--- test\_init.py ---

import sys, os, logging, pytest, asyncio  
import subprocess  
from unittest.mock import patch, MagicMock  
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))  
from utils.email\_utils import send\_email\_with\_attachments  
from utils.exportUtils import ExportUtils  
from control.BrowserControl import BrowserControl  
from control.AccountControl import AccountControl  
from control.AvailabilityControl import AvailabilityControl  
from control.PriceControl import PriceControl  
from control.BotControl import BotControl  
from DataObjects.AccountDAO import AccountDAO  
from entity.AvailabilityEntity import AvailabilityEntity  
from entity.BrowserEntity import BrowserEntity  
from entity.PriceEntity import PriceEntity  
#pytest -v > test\_results.txt  
#Run this command in the terminal to save the test results to a file  
  
async def run\_monitoring\_loop(control\_object, check\_function, url, date\_str, frequency, iterations=1):  
 """Run the monitoring loop for a control object and execute a check function."""  
 control\_object.is\_monitoring = True  
 results = []  
  
 while control\_object.is\_monitoring and iterations > 0:  
 try:  
 result = await check\_function(url, date\_str)  
 except Exception as e:  
 result = f"Failed to monitor: {str(e)}"  
 logging.info(f"Monitoring Iteration: {result}")  
 results.append(result)  
 iterations -= 1  
 await asyncio.sleep(frequency)  
  
 control\_object.is\_monitoring = False  
 results.append("Monitoring stopped successfully!")  
   
 return results  
  
def setup\_logging():  
 """Set up logging without timestamp and other unnecessary information."""  
 logger = logging.getLogger()  
 if not logger.hasHandlers():  
 logging.basicConfig(level=logging.INFO, format='%(message)s')  
  
  
def save\_test\_results\_to\_file(output\_file="test\_results.txt"):  
 """Helper function to run pytest and save results to a file."""  
 print("Running tests and saving results to file...")  
 output\_path = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)), output\_file)  
 with open(output\_path, 'w') as f:  
 # Use subprocess to call pytest and redirect output to file  
 subprocess.run(['pytest', '-v'], stdout=f, stderr=subprocess.STDOUT)  
   
# Custom fixture for logging test start and end  
@pytest.fixture(autouse=True)  
def log\_test\_start\_end(request):  
 test\_name = request.node.name  
 logging.info(f"------------------------------------------------------\nStarting test: {test\_name}\n")  
   
 # Yield control to the test function  
 yield  
   
 # Log after the test finishes  
 logging.info(f"\nFinished test: {test\_name}\n------------------------------------------------------")  
  
  
@pytest.fixture  
def base\_test\_case():  
 """Base test setup that can be used by all test functions."""  
 test\_case = MagicMock()  
 test\_case.browser\_control = BrowserControl()  
 test\_case.account\_control = AccountControl()  
 test\_case.availability\_control = AvailabilityControl()  
 test\_case.price\_control = PriceControl()  
 test\_case.bot\_control = BotControl()  
 test\_case.account\_dao = AccountDAO()  
 test\_case.availability\_entity = AvailabilityEntity()  
 test\_case.browser\_entity = BrowserEntity()  
 test\_case.price\_entity = PriceEntity()  
 test\_case.email\_dao = send\_email\_with\_attachments  
 test\_case.export\_utils = ExportUtils()  
 return test\_case  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Save the pytest output to a file in the same folder  
 save\_test\_results\_to\_file(output\_file="test\_results.txt")

