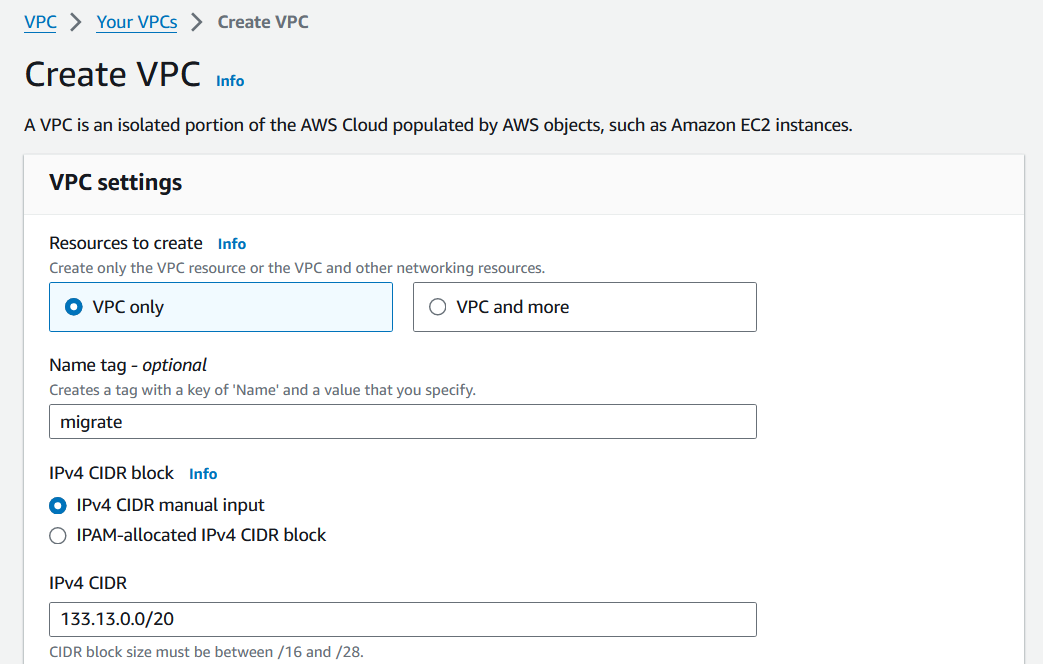
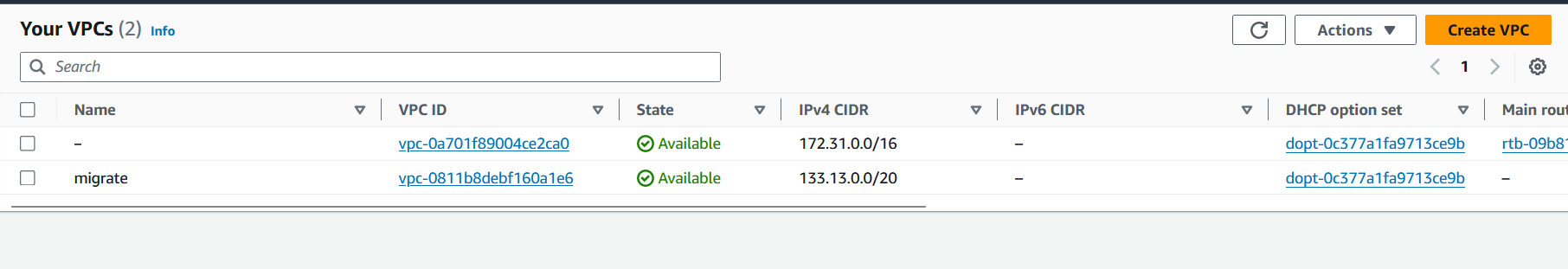
**Migration from AWS to Azure**

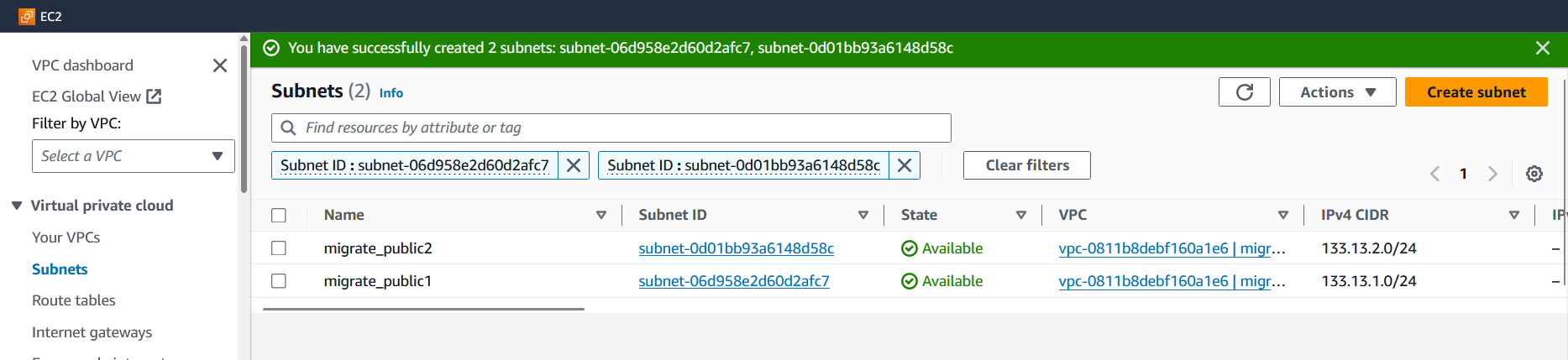
**Part I – Setup web application in aws**

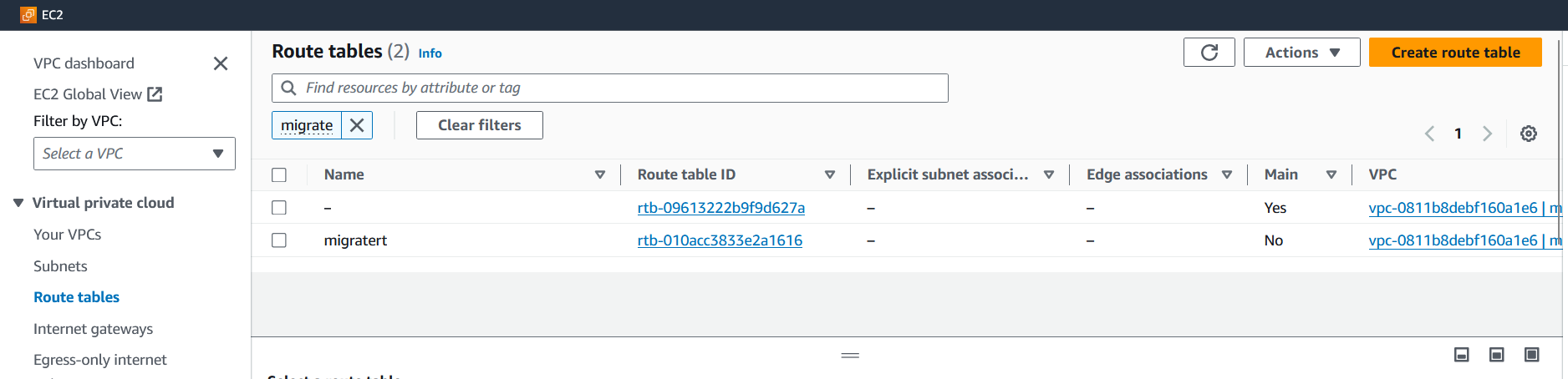
1. Create a VPC

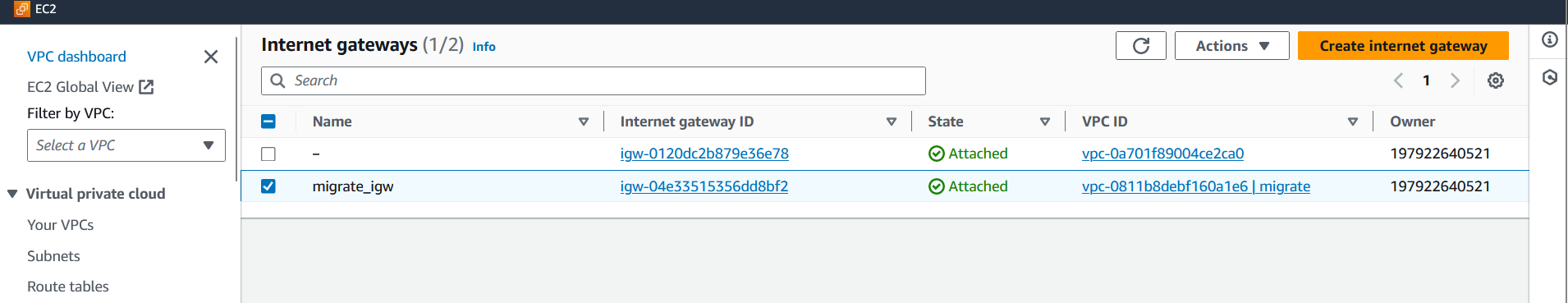




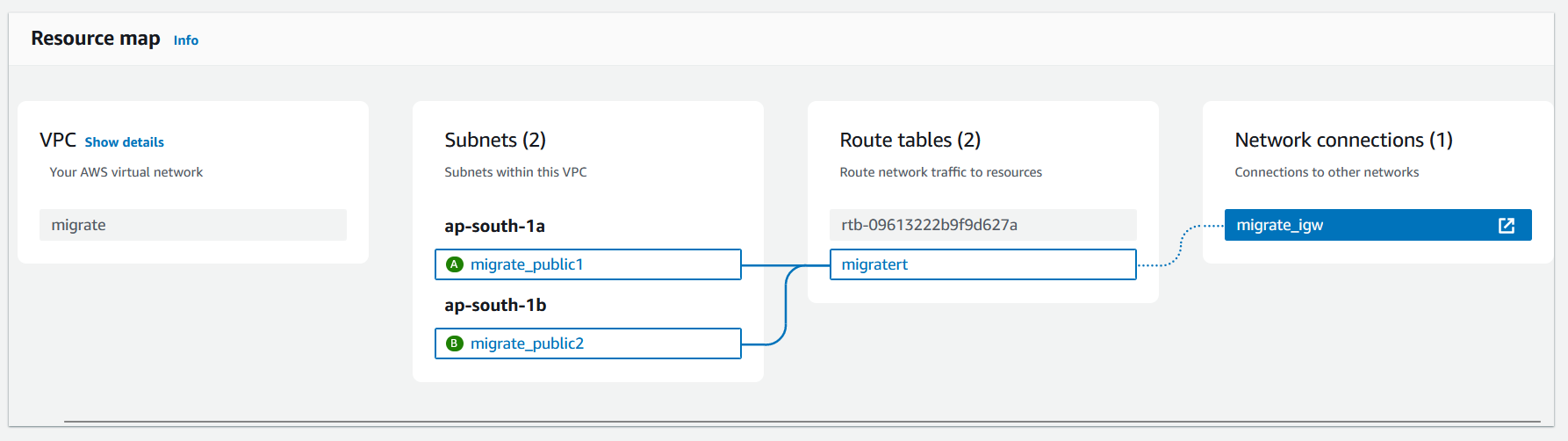
1. Configure public subnets, route table and internet gateway.



