.

**Validate Terraform Configuration**  
terraform validate

If successful, you’ll see:  
Success! The configuration is valid.

**Plan the Terraform Execution**

terraform plan

This shows what Terraform will create.

**Apply the Terraform Configuration**  
terraform apply -auto-approve

This will create the EC2 instance.

## **Step 5: Verify EC2 Instance**

After Terraform applies the configuration, verify the instance:

aws ec2 describe-instances --query 'Reservations[\*].Instances[\*].{ID:InstanceId,State:State.Name,PublicIP:PublicIpAddress}'

Verify instance from your EC2 console as well.

## **Step 6: Destroy the EC2 Instance**

To clean up resources:

terraform destroy -auto-approve

This setup ensures Terraform is installed and provides an EC2 instance on AWS.

### **Creating a DynamoDB instance to store and secure passwords (Part 2)**

Here are the steps to modify your Terraform setup to include a DynamoDB table for storing an encrypted admin password, create a Python script to encrypt and store the password, and verify the record creation:

**1. Modify Terraform Configuration**

Add the following to your main.tf file in the terraform-ec2 directory:

resource "aws\_dynamodb\_table" "admin\_passwords" {

name = "AdminPasswords"

billing\_mode = "PAY\_PER\_REQUEST"

hash\_key = "username"

attribute {

name = "username"

type = "S"

}

tags = {

Name = "AdminPasswords"

}

}

resource "aws\_kms\_key" "password\_encryption\_key" {

description = "KMS key for encrypting admin passwords"

enable\_key\_rotation = true

}

resource "aws\_kms\_alias" "password\_encryption\_key\_alias" {

name = "alias/admin-password-encryption"

target\_key\_id = aws\_kms\_key.password\_encryption\_key.key\_id

}

**2. Create Python Script**

Create a file named encrypt\_password.py in the terraform-ec2 directory:

import boto3, base64, sys

def encrypt\_password(username, password):

kms\_client = boto3.client('kms')

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('AdminPasswords')

# Encrypt the password via AWS KMS

response = kms\_client.encrypt(

KeyId='alias/admin-password-encryption',

Plaintext=password.encode('utf-8')

)

encrypted\_password = base64.b64encode(response['CiphertextBlob']).decode('utf-8')

# Store in DynamoDB

table.put\_item(

Item={

'username': username,

'password': encrypted\_password

}

)

print(f"Encrypted password: {encrypted\_password}")

return encrypted\_password

if \_\_name\_\_ == "\_\_main\_\_":

if len(sys.argv) != 3:

print("Usage: python encrypt\_password.py <username> <password>")

sys.exit(1)

username = sys.argv[1]

password = sys.argv[2]

encrypt\_password(username, password)

To execute the Python script, run:

Python3 encrypt\_password.py admin your\_password\_here

A black background with white text

AI-generated content may be incorrect.

**3. CLI Command to Confirm Record Creation**

After running the Python script, use this AWS CLI command to verify the record creation:

aws dynamodb get-item --table-name AdminPasswords --key '{"username": {"S": "admin"}}' --query 'Item.password.S' --output text

This command will retrieve and display the encrypted password for the "admin" user.

A black screen with white text

AI-generated content may be incorrect.

To echo the password, you can also use:

echo $(aws dynamodb get-item --table-name AdminPasswords --key '{"username": {"S": "admin"}}' --query 'Item.password.S' --output text)

Remember to run terraform apply -auto-approve after modifying the Terraform configuration to create the DynamoDB table and KMS key before running the Python script.

A screen shot of a computer

AI-generated content may be incorrect.

Note, you can view the DynamoDB Dashboard table creation in AWS by going to the “Database” section from services menu and on the left panel click on Tables.

Further you can verify records in your Dashboard by clicking on Explore items in your left panel

A screenshot of a computer

AI-generated content may be incorrect.

If you decide to decrypt the encrypted password created in python you can do the following

import boto3, base64

def decrypt\_password(encrypted\_password):

kms\_client = boto3.client('kms')

# Decode the base64 encoded encrypted password

encrypted\_blob = base64.b64decode(encrypted\_password)

# Decrypt the password using AWS KMS

response = kms\_client.decrypt(CiphertextBlob=encrypted\_blob)

# Decode the decrypted password from bytes to string

decrypted\_password = response['Plaintext'].decode('utf-8')

return decrypted\_password

# Example usage

encrypted\_password = "Your\_Base64\_Encoded\_Encrypted\_Password\_Here"

decrypted\_password = decrypt\_password(encrypted\_password)

print(f"Decrypted password: {decrypted\_password}")

To use this script:

1. Replace "Your\_Base64\_Encoded\_Encrypted\_Password\_Here" with the actual encrypted password you obtained from the encryption process.