--- unitTest\_add\_account.py ---

import pytest, os, sys  
from unittest.mock import MagicMock  
from test\_init import setup\_logging, base\_test\_case, save\_test\_results\_to\_file, log\_test\_start\_end, logging  
  
  
setup\_logging() # Initialize logging if needed  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestAccountDAO:  
 @pytest.fixture  
 def account\_dao(self,base\_test\_case, mocker):  
 # Mock the psycopg2 connection and cursor  
 mocker.patch('psycopg2.connect')  
 account\_dao = base\_test\_case.account\_dao  
 account\_dao.connection = MagicMock()  
 account\_dao.cursor = MagicMock()  
 logging.info("Fake database connection established")  
 return account\_dao  
  
 def test\_entity\_add\_account\_success(self, account\_dao):  
 # Setup the cursor's behavior for successful insertion  
 account\_dao.cursor.execute = MagicMock()  
 account\_dao.cursor.rowcount = 1  
 account\_dao.connection.commit = MagicMock()  
   
 # Test the add\_account method for success  
 result = account\_dao.add\_account("test\_user", "password123", "example.com")  
   
 # Log the result of the operation  
 logging.info(f"AccountDAO.add\_account returned {result}")  
 logging.info("Expected result: True")  
  
 # Assert and log the final outcome  
 assert result == True, "Account should be added successfully"  
 logging.info("Test add\_account\_success passed")  
  
 def test\_entity\_add\_account\_fail(self, account\_dao):  
 # Setup the cursor's behavior to simulate a failure during insertion  
 account\_dao.cursor.execute.side\_effect = Exception("Database error")  
 account\_dao.cursor.rowcount = 0  
 account\_dao.connection.commit = MagicMock()  
  
 # Perform the test  
 result = account\_dao.add\_account("fail\_user", "fail123", "fail.com")  
   
 # Log the result of the operation  
 logging.info(f"AccountDAO.add\_account returned {result}")  
 logging.info("Expected result: False")  
   
 # Assert and log the final outcome  
 assert result == False, "Account should not be added"  
 logging.info("Test add\_account\_fail passed")  
  
  
  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestAccountControl:  
 @pytest.fixture  
 def account\_control(self, base\_test\_case, mocker):  
 # Get the mocked AccountControl from base\_test\_case  
 account\_control = base\_test\_case.account\_control  
 account\_control.account\_dao = MagicMock(spec=base\_test\_case.account\_dao)  
   
 # Mock methods used in the control layer's add\_account  
 mocker.patch.object(account\_control.account\_dao, 'connect')  
 mocker.patch.object(account\_control.account\_dao, 'close')  
 logging.info("Mocked AccountDAO connection and close methods")  
 return account\_control  
  
 def test\_control\_add\_account\_success(self, account\_control):  
 # Mock successful addition in the DAO layer  
 account\_control.account\_dao.add\_account.return\_value = True  
   
 # Call the control method and check the response  
 result = account\_control.add\_account("test\_user", "password123", "example.com")  
 expected\_message = "Account for example.com added successfully."  
   
 # Log the response and expectations  
 logging.info(f"Control method add\_account returned: '{result}'")  
 logging.info("Expected message: 'Account for example.com added successfully.'")  
  
 assert result == expected\_message, "The success message should match expected output"  
 logging.info("Test control\_add\_account\_success passed")  
  
 def test\_control\_add\_account\_fail(self, account\_control):  
 # Mock failure in the DAO layer  
 account\_control.account\_dao.add\_account.return\_value = False  
   
 # Call the control method and check the response  
 result = account\_control.add\_account("fail\_user", "fail123", "fail.com")  
 expected\_message = "Failed to add account for fail.com."  
   
 # Log the response and expectations  
 logging.info(f"Control method add\_account returned: '{result}'")  
 logging.info("Expected message: 'Failed to add account for fail.com.'")  
  
 assert result == expected\_message, "The failure message should match expected output"  
 logging.info("Test control\_add\_account\_fail passed")  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_]) # Run pytest directly

--- unitTest\_check\_availability.py ---

import pytest, logging  
from unittest.mock import patch  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
# Test for successful availability check (Control and Entity Layers)  
async def test\_check\_availability\_success(base\_test\_case):  
 with patch('entity.AvailabilityEntity.AvailabilityEntity.check\_availability') as mock\_check:  
 url = "https://example.com"  
 mock\_check.return\_value = f"Selected or default date current date is available for booking."  
 expected\_entity\_result = f"Selected or default date current date is available for booking."  
 expected\_control\_result = f"Checked availability: Selected or default date current date is available for booking."  
  