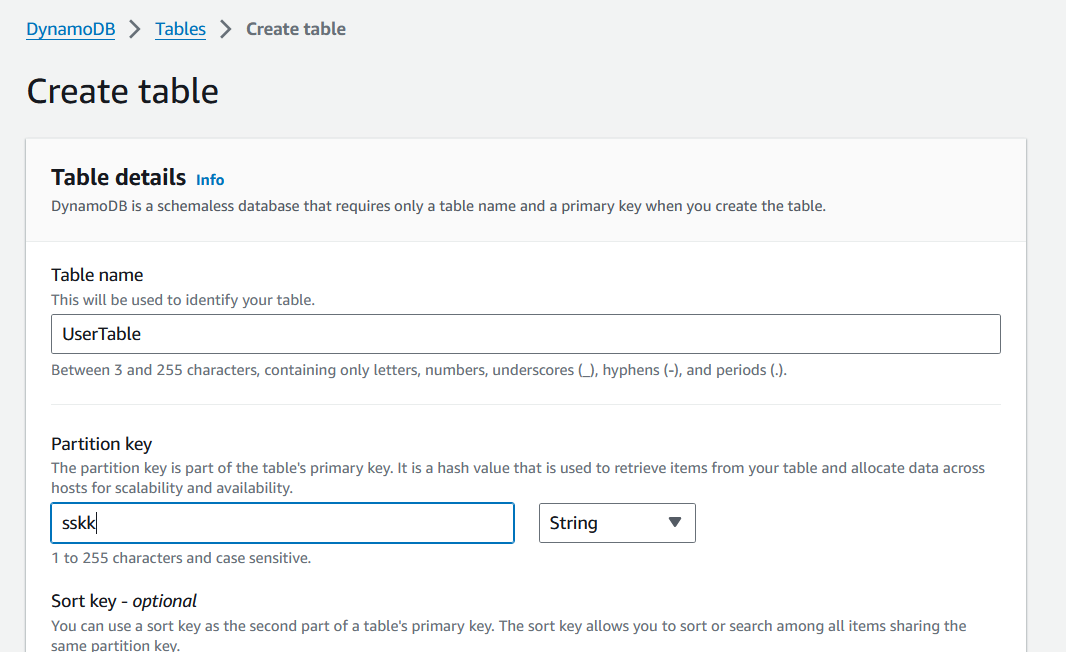


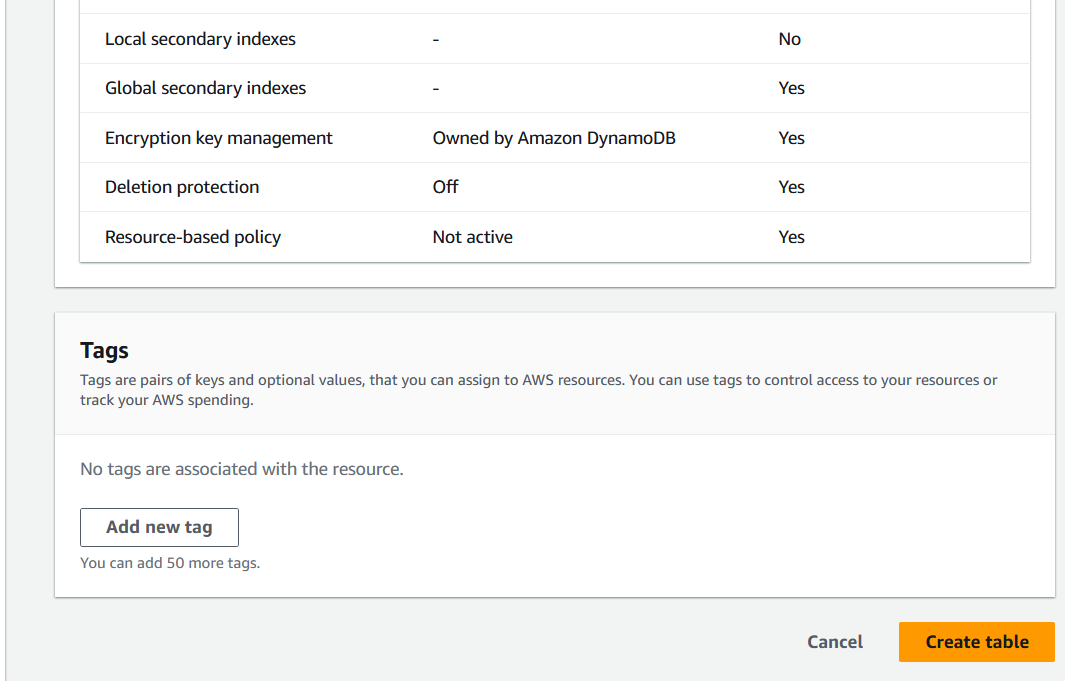


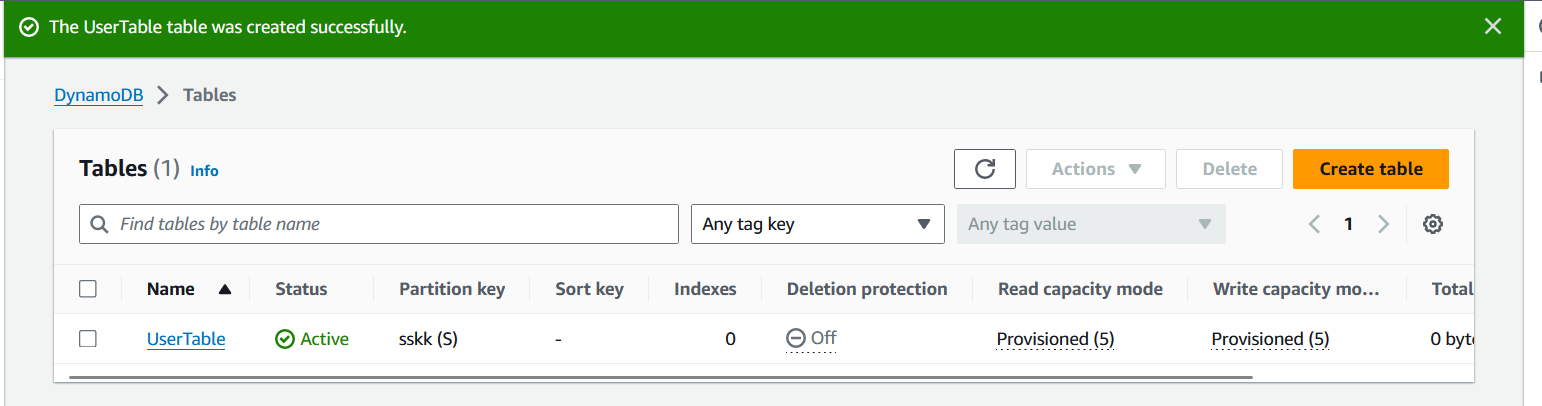
1. Attach VPC to gateway and associate the subnets. Add the internet gateway to the routetable. Verify using the resourcemap of VPC



1. Create a dynamodb database for the simple site. Since we need the details of db to be embeddeded in the coding part, we are launching the database first.

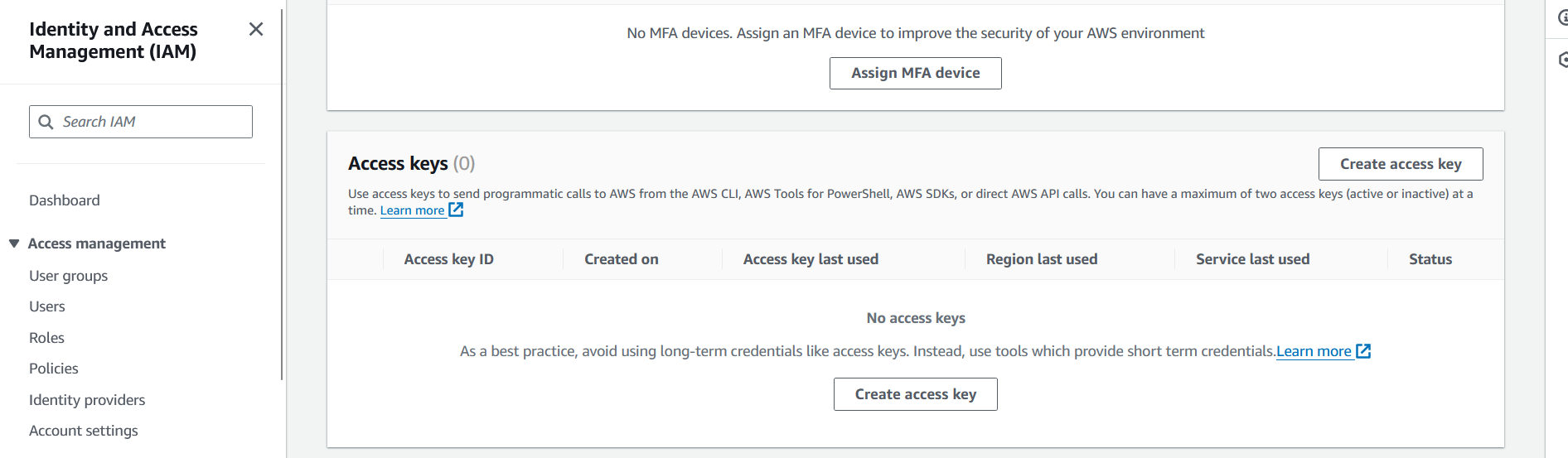


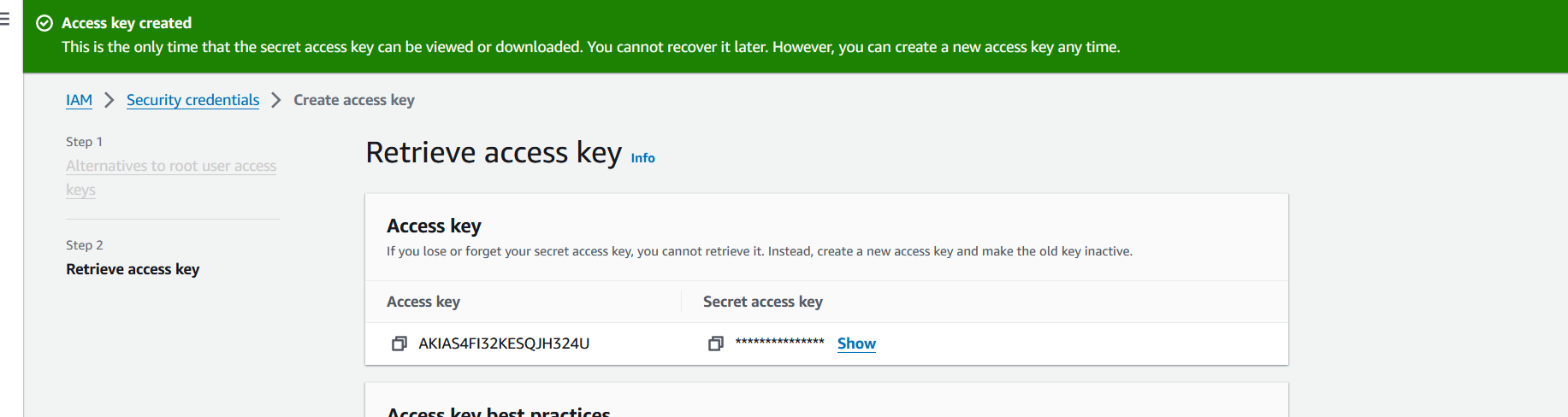




1. **Access Key ID and Secret Access Key**

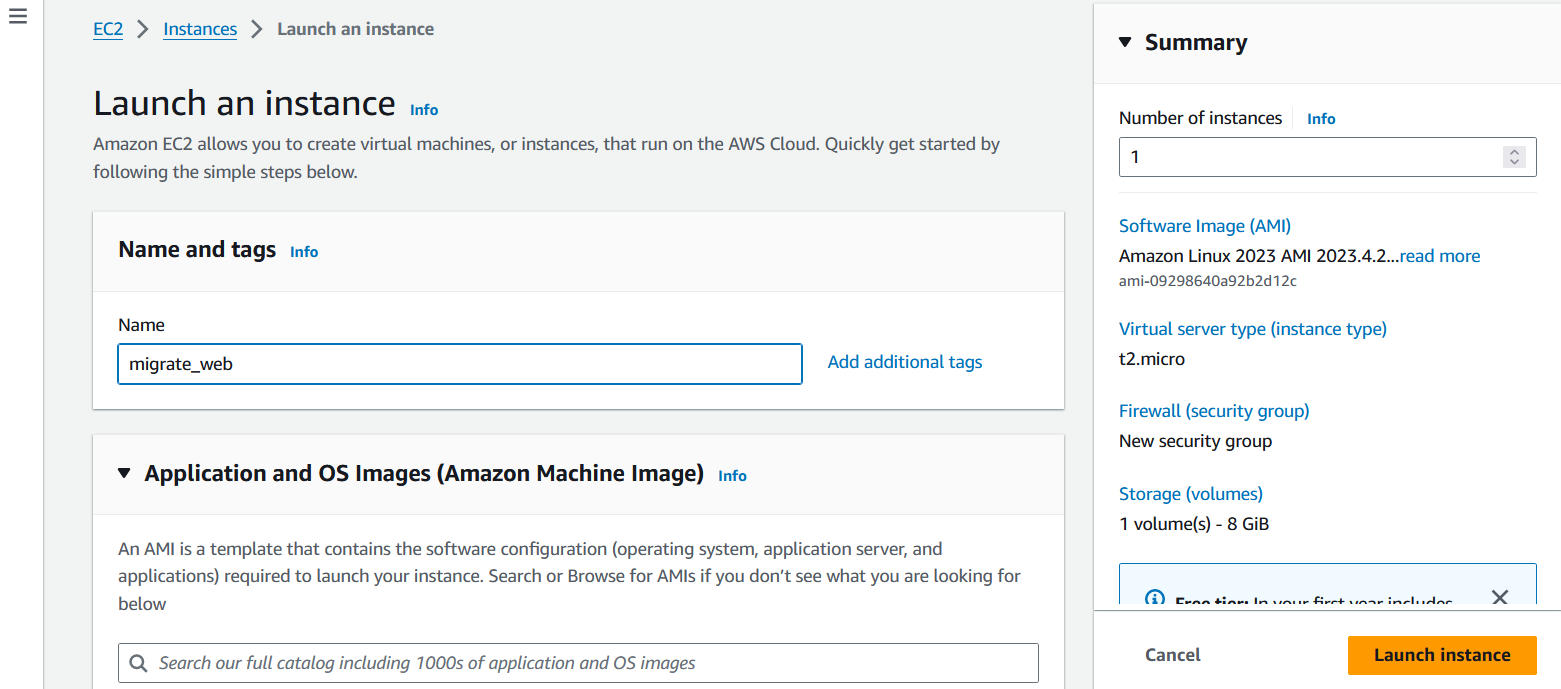
Click on "Create access key" if you need to generate a new access key pair. Make sure to save the secret access key securely as it will not be shown again.