2. Ensure you have the necessary AWS permissions to use the KMS decrypt operation.

3. Run the script in the same AWS environment where the encryption was performed.

This decryption process uses the same KMS key that was used for encryption, so you don't need to specify the key explicitly if you're using the same AWS credentials and region

----  
AWS Key Management Service (KMS) is a fully managed encryption and key management service provided by Amazon Web Services (AWS). It allows users to create and control the encryption keys used to encrypt their data, and it provides a centralized service for managing cryptographic keys across a variety of AWS services and applications.

Key Features of AWS KMS:

* Key Management: AWS KMS provides a simple and easy way to create and manage encryption keys, including both customer-managed and AWS-managed keys.
* Data Encryption: You can use KMS to encrypt data stored in various AWS services, such as Amazon S3, Amazon EBS, and Amazon RDS. It allows you to encrypt individual objects or entire buckets in S3, volumes in EBS, and database entries in RDS.
* Integration with Other AWS Services: KMS integrates with many AWS services, enabling encryption and key management seamlessly within the AWS ecosystem. Services like Amazon S3, Amazon EBS, Amazon RDS, AWS Lambda, and others can use KMS for encryption.
* Access Control: AWS Identity and Access Management (IAM) policies can be used to control access to KMS keys. This allows you to define who can use the keys and how they can be used.
* Audit and Compliance: KMS supports logging and monitoring of key usage through AWS CloudTrail. This allows you to track the use of your keys and integrate with compliance requirements.
* Regional and Multi-Region Keys: Keys can be created and managed at the regional level, and you can also enable multi-region keys that can be used across different AWS regions.
* HSM-backed Keys: AWS KMS uses hardware security modules (HSMs) to protect the security of your keys. It also offers the option to manage your own keys outside of KMS using the CloudHSM service.
* Custom Key Store: You can create a custom key store in AWS KMS to store your cryptographic keys in a custom hardware security module (HSM) that you manage.

Use Cases for AWS KMS:

* Data Encryption: Protect sensitive data stored in AWS services by encrypting it with keys managed by KMS.
* Secure Application Development: Integrate KMS into applications to securely handle cryptographic operations and key management.
* Regulatory Compliance: Use KMS to meet various compliance requirements related to data encryption and key management.

Pricing:

AWS KMS operates on a pay-as-you-go pricing model, which typically includes charges for key management, API requests, and additional features. It's essential to consult the AWS Pricing page for KMS to understand current pricing models.

Best Practices:

* Follow the principle of least privilege by defining IAM policies that minimize access to KMS keys.
* Regularly audit key usage and access logs.
* Use AWS CloudTrail to monitor and log all API calls made to KMS.
* Rotate keys regularly to enhance security.

AWS KMS is a robust solution for managing encryption keys in the cloud, providing security and compliance benefits for applications and data hosted in AWS.

Further explorations:

**Steps to Query the Password via Postman:**

1. **Open Postman and Create a New Request:**
   * Launch Postman.
   * Click the "+" icon to create a new request tab.
2. **Set Request Type and URL:**
   * Choose POST as the HTTP method.
   * Enter the DynamoDB endpoint for your region. This will be in the format: https://dynamodb.YOUR\_AWS\_REGION.amazonaws.com/ Replace YOUR\_AWS\_REGION with your AWS region (e.g., us-east-1, us-west-2).
3. **Set Headers:**
   * Go to the "Headers" tab.
   * Add the following headers:

| **Key** | **Value** |
| --- | --- |
| Content-Type | application/x-amz-json-1.0 |
| X-Amz-Target | DynamoDB\_20120810.GetItem |

1. **Configure AWS Authentication:**
   * Go to the "Authorization" tab.
   * Select "AWS Signature" as the type. This will expand fields for your AWS credentials.
   * Enter the following:
     + **Access Key:** Your AWS Access Key ID
     + **Secret Key:** Your AWS Secret Access Key
     + **AWS Region:** Your AWS region (e.g., us-east-1)
     + **Service Name:** dynamodb
2. **Set Request Body:**
   * Go to the "Body" tab.
   * Select "raw" and then choose "JSON" from the dropdown.
   * Enter the following JSON payload. Replace admin with the username you used when encrypting/storing the password.

{

"TableName": "AdminPasswords",

"Key": {

"username": {

"S": "admin"

}

},

"ProjectionExpression": "password"

}

* + - TableName: Specifies the DynamoDB table name.
    - Key: Specifies the primary key of the item you want to retrieve. In this case, we're querying by the username attribute (hash key). S indicates that the username value is a string.
    - ProjectionExpression: This is optional, but very useful. It tells DynamoDB *only* to return the password attribute from the item. Without this, DynamoDB would return the *entire* item (username and password). Returning only what you need is generally more efficient.

1. **Send the Request:**
   * Click the "Send" button.
2. **Examine the Response:**
   * In the bottom section of Postman, you'll see the response from DynamoDB.
   * The response will be a JSON object containing the item with the requested attributes. If you used the ProjectionExpression, you should only see the password attribute:

json

{

"Item": {

"password": {

"S": "YOUR\_ENCRYPTED\_PASSWORD"

}

}

}

Replace YOUR\_ENCRYPTED\_PASSWORD with the actual encrypted password from DynamoDB.

