**10 Projects AWS**

**Master Core AWS Services and Architectures *with***

**CLI and Console**

These 10 AWS projects include a scenario that demonstrates real-world use cases, making the tasks more practical and relatable.

**1. Create a Static Website Using Amazon S3**

Scenario: You’re tasked with hosting a corporate website with **no** server infrastructure. Your company wants to avoid the complexity of managing servers and is looking for a **cost-effective** solution to deliver static content.

1. Create an S3 bucket to store website files:

**aws s3api create-bucket --bucket** my-static-site-bucket **--region** us-east-1 **--acl** **public**-read

2. Upload website files (HTML, CSS, JS)

**aws s3 sync ./website s3://**my-static-site-bucket/

3. Enable static website hosting on the bucket:

**aws s3 website s3://**my-static-site-bucket/ --index-document index.html --error-document error.html

Console: S3 > Bucket > Properties > **Enable** Static Website Hosting.

Outcome: The website is now publicly available via the S3 bucket **URL**.

**2. Launch and Configure an EC2 Instance**

Scenario: You are deploying a web application that requires a virtual server for backend processing. You will configure an EC2 instance with the required security settings and SSH into it for further configuration.

1. Launch EC2 instance\*\* with the required AMI and instance type:

**aws ec2 run-instances --image-id** ami-0c55b159cbfafe1f0 --instance-**type** t2.micro --**key**-name MyKeyPair --security-group-ids **sg**-0123456789abcdef --**subnet**-id subnet-0abcd123

2. Describe the instance to obtain its **public** **IP**:

**aws ec2 describe-instances** --instance-ids i-0123456789abcdef

3. SSH into the instance using the key pair:

**ssh -i** MyKeyPair.pem **ec2-user@**<instance-public-ip>

Console: EC2 > Launch Instance > Configure Details.

Outcome: A running EC2 instance accessible via SSH to configure the application or install additional software.

**3. Set Up an Application Load Balancer**

Scenario: You need to balance traffic across **multiple** EC2 instances running a web application. Your goal is to ensure high availability and fault tolerance by distributing traffic between instances.

1. Create security group for the ALB:

**aws ec2 create-security-group** **--group-name** MyALBSG **--description** "Security group for ALB"

2. Create ALB in the specified subnets:

**aws** **elbv2 create-load-balancer** **--name** my-load-balancer **--subnets** subnet-abc123 subnet-def456 --security-groups sg-0123456789abcdef

3. Create target group and register instances:

**aws** **elbv2 create-target-group** --name my-targets --protocol HTTP --port 80 --vpc-id vpc-0abcd1234

aws **elbv2 register-targets** --target-group-arn arn:aws:elasticloadbalancing:us-east-1:123456789012:targetgroup/my-targets/12345678 --targets Id=i-0123456789abcdef

Console: EC2 > Load Balancers > Create Load Balancer.

Outcome: The ALB is **distributing** traffic evenly across your EC2 instances, ensuring scalability and fault tolerance.

**4. Implement Auto Scaling**

Scenario: Your web application experiences variable **traffic**, and you need to ensure that the system scales **dynamically**, maintaining performance during peak times and saving **costs** during low traffic periods.

1. Create a launch configuration for EC2 instances:

aws **autoscaling create-launch-configuration** --launch-configuration-name my-config --image-id ami-0c55b159cbfafe1f0 --instance-type t2.micro

2. Create an Auto Scaling group:

**aws autoscaling create-auto-scaling-group** --auto-scaling-group-name my-asg --launch-configuration-name my-config --**min-size 1 --max-size 5 --desired-capacity 2** --vpc-zone-identifier subnet-abc123

3. Set up scaling policies based on **CloudWatch** alarms:

aws autoscaling put-scaling-policy --auto-scaling-group-name my-asg --policy-name my-scale-out-policy --scaling-adjustment 1 --adjustment-type ChangeInCapacity

Console: **EC2 > Auto Scaling > Create Auto Scaling Group**.

Outcome: The system automatically **scales** based on **traffic**, ensuring optimal resource usage.

**5. Create a VPC with Public and Private Subnets**

Scenario: You are tasked with setting up an isolated network for a **multi-tier** application, separating public and private layers. The frontend should be in the **public** subnet, while the backend database and services reside in the private subnet.