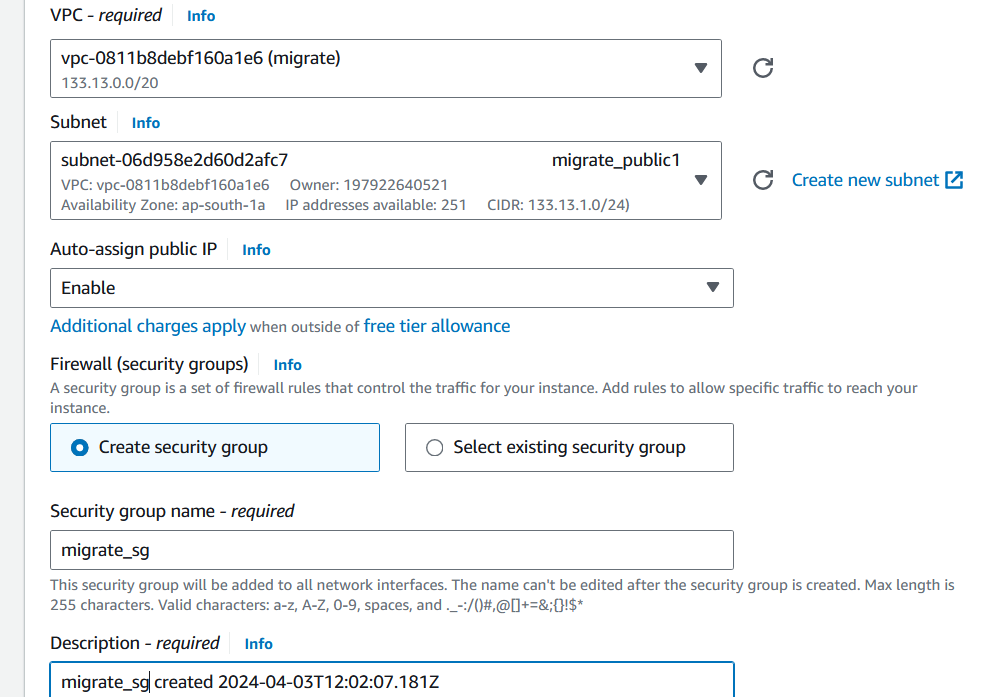


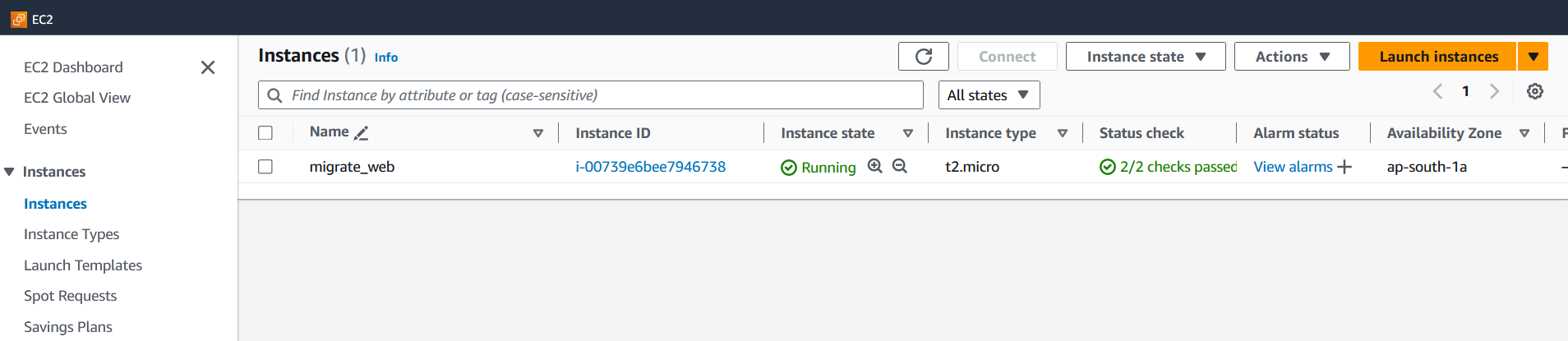


Note down the Access key and Secret Access Key for further process during coding

1. **Create an EC2 instance to serve the simple site**







**Create an IAM Policy**:

Go to the IAM console in the AWS Management Console.

Navigate to "Policies" and click on "Create policy".

Choose the "JSON" tab and enter a policy document that grants the necessary permissions to write data to DynamoDB.

**Create an IAM Role**:

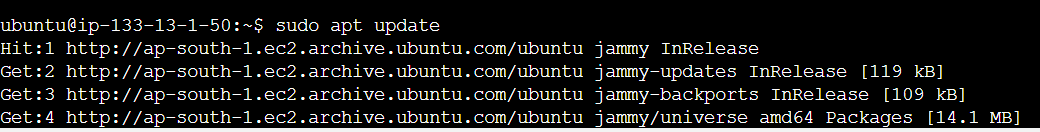
Go to the IAM console and navigate to "Roles".

Click on "Create role".

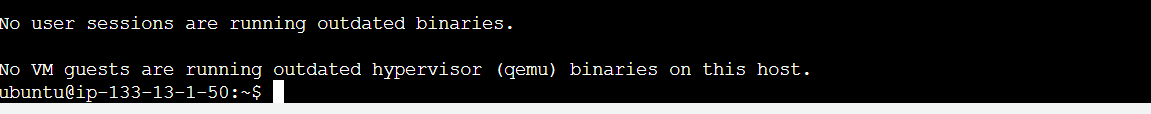
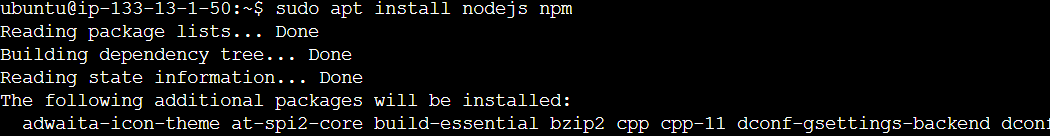
Choose the service that will use this role (select "EC2").

Attach the policy you created earlier to this role

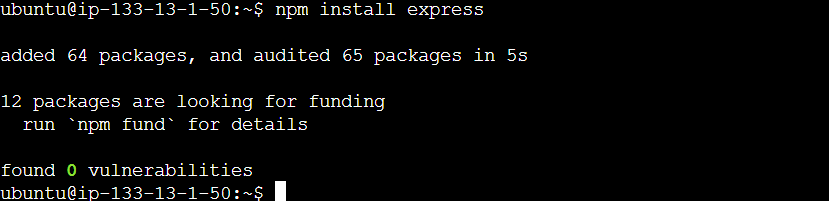
1. **Install dependencies**



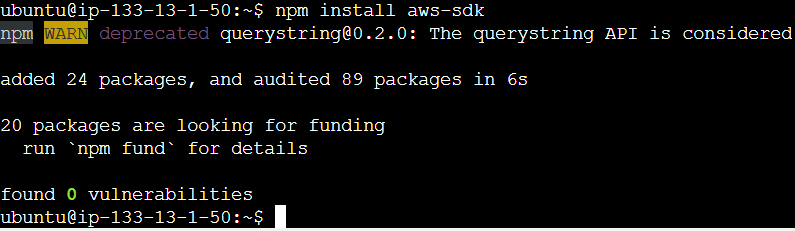
**Node.js and npm**: These are required to run your JavaScript code. You can install them by following the instructions provided by the official Node.js website or by using a package manager like apt on Ubuntu.



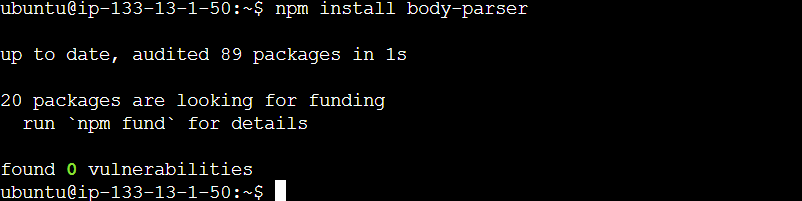
**Express.js**: This is a web application framework for Node.js. It simplifies the process of creating web servers and handling HTTP requests.

****

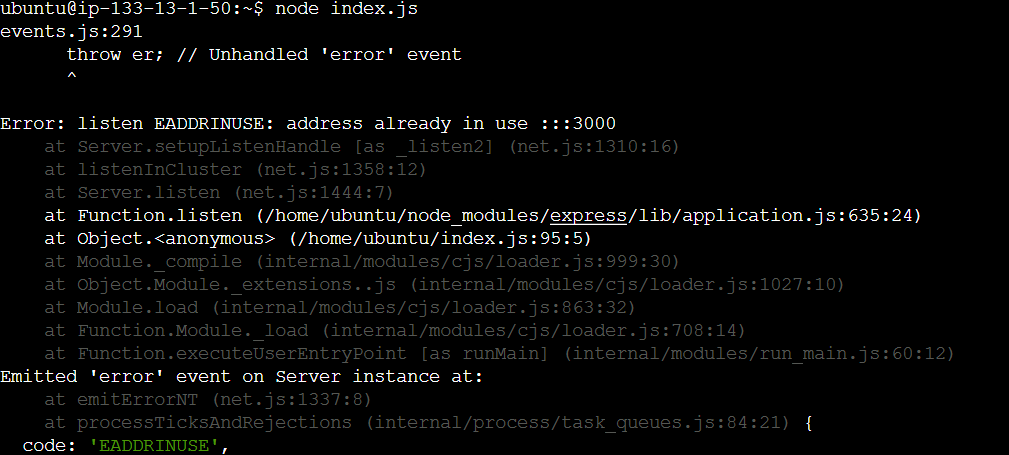
**AWS SDK**: If you're using AWS DynamoDB in your application, you'll need the AWS SDK to interact with DynamoDB.



**Body-parser**: This middleware parses incoming request bodies in a middleware before your handlers, available under the req.body property.

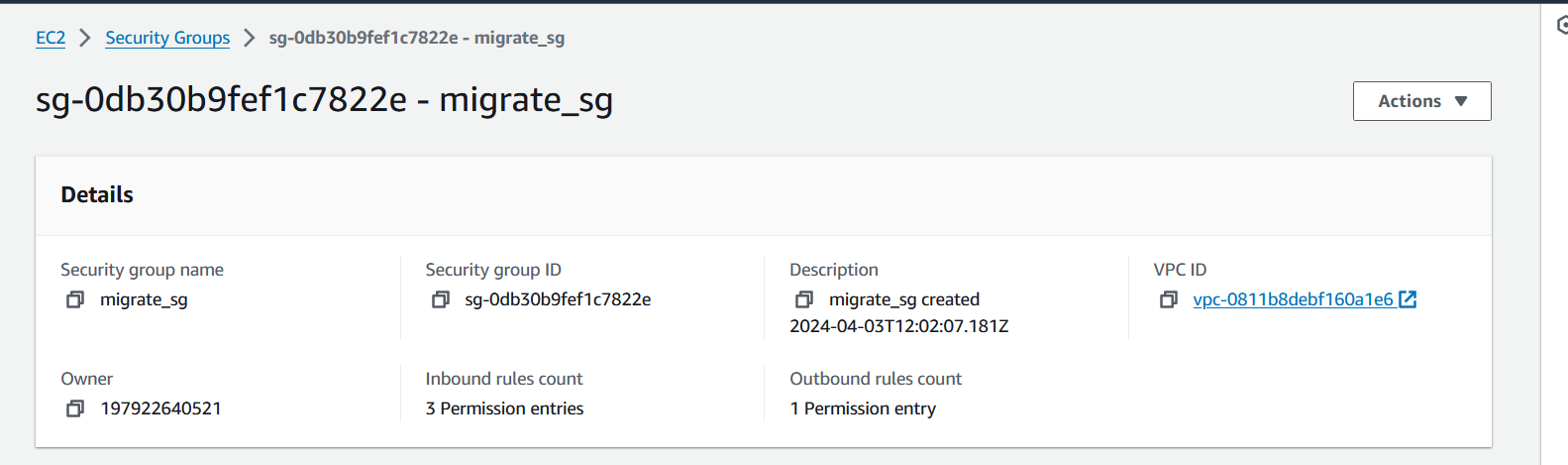


With these dependencies installed, we can create a single JavaScript file that includes your frontend and backend logic, and then run it using Node.js.



1. **Configure the instance to serve the webfiles**

* **Select the EC2 Instance** where your Node.js server is running.
* **Find the Security Group** associated with your EC2 instance. This is listed under the "Description" tab of the instance details.
* **Click on the Security Group** to view its details.



* **Edit the Inbound Rules of the security group to allow traffic on port 3000.**

Click on the "Inbound rules" tab.

Click "Edit inbound rules".

Add a new rule with the following configuration:

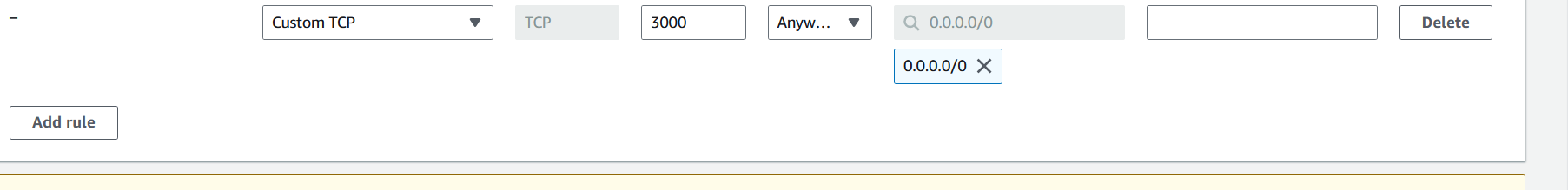
Type: Custom TCP

Protocol: TCP

Port Range: 3000

Source: 0.0.0.0/0 (this allows traffic from any IP address)

Description: Optionally, provide a description for the rule.