 # Execute the command  
 result = await base\_test\_case.availability\_control.receive\_command("check\_availability", url)  
  
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Expected: {expected\_entity\_result}")  
 logging.info(f"Entity Layer Received: {mock\_check.return\_value}")  
 assert mock\_check.return\_value == expected\_entity\_result, "Entity layer assertion failed."  
 logging.info("Unit Test Passed for entity layer.\n")  
  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
# Test for failure in entity layer (Control should handle it gracefully)  
async def test\_check\_availability\_failure\_entity(base\_test\_case):  
 with patch('entity.AvailabilityEntity.AvailabilityEntity.check\_availability', side\_effect=Exception("Failed to check availability")) as mock\_check:  
 url = "https://example.com"  
 expected\_control\_result = "Failed to check availability: Failed to check availability"  
  
 # Execute the command  
 result = await base\_test\_case.availability\_control.receive\_command("check\_availability", url)  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer failed to handle entity error correctly."  
 logging.info("Unit Test Passed for entity layer error handling.")  
  
# Test for no availability scenario (control and entity)  
async def test\_check\_availability\_no\_availability(base\_test\_case):  
 with patch('entity.AvailabilityEntity.AvailabilityEntity.check\_availability') as mock\_check:  
 url = "https://example.com"  
 mock\_check.return\_value = "No availability for the selected date."  
 expected\_control\_result = "Checked availability: No availability for the selected date."  
  
 # Execute the command  
 result = await base\_test\_case.availability\_control.receive\_command("check\_availability", url)  
  
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Received: {mock\_check.return\_value}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer failed to handle no availability scenario."  
 logging.info("Unit Test Passed for control layer no availability handling.")  
  
# Test for control layer failure scenario  
async def test\_check\_availability\_failure\_control(base\_test\_case):  
 with patch('control.AvailabilityControl.AvailabilityControl.receive\_command', side\_effect=Exception("Control Layer Failed")) as mock\_control:  
 url = "https://example.com"  
 expected\_control\_result = "Control Layer Exception: Control Layer Failed"  
  
 # Execute the command and catch the raised exception  
 try:  
 result = await base\_test\_case.availability\_control.receive\_command("check\_availability", url)  
 except Exception as e:  
 result = f"Control Layer Exception: {str(e)}"  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer failure.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_close\_browser.py ---

import pytest, logging  
from unittest.mock import patch  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
async def test\_close\_browser\_success(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.close\_browser') as mock\_close:  
 # Set up mock and expected outcomes  
 mock\_close.return\_value = "Browser closed."  
 expected\_entity\_result = "Browser closed."  
 expected\_control\_result = "Control Object Result: Browser closed."  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("close\_browser")  
   
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Expected: {expected\_entity\_result}")  
 logging.info(f"Entity Layer Received: {mock\_close.return\_value}")  
 assert mock\_close.return\_value == expected\_entity\_result, "Entity layer assertion failed."  
 logging.info("Unit Test Passed for entity layer.\n")  
   
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
async def test\_close\_browser\_not\_open(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.close\_browser') as mock\_close:  
 # Set up mock and expected outcomes  
 mock\_close.return\_value = "No browser is currently open."  
 expected\_entity\_result = "No browser is currently open."  
 expected\_control\_result = "Control Object Result: No browser is currently open."  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("close\_browser")  
   
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Expected: {expected\_entity\_result}")  
 logging.info(f"Entity Layer Received: {mock\_close.return\_value}")  
 assert mock\_close.return\_value == expected\_entity\_result, "Entity layer assertion failed."  
 logging.info("Unit Test Passed for entity layer.\n")  
   
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
async def test\_close\_browser\_failure\_control(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.close\_browser', side\_effect=Exception("Unexpected error")) as mock\_close:  
 # Set up expected outcome  
 expected\_result = "Control Layer Exception: Unexpected error"  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("close\_browser")  
   