**Important Considerations:**

* **Security:** Storing AWS credentials directly in Postman is **not recommended for production environments**. For local testing, it's acceptable, but for anything beyond that, explore more secure options like using IAM roles or temporary credentials. Avoid committing Postman collections containing your AWS credentials to version control!
* **Error Handling:** Check the Postman console for any errors. Common problems include incorrect region, incorrect table name, or invalid AWS credentials.
* **Permissions:** Make sure your AWS user (the one associated with the credentials you're using in Postman) has the necessary IAM permissions to perform dynamodb:GetItem on the AdminPasswords table and permission to use the KMS key. The user must also have kms:Decrypt permission on the key alias alias/admin-password-encryption.
* **KMS Key Policy:** The KMS key's policy must allow the AWS user associated with the credentials you're using in Postman to decrypt data using the KMS key.

Even further explorations:

How to retrieve the encrypted password from DynamoDB using the AWS CLI, after you've stored it using the Python script and Terraform resources from your uploaded file. Here's a breakdown of the AWS CLI command and how it works:

**AWS CLI Command:**

aws dynamodb get-item \

--table-name AdminPasswords \

--key '{"username": {"S": "admin"}}' \

--projection-expression "password" \

--expression-attribute-names '{"#P": "password"}' \

--query 'Item.#P.S' \

--output text

**Explanation:**

* aws dynamodb get-item: This is the AWS CLI command for retrieving a single item from a DynamoDB table.
* --table-name AdminPasswords: This specifies the name of the DynamoDB table you created (the one storing the encrypted passwords). Make sure this matches the exact name of your table.
* --key '{"username": {"S": "admin"}}': This defines the primary key of the item you want to retrieve. It's providing the value for the "username" attribute (the hash key).
  + "username": The name of your hash key attribute in DynamoDB.
  + {"S": "admin"}: This specifies the value for the "username" attribute. "S" means the value is a string, and "admin" is the specific username you are querying for. Make sure this matches the *exact* username you used when storing the password.
* --projection-expression "password": This instructs DynamoDB to only retrieve the value of the "password" attribute. This is more efficient than retrieving the entire item.
* --expression-attribute-names '{"#P": "password"}': This is required because password is a reserved word, so we need to use an expression attribute name as a placeholder.
* --query 'Item.#P.S': This extracts the string value (.S) from the attribute called '#P' (which refers to password). This makes sure the output only contains the encrypted password itself.
* --output text: This formats the output as plain text, making it easier to copy and paste the password value. Without this, the output would be in JSON format.

**How to use it:**

1. **Make sure the AWS CLI is configured:** You must have configured the AWS CLI with valid credentials for your AWS account. If you haven't already, run aws configure and enter your AWS Access Key ID, Secret Access Key, default region, and default output format.
2. **Execute the command:** Copy and paste the command into your terminal and press Enter.
3. **View the output:** The command will output the encrypted password (if the command is executed successfully):

YOUR\_ENCRYPTED\_PASSWORD

**Troubleshooting:**

* **Incorrect Table Name:** Double-check that AdminPasswords is the correct table name.
* **Incorrect Username:** Make sure "admin" matches the exact username you stored the password under.
* **AWS Credentials:** Verify that your AWS CLI credentials are valid and have the necessary permissions to access DynamoDB ( dynamodb:GetItem permission to the AdminPasswords table).
* **Missing Attributes:** If the command returns nothing or an error, double-check that the item with the specified key ("username" = "admin") actually exists in the table.

This command provides a concise way to retrieve the encrypted password from DynamoDB using the AWS CLI. Remember to keep your AWS credentials secure and avoid exposing them in scripts or configuration files. If you're using this in a more automated way, consider using IAM roles and instance profiles to provide temporary credentials to your resources.

### **A linting will go! (Part 3)**

To lint YAML or JSON files in Ubuntu, you can use tools like yamllint for YAML and jq for JSON. Here's how to install and use jq for JSON linting:

**Installing jq**

To install jq on Ubuntu, use the following command:

sudo apt-get update

sudo apt-get install jq

**Linting JSON with jq**

After installation, you can use jq to lint JSON files. Here's an example:

jq '.' your\_file.json

If the JSON is valid, jq will output the formatted JSON. If there are syntax errors, it will display an error message.

**Linting YAML with yamllint**

For YAML files, you can use yamllint. First, install it:

sudo apt-get install yamllint

Then, use it to lint YAML files:

yamllint your\_file.yaml

yamllint will check for syntax validity and other issues like key repetition, line length, and indentation problems.

**Additional JSON Linting Options**

For more detailed JSON validation, you can use jsonlint:

sudo apt-get install jsonlint

jsonlint-php your\_file.json

jsonlint provides more verbose output and can be useful for identifying specific issues in JSON files.

Remember, these tools not only check for syntax validity but also for formatting and style issues, helping maintain consistent and clean code across your projects.