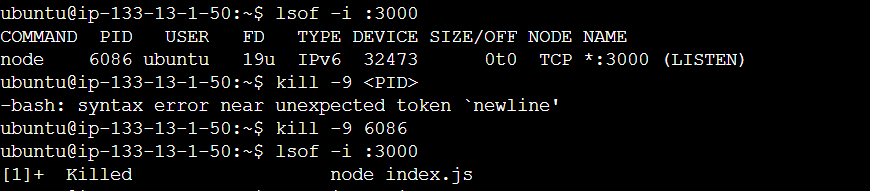


1. **Troubleshooting**

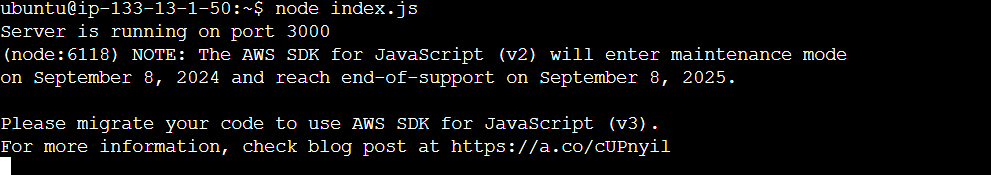
The error message EADDRINUSE: address already in use indicates that the port 3000, on which your Node.js server is trying to listen, is already in use by another process.

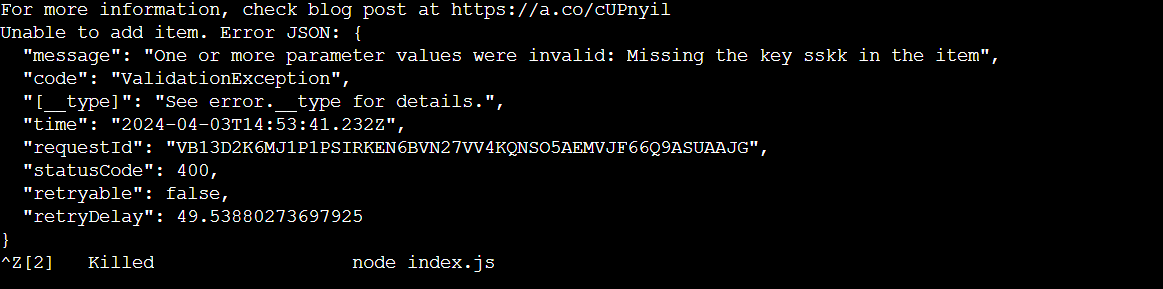
To resolve this issue, you have a few options:

* **Terminate the Existing Process**:



* **Now run the js file**





* **Modify the code to include the partition key value as given while creating database**

<form id="addItemForm">

<label for="partitionKey">Partition Key:</label>

<input type="text" id="partitionKey" name="partitionKey"><br><br>

<label for="attribute1">Attribute 1:</label>

<input type="text" id="attribute1" name="attribute1"><br><br>

<label for="attribute2">Attribute 2:</label>

<input type="text" id="attribute2" name="attribute2"><br><br>

<button type="submit">Add Item</button>

</form>

const params = {

TableName: tableName,

Item: {

'partitionKey': partitionKey,

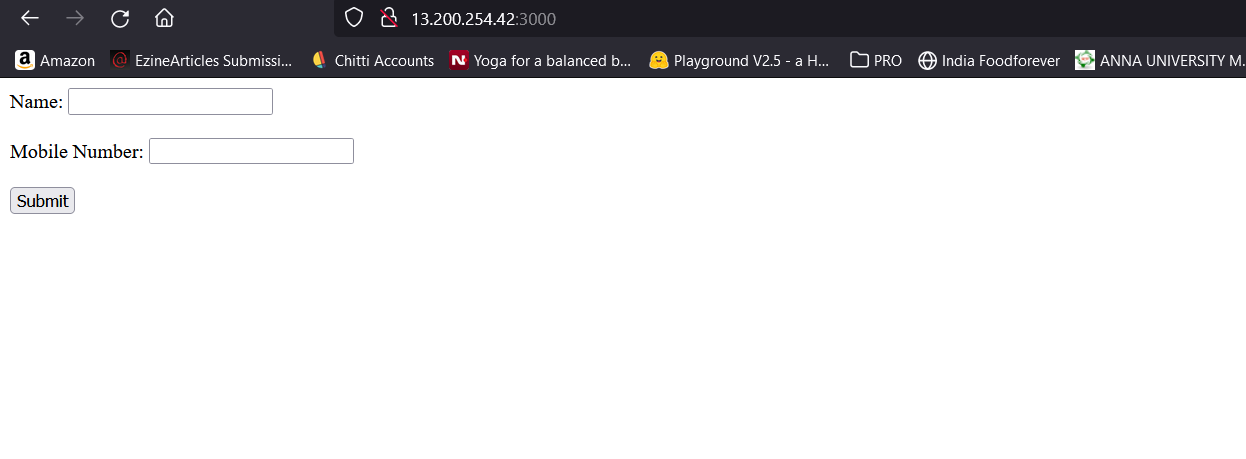
'attribute1': attribute1,

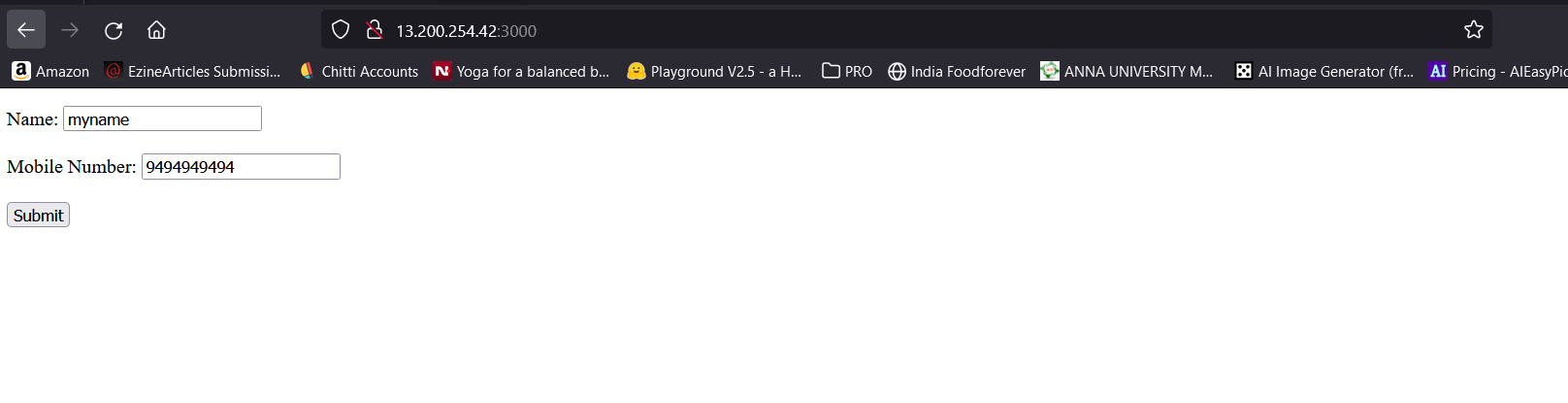
'attribute2': attribute2

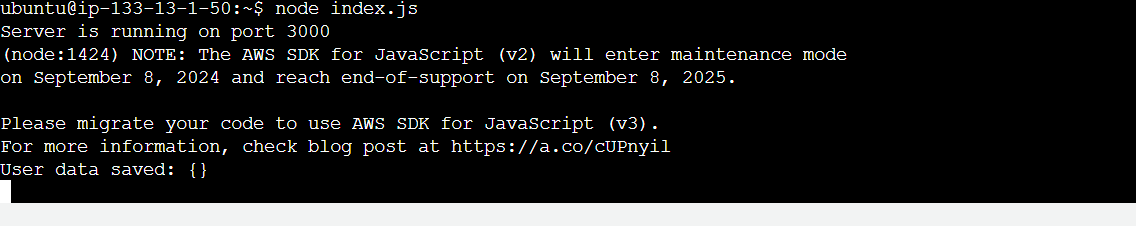
}

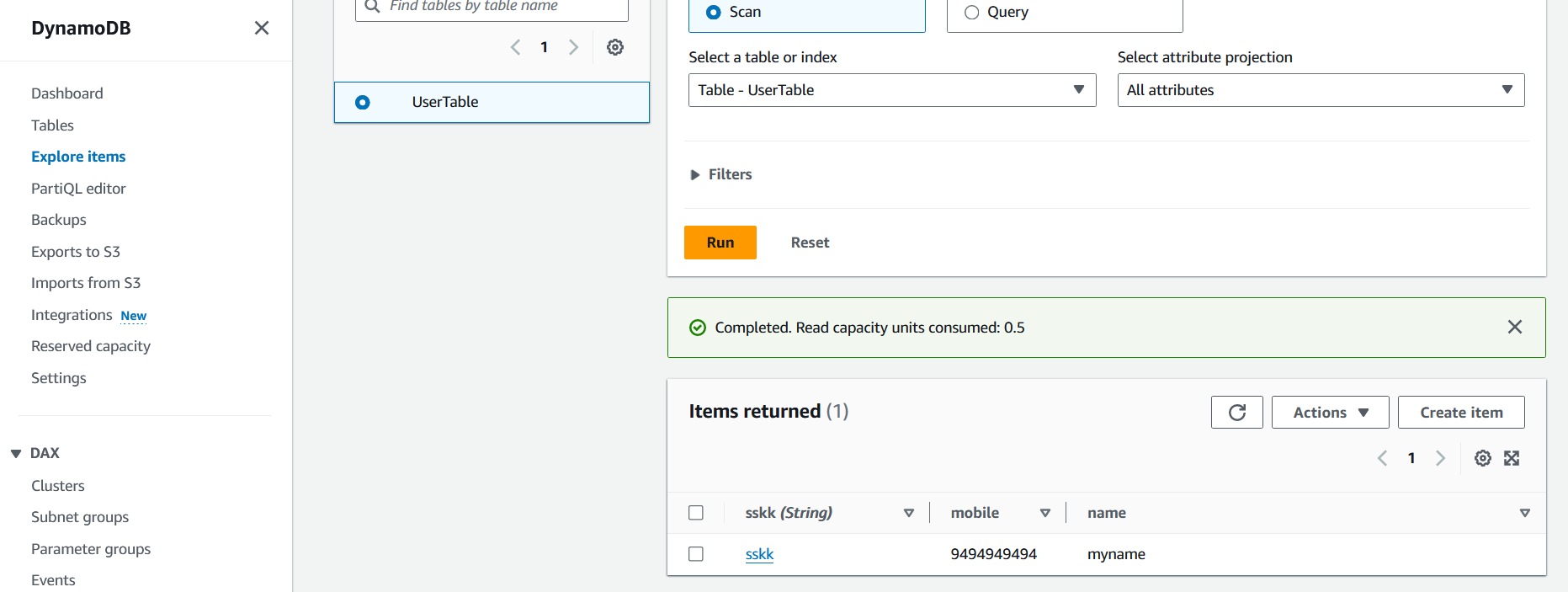
};

1. **Verification of web application**









**Part II – Migration to Azure**

Migrating from DynamoDB to Azure Cosmos DB involves several steps. Here's a general approach to migrating your data:

