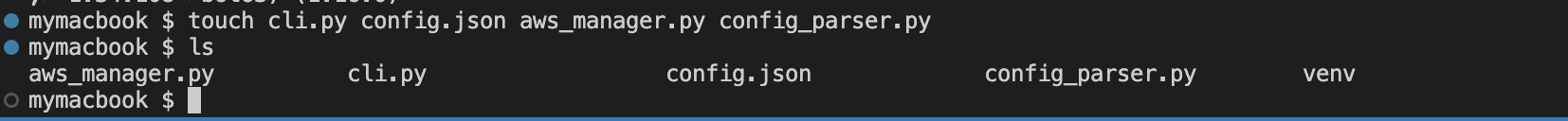
**Setup:**

-Initiated a virtual python venv

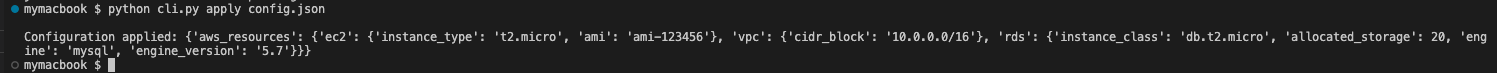
-Installing required libraries like click

-Created a basic project structure



-Just like my work earlier, with python script, I am implementing CLI using cli.py file with required imports like json, click

-Copied boiler plate code in my 3 files **cli.py, config\_parser.py and aws\_manager.py** and **config.json** (I have left placeholders as required)

-Tested basic JSON parsing and successful  


This output indicates that my CLI is currently able to read the configuration from the JSON file and pass it through my system.

**Developing**

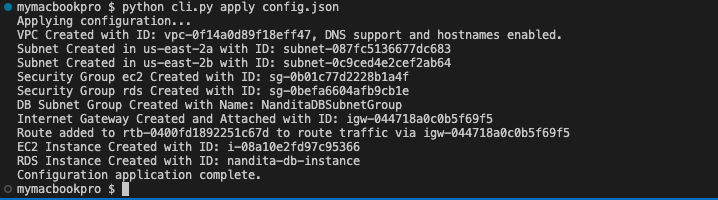
In the nandita\_aws\_manager.py file, I would like to separate all my aws related function definitions. For example **defining** vpc, subnets, sg, ec2, rds and also a definition to apply our configurations.

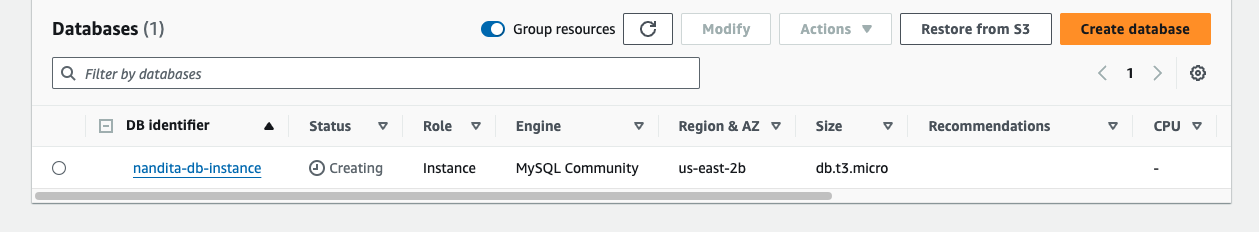
The above will keep our cli.py main file near and clear. As we are just importing the function definitions from nandita\_aws\_manager.py file

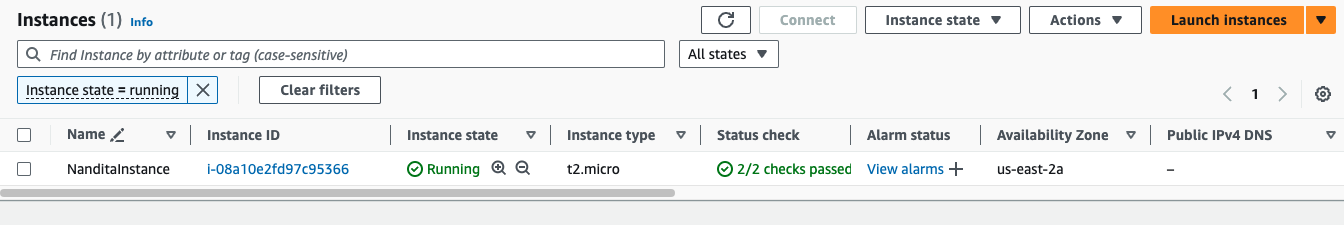
*My approach to building nandita\_aws\_manager.py file to apply configurations:*  
  
As mentioned earlier, write functions using that reflects the sequence of operations that generally would happen in an AWS environment

* A function for creating VPC
* A function for creating subnet
* An individual function for rest of the networking elements like an Internet Gateway, Route Table Updates, Security Groups, DB Subnet Group
* Functions for EC2 and RDS creation
* Finally, my favorite function apply\_configuration to APPLY !