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected to Report: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer failed to handle or report the error correctly."  
 logging.info("Unit Test Passed for control layer error handling.")  
  
async def test\_close\_browser\_failure\_entity(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.close\_browser', side\_effect=Exception("BrowserEntity\_Failed to close browser: Internal error")) as mock\_close:  
 # Set up expected outcome  
 internal\_error\_message = "BrowserEntity\_Failed to close browser: Internal error"  
 expected\_control\_result = f"Control Layer Exception: {internal\_error\_message}"  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("close\_browser")  
   
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Expected Failure: {internal\_error\_message}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer failed to report entity error correctly."  
 logging.info("Unit Test Passed for entity layer error handling.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_delete\_account.py ---

import pytest, os, sys  
from unittest.mock import MagicMock  
from test\_init import setup\_logging, base\_test\_case, save\_test\_results\_to\_file, log\_test\_start\_end, logging  
  
  
setup\_logging() # Initialize logging if needed  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestAccountDAO:  
 @pytest.fixture  
 def account\_dao(self, base\_test\_case, mocker):  
 # Mock the psycopg2 connection and cursor  
 mocker.patch('psycopg2.connect')  
 account\_dao = base\_test\_case.account\_dao  
 account\_dao.connection = MagicMock()  
 account\_dao.cursor = MagicMock()  
 logging.info("Fake database connection established")  
 return account\_dao  
  
 def test\_entity\_delete\_account\_success(self, account\_dao):  
 # Setup the cursor's behavior for successful deletion  
 account\_dao.cursor.execute = MagicMock()  
 account\_dao.cursor.rowcount = 1  
 account\_dao.connection.commit = MagicMock()  
   
 # Test the delete\_account method for success  
 result = account\_dao.delete\_account(1)  
   
 # Log the result of the operation  
 logging.info(f"AccountDAO.delete\_account returned {result}")  
 logging.info("Expected result: True")  
  
 # Assert and log the final outcome  
 assert result == True, "Account should be deleted successfully"  
 logging.info("Test delete\_account\_success passed")  
  
 def test\_entity\_delete\_account\_fail(self, account\_dao):  
 # Setup the cursor's behavior to simulate a failure during deletion  
 account\_dao.cursor.execute.side\_effect = Exception("Database error")  
 account\_dao.cursor.rowcount = 0  
 account\_dao.connection.commit = MagicMock()  
  
 # Perform the test  
 result = account\_dao.delete\_account(9999)  
   
 # Log the result of the operation  
 logging.info(f"AccountDAO.delete\_account returned {result}")  
 logging.info("Expected result: False")  
   
 # Assert and log the final outcome  
 assert result == False, "Account should not be deleted"  
 logging.info("Test delete\_account\_fail passed")  
  
  
  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestAccountControl:  
 @pytest.fixture  
 def account\_control(self, base\_test\_case, mocker):  
 # Get the mocked AccountControl from base\_test\_case  
 account\_control = base\_test\_case.account\_control  
 account\_control.account\_dao = MagicMock(spec=base\_test\_case.account\_dao)  
   
 # Mock methods used in the control layer's delete\_account  
 mocker.patch.object(account\_control.account\_dao, 'connect')  
 mocker.patch.object(account\_control.account\_dao, 'close')  
 logging.info("Mocked AccountDAO connection and close methods")  
 return account\_control  
  
 def test\_control\_delete\_account\_success(self, account\_control):  
 # Mock successful deletion in the DAO layer  
 account\_control.account\_dao.delete\_account.return\_value = True  
   
 # Call the control method and check the response  
 result = account\_control.delete\_account(1)  
 expected\_message = "Account with ID 1 deleted successfully."  
   