* + 1. **Data Assessment and Planning:**

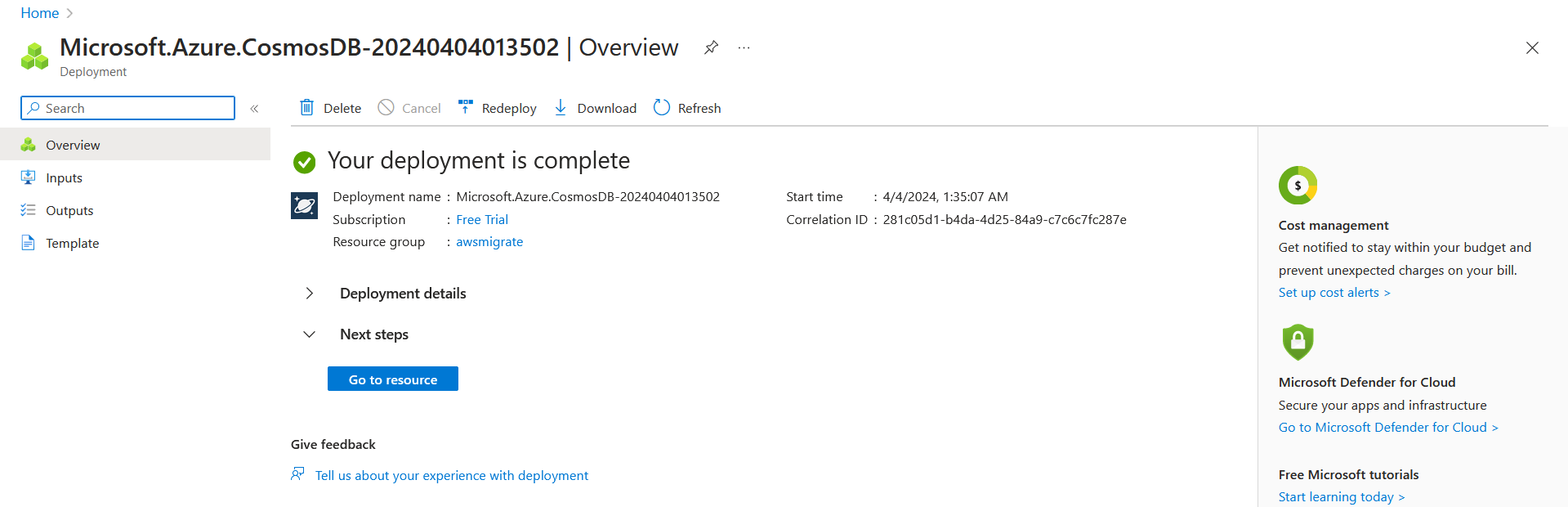
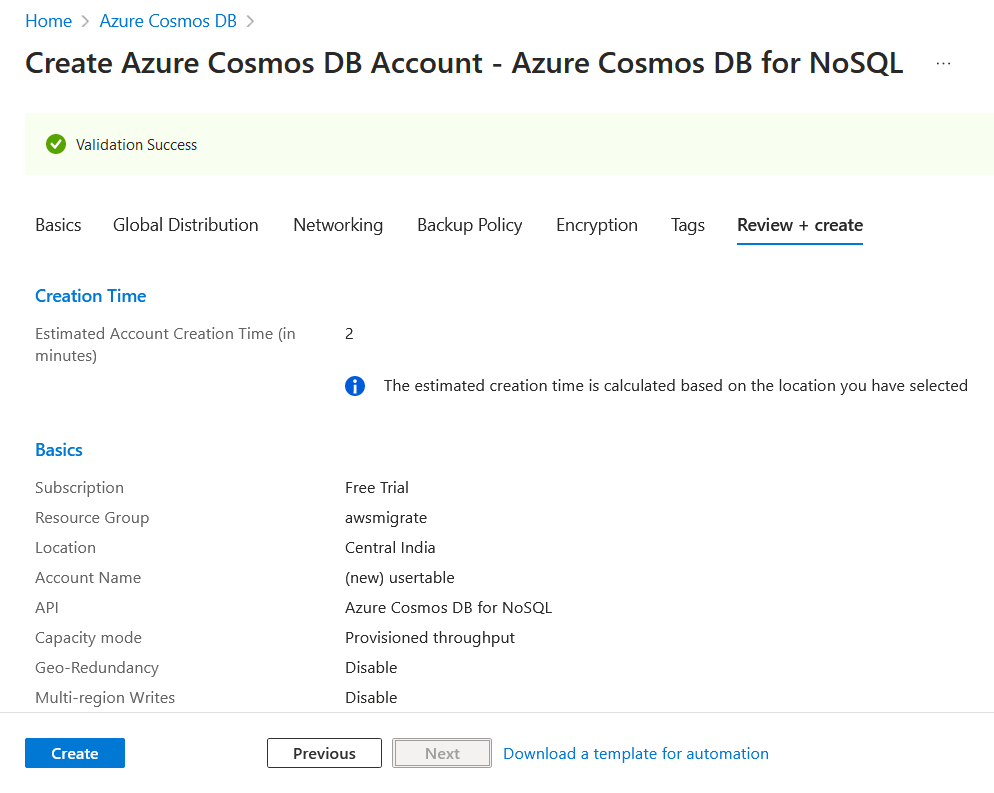
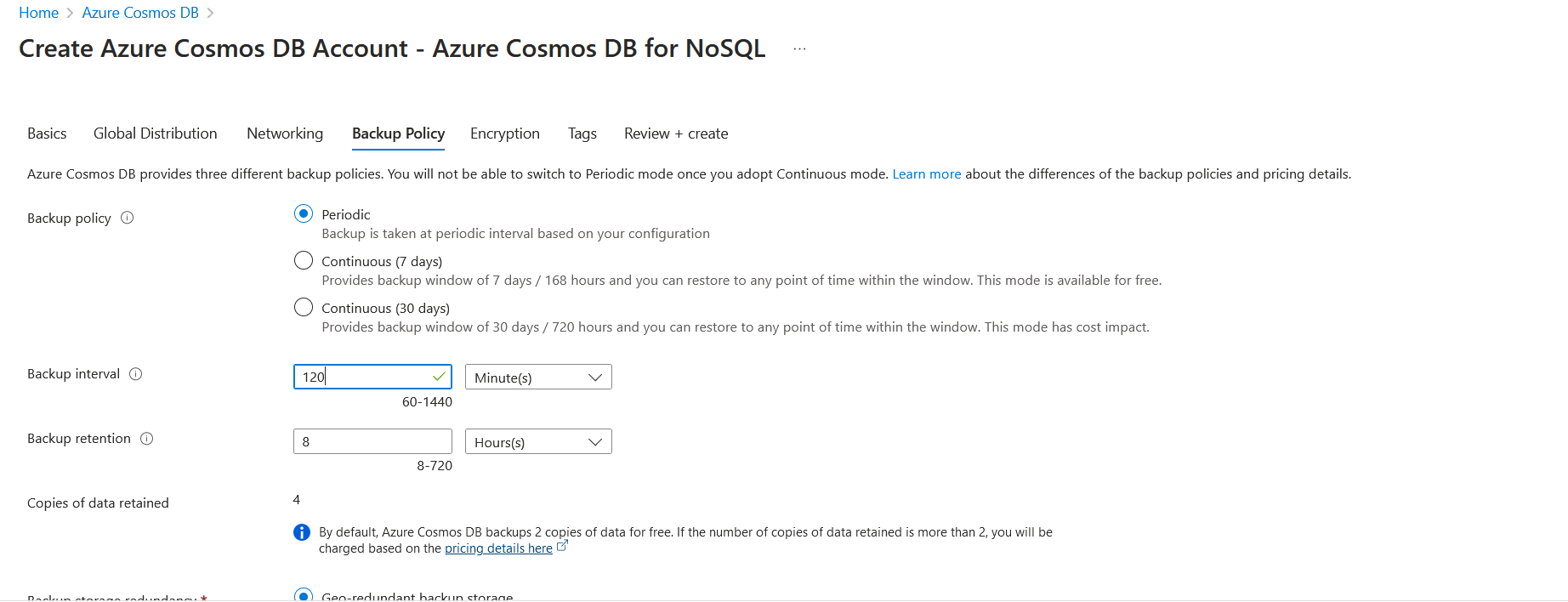
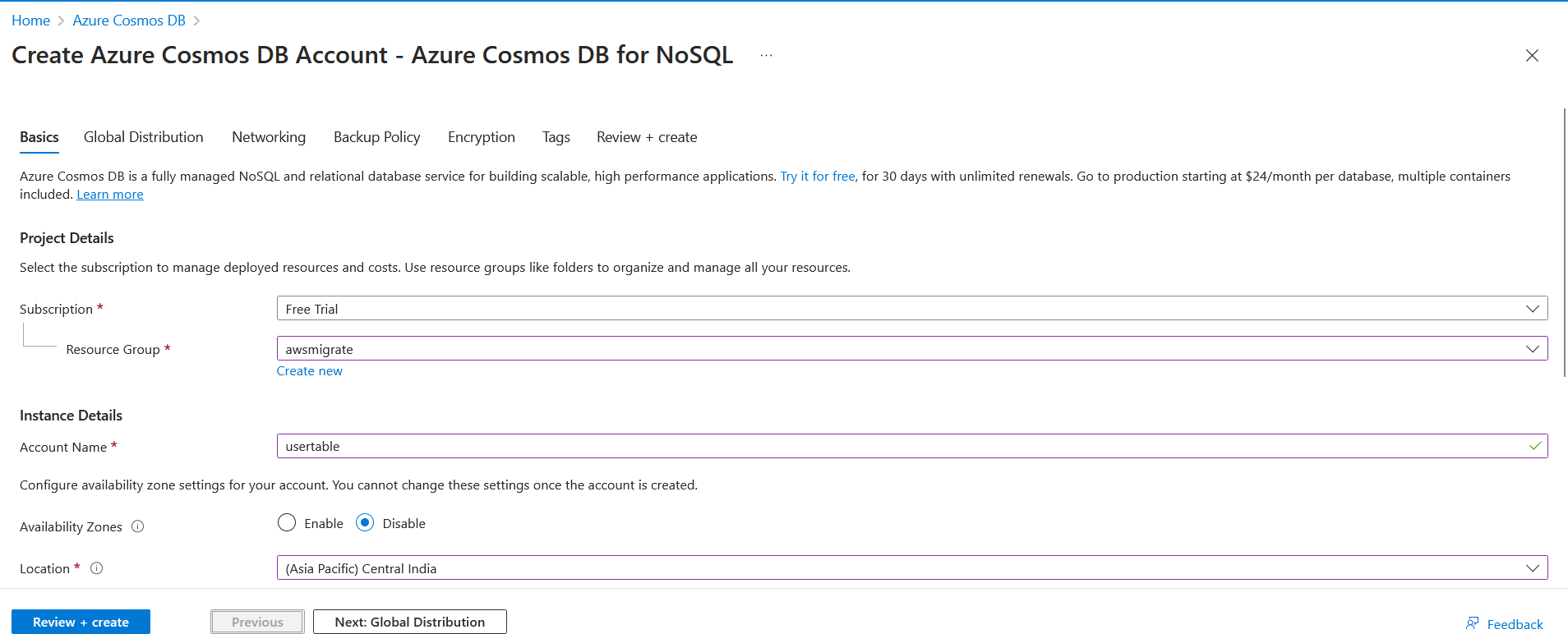
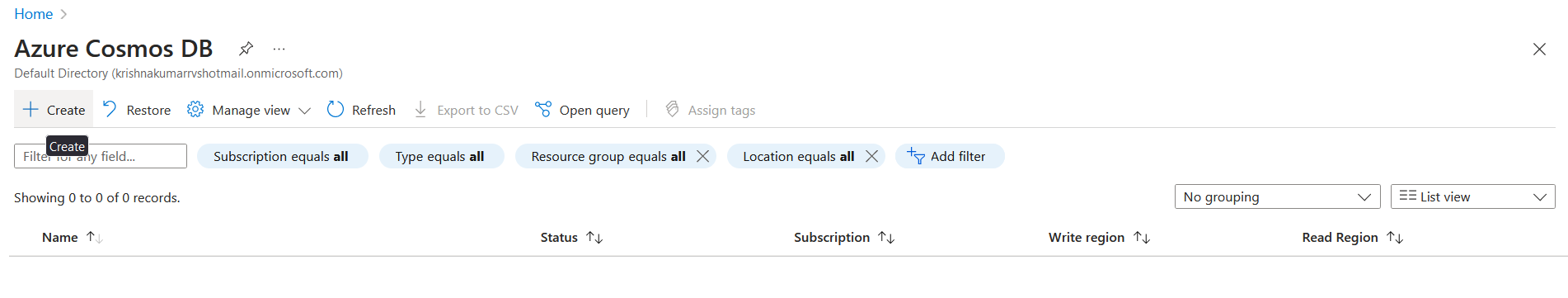
Understand your DynamoDB data model, including table structures, primary keys, secondary indexes, and throughput settings.

Identify the data access patterns used in your application to design the appropriate data model in Azure Cosmos DB.

* + 1. **Provision Azure Cosmos DB Account:**

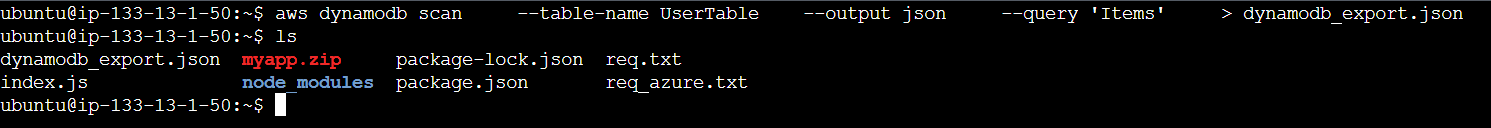
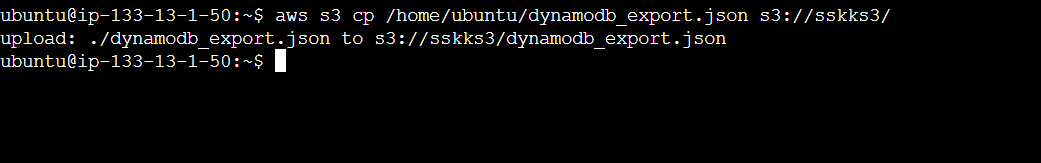
Log in to the Azure portal (<https://portal.azure.com>) and create a new Azure Cosmos DB account.