1. Create a VPC:

aws **ec2 create-vpc** **--cidr-block** 10.0.0.0/**16**

2. Create a public subnet:

aws **ec2 create-subnet** --vpc-id vpc-0abcd1234 --cidr-block 10.0.1.0/**24** --availability-zone us-east-1a

3. Create a private subnet:

aws **ec2 create-subnet** --vpc-id vpc-0abcd1234 --cidr-block 10.0.2.0/24 --availability-zone us-east-1a

Console: **VPC > Create VPC > Create Subnets**.

Outcome: The VPC is segregated into public and private subnets for better security and architectural design.

**6. Set Up an Amazon RDS Database**

Scenario: You need to set up a managed relational database for a web application. The database should be highly available, and maintenance tasks like backups and updates should be automated.

1. Create an RDS instance:

aws **rds create-db-instance** --db-instance-**identifier** mydb --db-instance-class **db.t2.micro** --**engine** mysql --allocated-storage 20 --master-username **admin** --master-user-password mypassword

2. Connect to the RDS instance using MySQL client:

**mysql -h** <**rds-endpoint**> **-u admin** **-p**

Console: **RDS > Create Database > MySQL**.

Outcome: A managed MySQL database is running, and connections can be established from the application.

**7. Implement an S3 Lifecycle Policy**

Scenario: You need to optimize storage costs for a data archiving system. Old files should automatically transition to cheaper storage tiers, and very old data should be deleted after a specified time.

1. Create a lifecycle **policy** in JSON:

**vim** lifecycle.json

{

"**Rules**": [

{

"ID": "MoveToIA",

"Prefix": "",

"Status": "Enabled",

"Transitions": [

{

"Days": **30**,

"StorageClass": "**STANDARD\_IA**"

}

],

"Expiration": {

"Days": **365**

}

}

]

}

2. Apply the policy:

**aws s3api put-bucket-lifecycle-configuration --bucket** my-bucket **--lifecycle-configuration file://**lifecycle.json

Console: **S3 > Bucket > Management > Lifecycle**.

Outcome: Data in S3 **transitions** to lower-cost storage after 30 days and is eventually deleted (after 365) based on the policy.

**8. Set Up CloudFront Distribution**

Scenario: You need to deliver a global website with low latency, using cached static content. The goal is to ensure quick response times for users worldwide by leveraging CloudFront’s CDN.

1. Create CloudFront distribution with **S3** as the **origin**:

**aws cloudfront create-distribution --origin-domain-name** my-static-site-bucket.s3.amazonaws.com

Console: **CloudFront > Create Distribution > Configure Origin.**

Outcome: The content is now **distributed** **globally** with **lower** latency through CloudFront’s edge locations.

**9. Implement IAM Roles and Policies**

Scenario: You need to secure access to an S3 bucket for an application running on EC2, ensuring the EC2 instance has the right **permissions** without using hardcoded credentials.

1. Create an IAM role with S3 **read-**only access:

aws **iam create-role** --role-name MyEC2Role **--assume-role**-policy-document file://trust-policy.json

aws iam **attach**-role-policy --role-name MyEC2Role --policy-arn arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess

2. Assign the role to an EC2 instance:

aws **ec2 associate-iam-instance-profile --instance-id** i-0123456789abcdef --iam-instance-profile Name=MyEC2Role

Console: **IAM > Roles > Create Role > EC2.**

Outcome: The EC2 instance can now securely access S3 resources using the attached IAM role.

**10. Set Up CloudWatch Alarms and Logs**

Scenario: Your team needs to monitor application performance and resource utilization, ensuring that immediate action is taken if any **metrics** exceed defined **thresholds**.

1. Create a CloudWatch alarm for high **CPU utilization**:

aws cloudwatch put-metric-alarm --alarm-name HighCPUAlarm --metric-name CPUUtilization --namespace AWS/EC2 --statistic Average --period **300** --threshold **80** --comparison-operator **GreaterThanOrEqualToThreshold** --dimensions "Name=InstanceId,Value=i-0123456789abcdef" --evaluation-periods 2 --alarm-actions arn:aws:sns:us-east-1:123456789012:MySNSTopic

**Console: CloudWatch > Alarms > Create Alarm.**

Outcome: CloudWatch monitors the instance’s performance and triggers an alert when the CPU usage exceeds the defined threshold.

**Conclusion**

Certainly, these 10 projects will provide the power of learning AWS resources pragmatic way! Your **job searching will be easier lots, NOW!!**