 # Log the response and expectations  
 logging.info(f"Control method delete\_account returned: '{result}'")  
 logging.info("Expected message: 'Account with ID 1 deleted successfully.'")  
  
 assert result == expected\_message, "The success message should match expected output"  
 logging.info("Test control\_delete\_account\_success passed")  
  
 def test\_control\_delete\_account\_fail(self, account\_control):  
 # Mock failure in the DAO layer  
 account\_control.account\_dao.delete\_account.return\_value = False  
   
 # Call the control method and check the response  
 result = account\_control.delete\_account(9999)  
 expected\_message = "Failed to delete account with ID 9999."  
   
 # Log the response and expectations  
 logging.info(f"Control method delete\_account returned: '{result}'")  
 logging.info("Expected message: 'Failed to delete account with ID 9999.'")  
  
 assert result == expected\_message, "The failure message should match expected output"  
 logging.info("Test control\_delete\_account\_fail passed")  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_]) # Run pytest directly

--- unitTest\_ExportData.py ---

import pandas as pd  
import pytest  
from unittest.mock import MagicMock, patch  
from test\_init import setup\_logging, base\_test\_case, save\_test\_results\_to\_file, log\_test\_start\_end, logging  
  
# Initialize logging  
setup\_logging()  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestExportUtils:  
  
 @pytest.fixture  
 def setup\_mocked\_paths(self, mocker):  
 mocker.patch('os.path.exists', return\_value=False)  
 mocker.patch('os.makedirs') # Mock directory creation  
 mocker.patch('pandas.DataFrame.to\_excel') # Mock the Excel export method  
 mocker.patch('builtins.open', mocker.mock\_open()) # Mock open for HTML writing  
 logging.info("Mocks for os.path, os.makedirs, pandas.to\_excel, and open set up successfully.")  
  
 def test\_positive\_html\_export(self, base\_test\_case, setup\_mocked\_paths):  
 # Test positive case for HTML export  
 result = base\_test\_case.export\_utils.export\_to\_html("test\_command", "http://example.com", "Success")  
   
 # Assert and log the result  
 assert "HTML file saved and updated" in result  
 logging.info(f"Result: {result}")  
 logging.info("Test positive HTML export passed successfully.")  
  
 def test\_positive\_excel\_export(self, base\_test\_case, setup\_mocked\_paths):  
 # Mock reading from Excel and test positive case for Excel export  
 with patch('pandas.read\_excel', return\_value=pd.DataFrame(columns=["Timestamp", "Command", "URL", "Result", "Entered Date", "Entered Time"])):  
 result = base\_test\_case.export\_utils.log\_to\_excel("test\_command", "http://example.com", "Success")  
   
 # Assert and log the result  
 assert "Data saved to Excel file" in result  
 logging.info(f"Result: {result}")  
 logging.info("Test positive Excel export passed successfully.")  
  
 def test\_negative\_html\_export(self, base\_test\_case, setup\_mocked\_paths):  
 # Simulate an error during HTML export by raising an exception  
 with patch('builtins.open', side\_effect=Exception("Failed to write HTML")):  
 try:  
 result = base\_test\_case.export\_utils.export\_to\_html("test\_command", "http://example.com", "Success")  
 except Exception as e:  
 # Assert that the correct exception was raised and log the result  
 assert str(e) == "Failed to write HTML"  
 logging.info(f"Expected exception caught: {str(e)}")  
 logging.info("Test negative HTML export passed with expected exception.")  
  
 def test\_negative\_excel\_export(self, base\_test\_case, setup\_mocked\_paths):  
 # Simulate an error during Excel export by raising an exception  
 with patch('pandas.DataFrame.to\_excel', side\_effect=Exception("Failed to write Excel")):  
 try:  
 result = base\_test\_case.export\_utils.log\_to\_excel("test\_command", "http://example.com", "Success")  
 except Exception as e:  
 # Assert that the correct exception was raised and log the result  
 assert str(e) == "Failed to write Excel"  
 logging.info(f"Expected exception caught: {str(e)}")  
 logging.info("Test negative Excel export passed with expected exception.")  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 logging.info("Starting pytest for TestExportUtils...")  
 pytest.main([\_\_file\_\_])