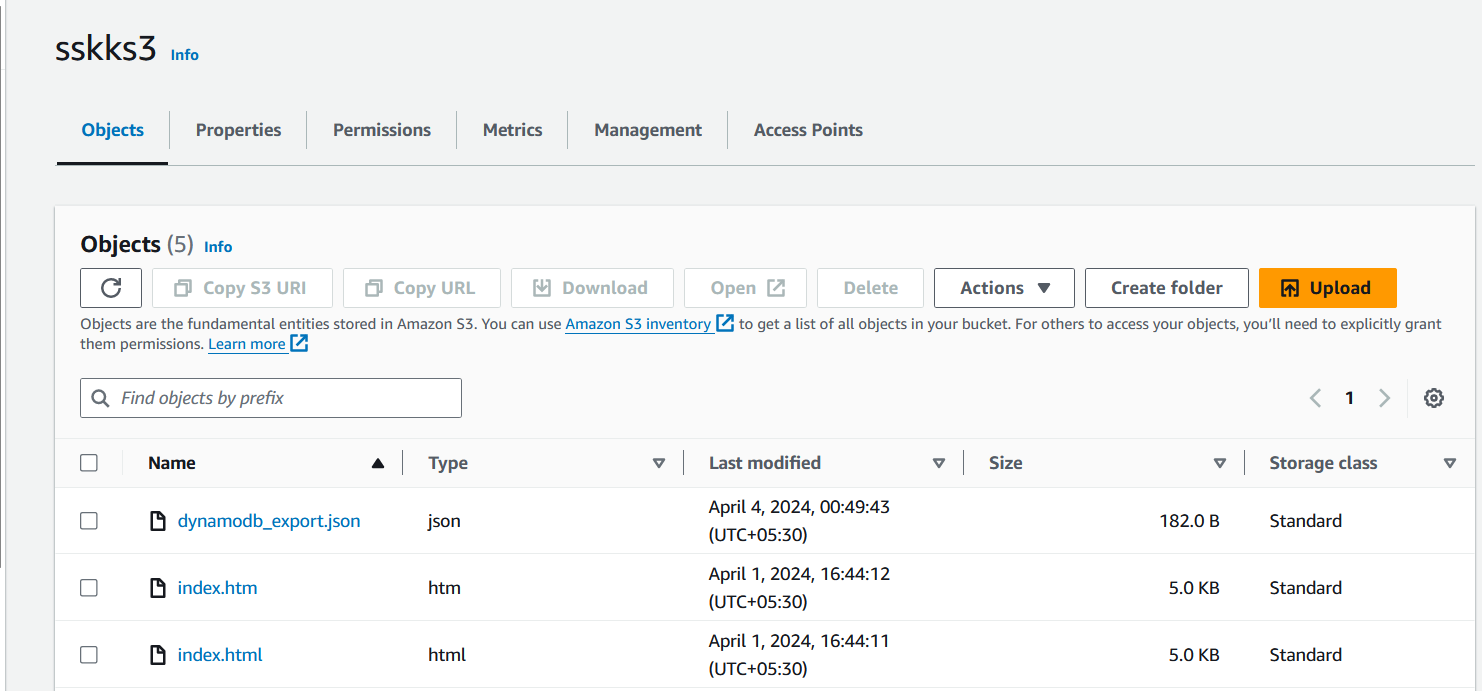
Choose the API that corresponds to your existing data model in DynamoDB (e.g., SQL API, MongoDB API, Cassandra API, etc.).



* + 1. **Export Data from DynamoDB:**

Use AWS Data Pipeline, AWS Glue, or the AWS CLI to export your data from DynamoDB.

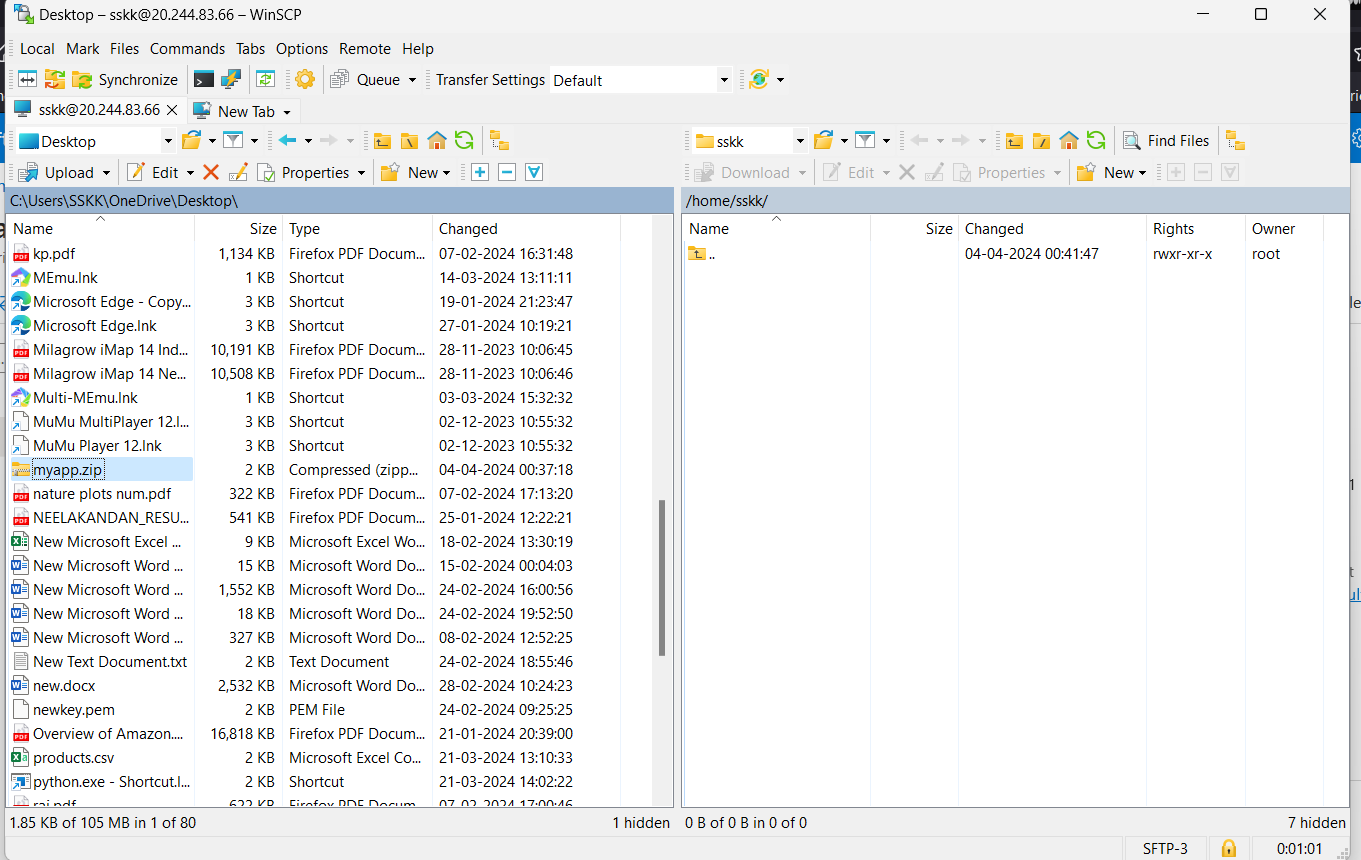


Export the data in a compatible format (e.g., JSON) that can be imported into Azure Cosmos DB.

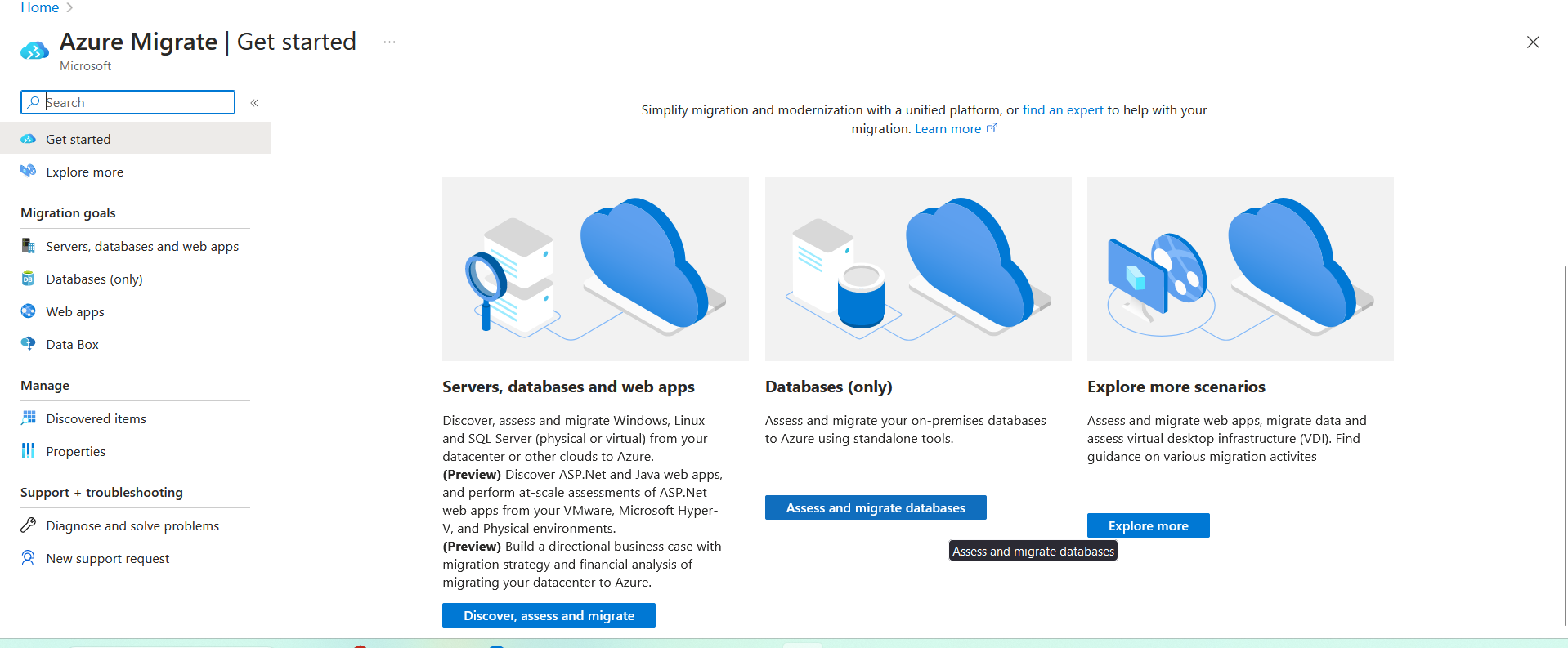
* + 1. **Import Data into Azure Cosmos DB:**

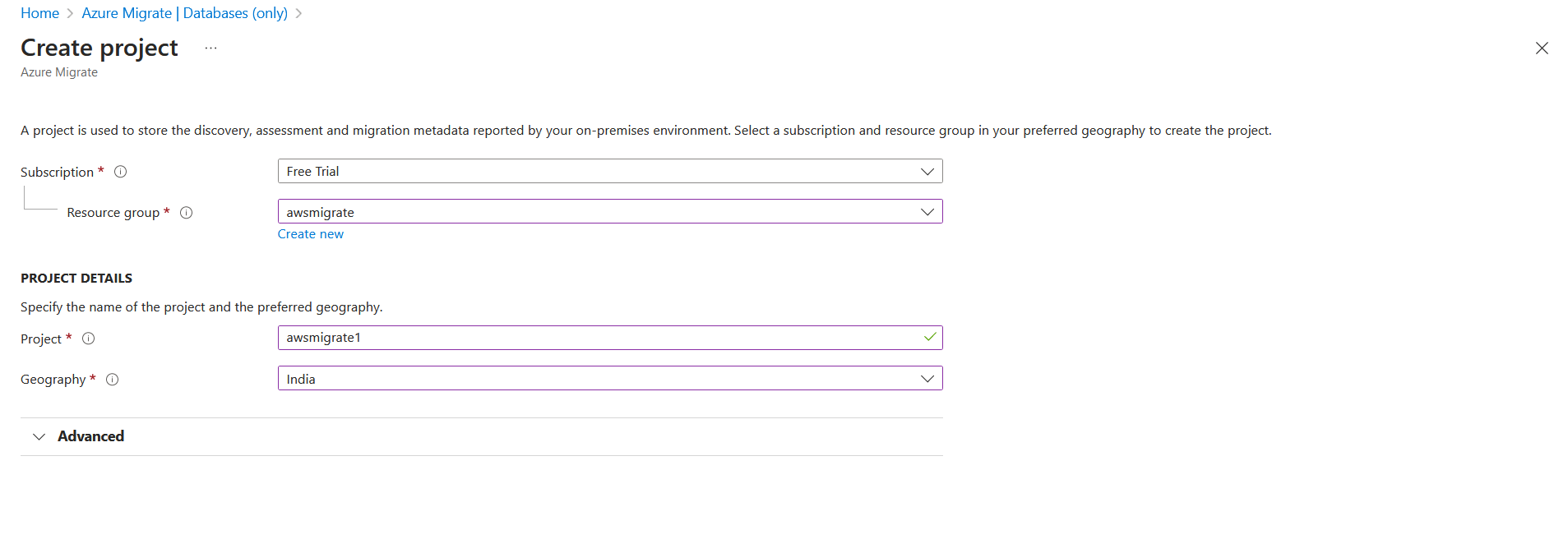
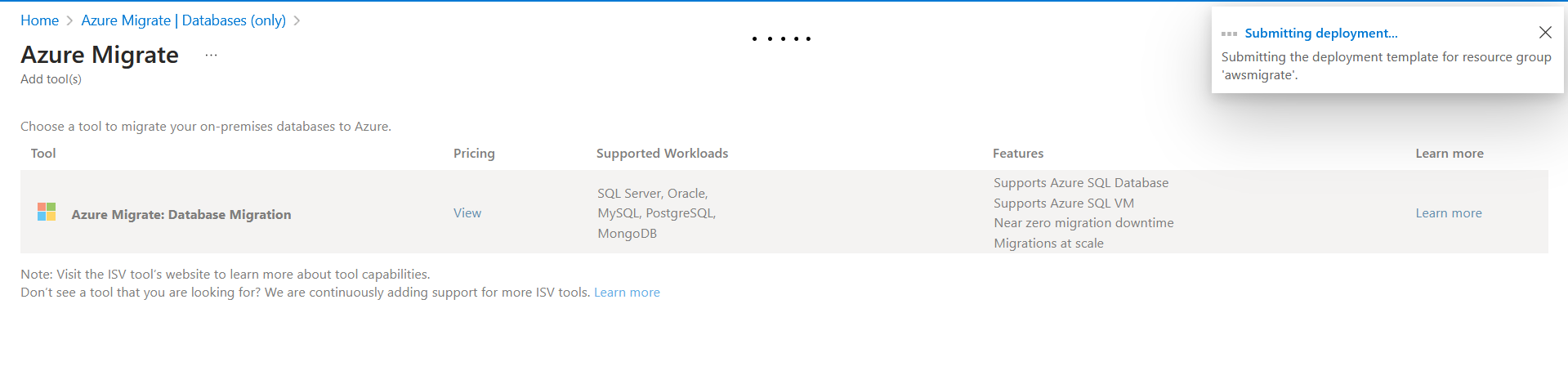
Use Azure Data Factory, Azure CLI, or AzCopy to import the exported data into Azure Cosmos DB.