My approach functions in nandita\_aws\_manager.py,

* Use individual functions for each resource specification for better code readability
* **Error handling**: I have added error handling for most error prone functions with the free tier, such as VPC creation function, db subnet group function and rds creation function, (will do more in phase 2)
* 





Phase 1: We now have,

* config.json, where a user can specify resource specifications
* nandita\_config\_parser.py, a file that parses above json specifications
* Nandita\_aws\_manager.py, a py file that interacts with aws by written functions
* cli.py that applies configuration and creates resources in aws

Phase 2: Since we have our cli.py successfully applying configurations, here I have focused on adding error messages to the rest of the functions. We can use the retry wrapper, but I think it would be an overkill. Focusing on clear error messages without adding retry logic will simplify the implementation and keep the error handling straightforward.

Dev best practices followed in this application:

* Function Naming

All my function names are descriptive and follow the Python naming convention of using lowercase with words separated by underscores (snake\_case). This makes the code more readable and easier to understand.

* Error handling  
  I have implemented error handling using try-except blocks in several functions, such as create\_vpc, create\_db\_subnet\_group, and create\_rds\_instance. This is a good practice as it helps catch and handle exceptions gracefully, preventing the program from crashing unexpectedly.
* Modular code  
  Each function has a specific responsibility, making the code more modular and easier to maintain. For example, create\_vpc, create\_subnets, create\_security\_group, and create\_ec2\_instance each handle a different aspect of the AWS infrastructure setup
* Meaningful print statements

The print statements provide useful information about the actions being performed and the resources being created.

* Consistent formatting

I have made the code follow a consistent formatting style, making it easier to read and understand. This includes proper indentation, spacing, and line breaks.

* Descriptive variable names

The variable names used in the code are descriptive and self-explanatory

* Handling the configuration

Our code handles configuration data from config.json, which is a good practice for separating configuration from the application logic. Also one of the specifications from our assignment objectives

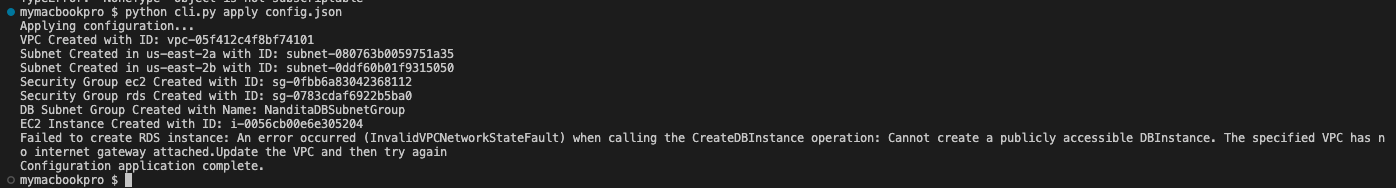
My usual challenges: Maximum vpc limit reached  


Subnet error:

botocore.errorfactory.DBSubnetGroupDoesNotCoverEnoughAZs: An error occurred (DBSubnetGroupDoesNotCoverEnoughAZs) when calling the CreateDBSubnetGroup operation: The DB subnet group doesn't meet Availability Zone (AZ) coverage requirement. Current AZ coverage: us-east-2b. Add subnets to cover at least 2 AZs.

(include - screenshot)

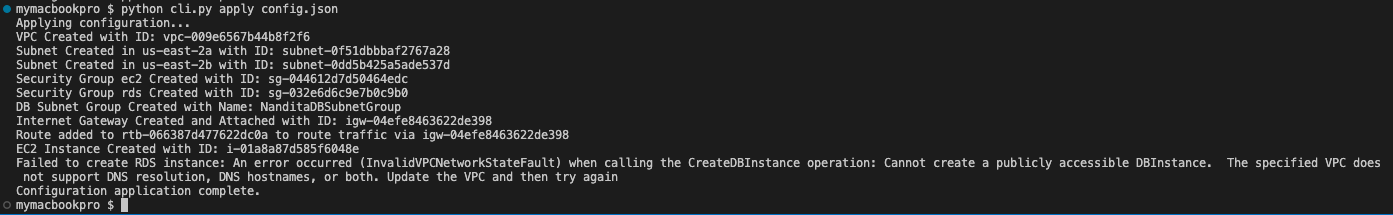
Fixed by changing the code to add

Error 3:  


“Failed to create RDS instance: An error occurred (InvalidVPCNetworkStateFault) when calling the CreateDBInstance operation: Cannot create a publicly accessible DBInstance. The specified VPC has no internet gateway attached.Update the VPC and then try again

Configuration application complete.”

Error 4:



“Failed to create RDS instance: An error occurred (InvalidVPCNetworkStateFault) when calling the CreateDBInstance operation: Cannot create a publicly accessible DBInstance. The specified VPC does not support DNS resolution, DNS hostnames, or both. Update the VPC and then try again

”

Resolved by adding,  
EnableDnsHostnames={'Value': True}

EnableDnsSupport={'Value': True}

In the script while creating vpc (create\_vpc function)

Docs used:  
<https://boto3.amazonaws.com/v1/documentation/api/latest/index.html>

<https://github.com/aws-samples>

<https://stackoverflow.com/>

<https://forums.aws.amazon.com/>

And got few boiler plate function definitions, validations from chatgpt 3.5