**Lab Exercise 4- Creating Separate Flows with Different Decorators in Metaflow**

**Objective**

By the end of this exercise, you will have:

1. Created multiple separate flows, each using a different Metaflow decorator.
2. Observed how these decorators influence the flow's behavior.
3. Learned to combine and configure decorators for various use cases.

**Part 1: Flow with @batch Decorator**

The @batch decorator runs a step in AWS Batch, allowing you to execute steps in the cloud. This is useful when you need more powerful compute resources than your local environment.

**Step 1: Create the BatchFlow**

from metaflow import FlowSpec, step, batch

class BatchFlow(FlowSpec):

@step

def start(self):

print("Starting BatchFlow")

self.numbers = list(range(10))

self.next(self.process)

@batch(cpu=2, memory=4096) # Request 2 CPUs and 4GB of memory in AWS Batch

@step

def process(self):

self.squares = [x\*\*2 for x in self.numbers]

print(f"Processed numbers into squares: {self.squares}")

self.next(self.end)

@step

def end(self):

print("BatchFlow completed")

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

BatchFlow()

**Task:**

1. Run the BatchFlow using:

python batch\_flow.py run

1. If AWS Batch is not configured, this flow will run locally but print a warning that @batch was ignored.

**Part 2: Flow with @retry Decorator**

The @retry decorator retries a step if it fails due to an exception, ensuring that temporary issues do not break the flow.

**Step 2: Create the RetryFlow**

from metaflow import FlowSpec, step, retry  
import random  
  
  
class RetryFlowNoException(FlowSpec):  
  
 @step  
 def start(self):  
 *"""  
 The start step initializes the counter.  
 """* self.retry\_count = 0  
 print("Starting RetryFlow with conditional retry logic")  
 self.next(self.process)  
  
 @retry(times=3, minutes\_between\_retries=1) *# Retry up to 3 times* @step  
 def process(self):  
 *"""  
 The process step where we simulate a condition for failure.  
 Instead of raising an exception, we retry based on a condition.  
 """* self.retry\_count += 1  
 print(f"Processing attempt {self.retry\_count}...")  
  
 *# Simulate a condition that fails randomly (like a network call)* self.random\_value = random.randint(1, 5) *# Get a random number between 1 and 5* print(f"Random value generated: {self.random\_value}")  
  
 *# If random value is not 5, we treat it as a failure and allow retry* if self.random\_value != 5:  
 print("Condition not met, allowing retry...")  
 *# No need to call self.next() here; just return to let @retry handle it* return  
  
 *# If condition is met, transition to the end step* print("Condition met! Moving to end step.")  
 self.next(self.end) *# This line should be reached only if the condition is met* @step  
 def end(self):  
 *"""  
 The end step which will only be reached if the condition is met.  
 """* print(f"RetryFlow completed successfully after {self.retry\_count} attempts.")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 RetryFlowNoException()

**Task:**

1. Run the RetryFlow using:

python retry\_flow.py run

1. Observe how the flow retries the process step after the simulated failure.

**Part 3: Flow with @catch Decorator**

The @catch decorator handles exceptions in a step and allows the flow to continue without crashing.

**Step 3: Create the CatchFlow**

from metaflow import FlowSpec, step, catch  
  
class CatchFlow(FlowSpec):  
  
 @step  
 def start(self):  
 print("Starting CatchFlow")  
 self.next(self.process)  
  
 @catch(var='error\_info') *# Capture errors in this step* @step  
 def process(self):  
 print("Processing...")  
 raise ValueError("Simulated error in process step")  
 self.next(self.end)  
  
 @step  
 def end(self):  
 if hasattr(self, 'error\_info'):  
 print(f"Error caught: {self.error\_info}")  
 else:  
 print("CatchFlow completed successfully")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 CatchFlow()

**Task:**

1. Run the CatchFlow using:

python catch\_flow.py run

1. Observe how the error in the process step is caught and handled gracefully by the @catch decorator.

**Part 4: Flow with @timeout Decorator**

The @timeout decorator specifies a maximum runtime for a step, and if the step exceeds this time, it will be terminated.

**Step 4: Create the TimeoutFlow**

from metaflow import FlowSpec, step, timeout

class TimeoutFlow(FlowSpec):

@step

def start(self):

print("Starting TimeoutFlow")

self.next(self.long\_step)

@timeout(seconds=5) # Step must complete within 5 seconds

@step

def long\_step(self):

import time

print("Simulating a long-running step...")

time.sleep(10) # Sleep for 10 seconds, which will trigger the timeout

self.next(self.end)

@step

def end(self):

print("TimeoutFlow completed")

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

TimeoutFlow()

**Task:**

1. Run the TimeoutFlow using:

python timeout\_flow.py run

1. Observe how the flow times out after 5 seconds and how the step is interrupted.

**Part 5: Flow with @resources Decorator**

The @resources decorator allocates specific resources (CPU, memory, and even GPUs) to a step, which is useful when working with resource-intensive tasks.

**Step 5: Create the ResourcesFlow**

from metaflow import FlowSpec, step, resources

class ResourcesFlow(FlowSpec):

@step

def start(self):

print("Starting ResourcesFlow")

self.numbers = list(range(10))

self.next(self.process)

@resources(cpu=2, memory=2000) # Request 4 CPUs and 16GB of memory

@step

def process(self):

print("Processing numbers with extra resources...")

self.squares = [x\*\*2 for x in self.numbers]

print(f"Processed numbers into squares: {self.squares}")

self.next(self.end)

@step

def end(self):

print("ResourcesFlow completed")

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

ResourcesFlow()

**Task:**

1. Run the ResourcesFlow using:

python resources\_flow.py run

1. Observe the resource allocation (if running in a cloud environment) or run locally to see how resources are configured.

**Conclusion**

By completing these separate flows using different Metaflow decorators, you now understand how to:

1. Use @batch for cloud execution.
2. Use @retry to handle temporary step failures.
3. Use @catch to handle errors without stopping the flow.
4. Use @timeout to set time limits on steps.
5. Use @resources to allocate resources for computational steps.

Each of these decorators provides flexibility and control, making your workflows more robust, scalable, and efficient.