Follow the documentation for your chosen method to import data efficiently.





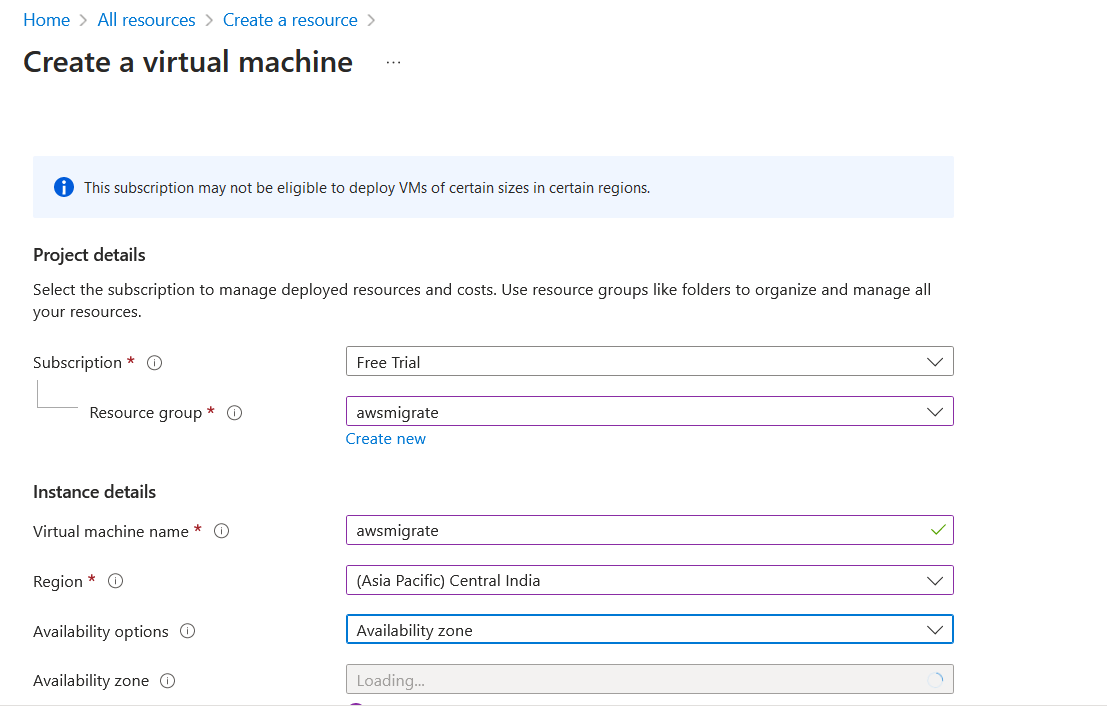


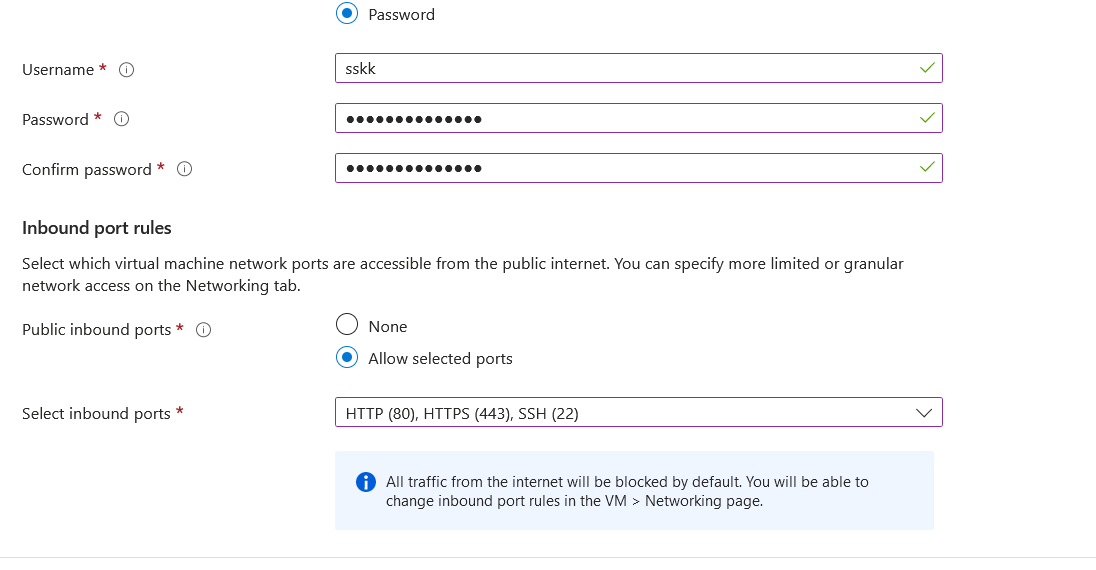


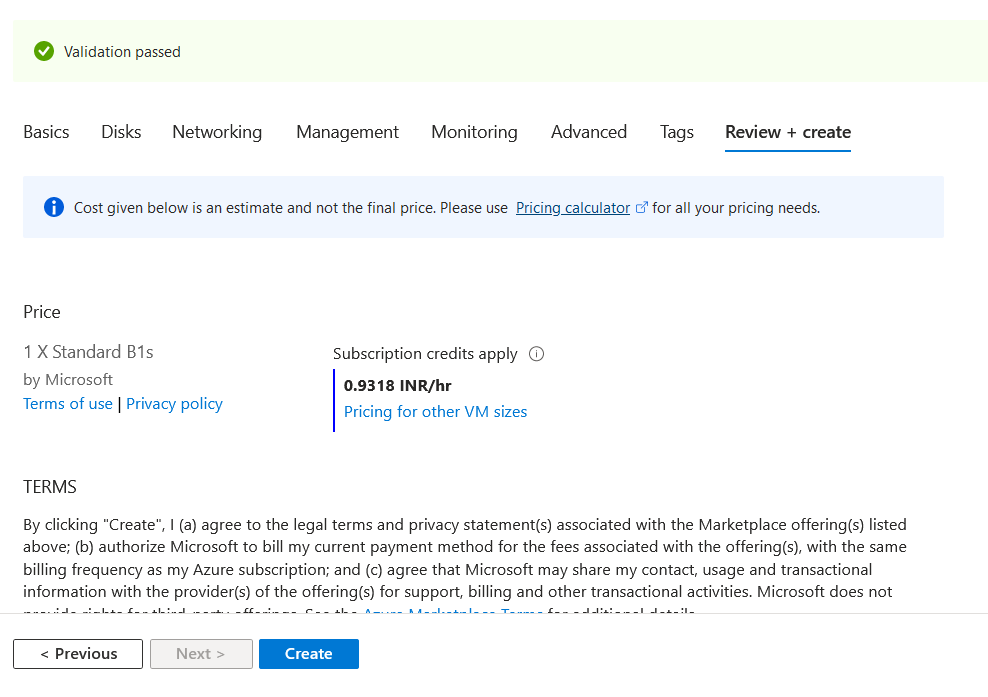
* + 1. **Set Up Azure VM:**

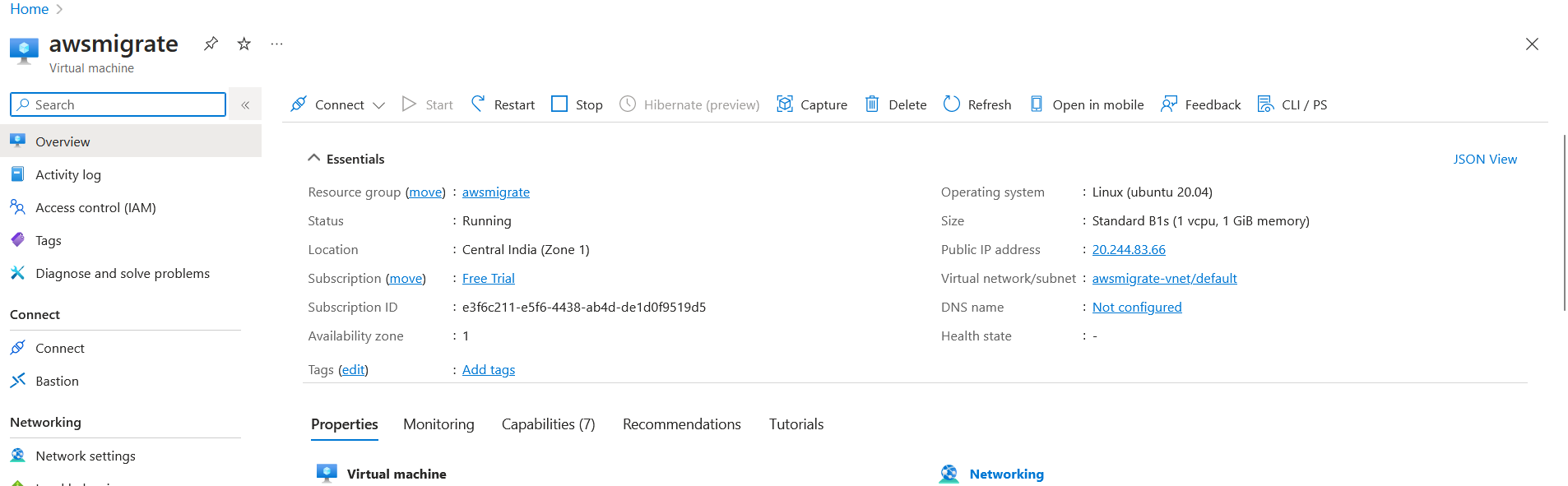
Provision an Azure VM with similar specifications as your AWS VM. Choose the appropriate operating system (e.g., Linux or Windows) and instance size.

Configure networking settings, security groups, and any other necessary configurations on the Azure VM.





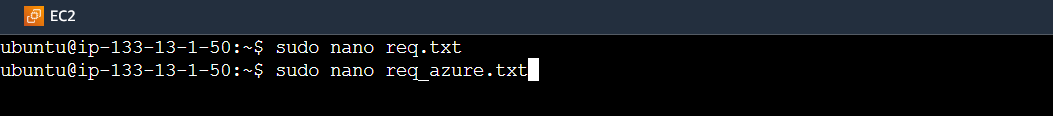


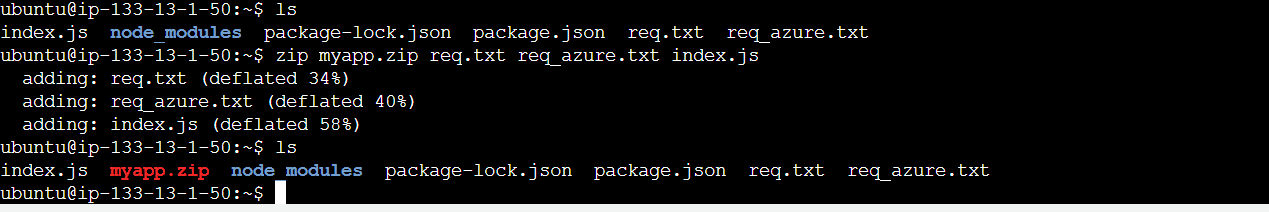


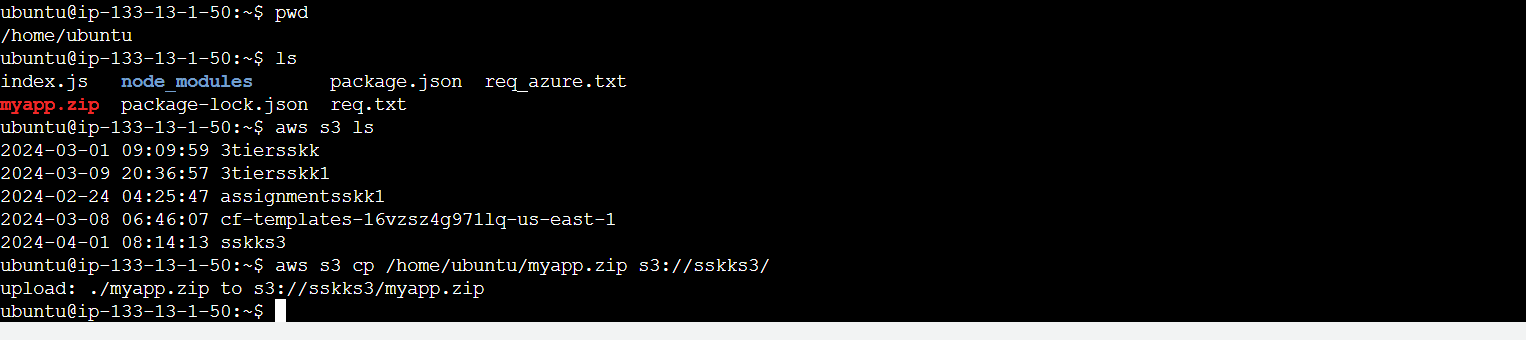
* + 1. **Transfer Application Code and Data:**

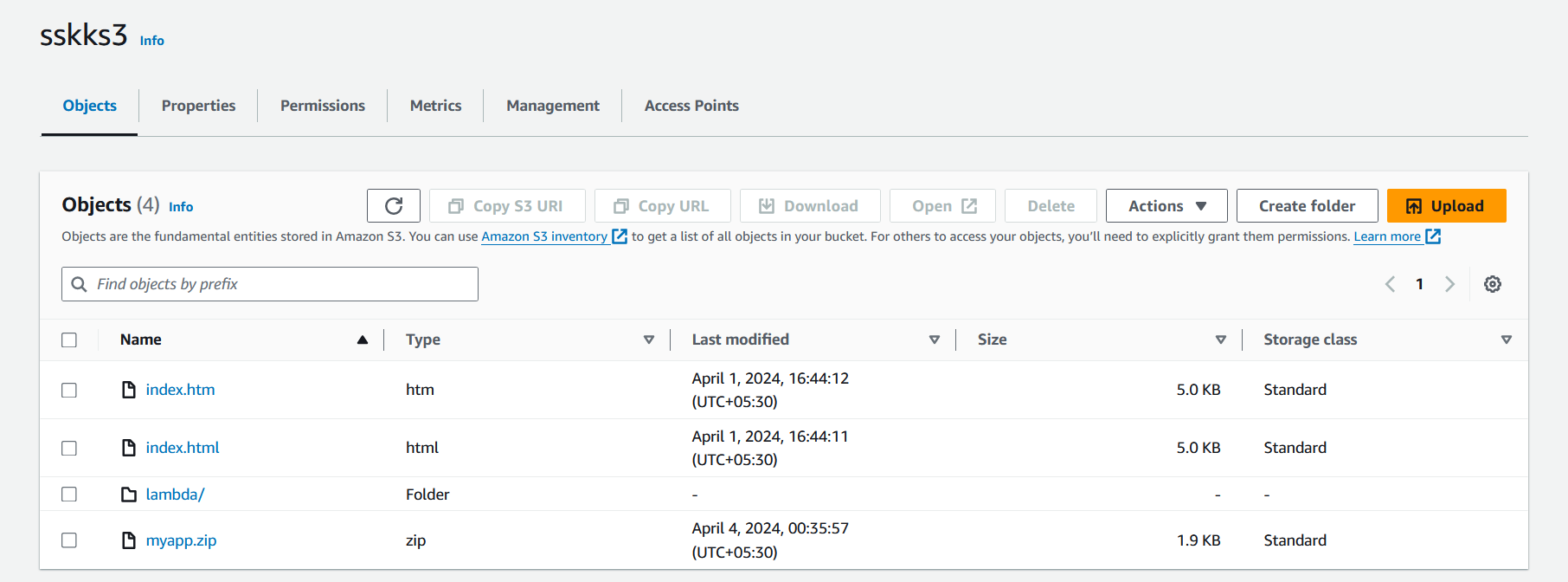
**Codebase**: Copy your web application code from the AWS VM to the Azure VM. You can use tools like SCP (for Linux) or WinSCP (for Windows) to transfer files securely.

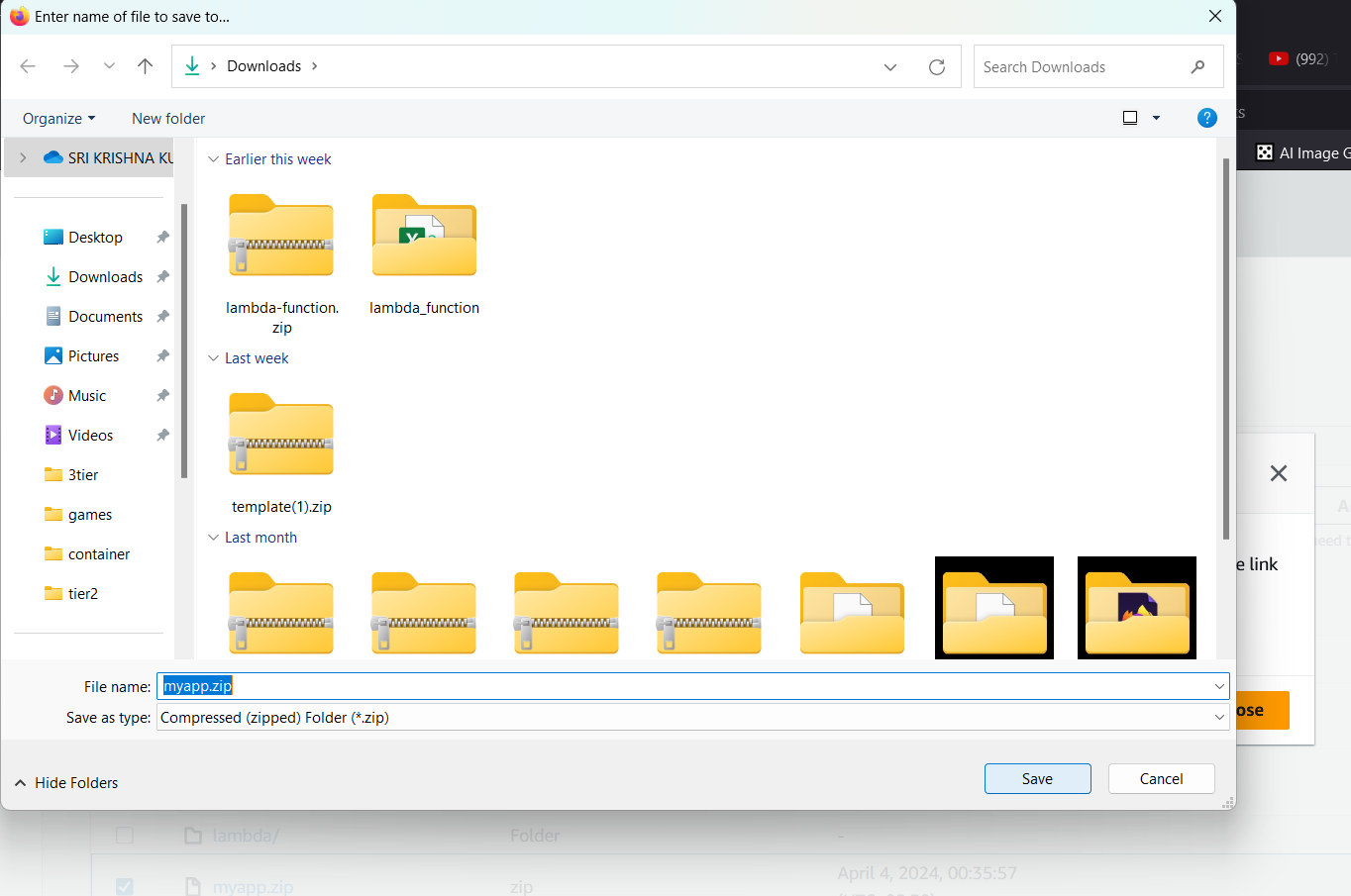
 



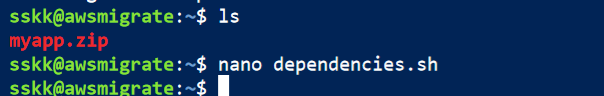






* + 1. **Set Up Dependencies:**

Install any dependencies required for your web application on the Azure VM. This may include web servers (e.g., Apache, Nginx), programming language runtimes (e.g., Node.js, Python, PHP), and any other required software components.

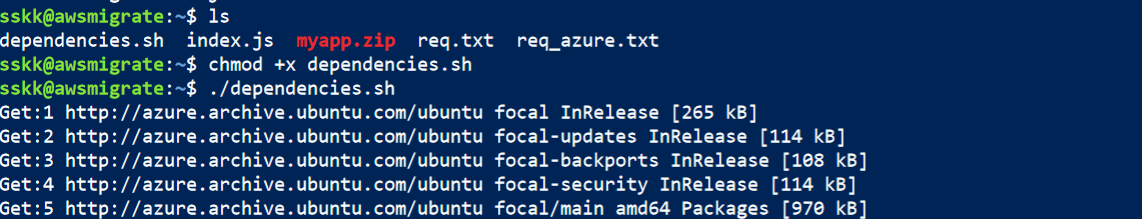
* + 
  + create a script to run these commands line by line.





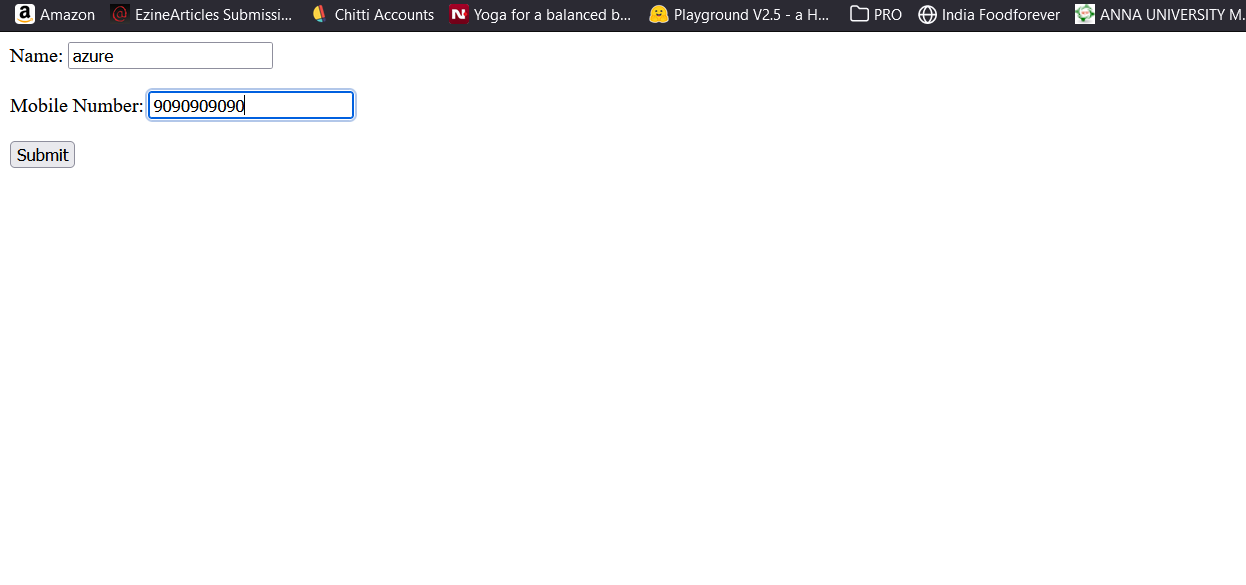
Install unzip and unzip the files from myapp.zip transferred from aws backup

Make sure to give execute permissions to the script and run the script

* + 1. **Configure Web Server and test the site**

Configure the VM to serve your web application. Update the server configuration files to point to the location of your web application code on the Azure VM.



sudo apt update

sudo apt install nodejs npm

npm install express

npm install aws-sdk

npm install body-parser

const express = require('express');

const AWS = require('aws-sdk');

const bodyParser = require('body-parser');

const app = express();

const PORT = process.env.PORT || 3000;

// Configure AWS SDK

AWS.config.update({

region: 'your-dynamodb-region',

accessKeyId: 'your-access-key-id',

secretAccessKey: 'your-secret-access-key'

});

const dynamoDB = new AWS.DynamoDB.DocumentClient();

const tableName = 'UserTable'; // Name of your DynamoDB table

app.use(bodyParser.json());

// Frontend HTML content

const frontendHTML = `

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>User Registration</title>

</head>

<body>

<!-- Your HTML form goes here -->

</body>

</html>

`;

// API endpoint to save user data

app.post('/api/user', (req, res) => {

// Your backend logic to save user data to DynamoDB

});

// Serve frontend HTML

app.get('/', (req, res) => {

res.send(frontendHTML);

});

// Start the server

app.listen(PORT, () => {

console.log(`Server is running on port ${PORT}`);

});

node index.js