--- unitTest\_fetch\_account\_by\_website.py ---

import pytest, os, sys  
from unittest.mock import MagicMock  
from test\_init import setup\_logging, base\_test\_case, save\_test\_results\_to\_file, log\_test\_start\_end, logging  
  
  
setup\_logging() # Initialize logging if needed  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestAccountDAOFetchByWebsite:  
 @pytest.fixture  
 def account\_dao(self, base\_test\_case, mocker):  
 # Mock the psycopg2 connection and cursor  
 mocker.patch('psycopg2.connect')  
 account\_dao = base\_test\_case.account\_dao  
 account\_dao.connection = MagicMock()  
 account\_dao.cursor = MagicMock()  
 logging.info("Fake database connection established")  
 return account\_dao  
  
 def test\_entity\_fetch\_account\_success(self, account\_dao):  
 # Setup the cursor's behavior for successful fetch  
 account\_dao.cursor.execute = MagicMock()  
 account\_dao.cursor.fetchone.return\_value = ("test\_user", "password123")  
   
 # Test the fetch\_account\_by\_website method for success  
 result = account\_dao.fetch\_account\_by\_website("example.com")  
   
 # Log the result of the operation  
 logging.info(f"AccountDAO.fetch\_account\_by\_website returned {result}")  
 logging.info("Expected result: ('test\_user', 'password123')")  
   
 # Assert and log the final outcome  
 assert result == ("test\_user", "password123"), "Account should be fetched successfully"  
 logging.info("Test fetch\_account\_success passed")  
  
 def test\_entity\_fetch\_account\_fail(self, account\_dao):  
 # Setup the cursor's behavior to simulate failure  
 account\_dao.cursor.execute = MagicMock()  
 account\_dao.cursor.fetchone.return\_value = None  
  
 # Perform the test  
 result = account\_dao.fetch\_account\_by\_website("fail.com")  
   
 # Log the result of the operation  
 logging.info(f"AccountDAO.fetch\_account\_by\_website returned {result}")  
 logging.info("Expected result: None")  
   
 # Assert and log the final outcome  
 assert result is None, "No account should be fetched"  
 logging.info("Test fetch\_account\_fail passed")  
  
  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestAccountControlFetchByWebsite:  
 @pytest.fixture  
 def account\_control(self, base\_test\_case, mocker):  
 # Get the mocked AccountControl from base\_test\_case  
 account\_control = base\_test\_case.account\_control  
 account\_control.account\_dao = MagicMock(spec=base\_test\_case.account\_dao)  
   
 # Mock methods used in the control layer's fetch\_account\_by\_website  
 mocker.patch.object(account\_control.account\_dao, 'connect')  
 mocker.patch.object(account\_control.account\_dao, 'close')  
 logging.info("Mocked AccountDAO connection and close methods")  
 return account\_control  
  
 def test\_control\_fetch\_account\_success(self, account\_control):  
 # Mock successful fetch in the DAO layer  
 account\_control.account\_dao.fetch\_account\_by\_website.return\_value = ("test\_user", "password123")  
   
 # Call the control method and check the response  
 result = account\_control.fetch\_account\_by\_website("example.com")  
 expected\_message = ("test\_user", "password123")  
   
 # Log the response and expectations  
 logging.info(f"Control method fetch\_account\_by\_website returned: '{result}'")  
 logging.info("Expected message: ('test\_user', 'password123')")  
   
 # Assert the success message  
 assert result == expected\_message, "The fetch result should match expected output"  
 logging.info("Test control\_fetch\_account\_success passed")  
  
 def test\_control\_fetch\_account\_fail(self, account\_control):  
 # Mock failure in the DAO layer  
 account\_control.account\_dao.fetch\_account\_by\_website.return\_value = None  
   
 # Call the control method and check the response  
 result = account\_control.fetch\_account\_by\_website("fail.com")  
 expected\_message = "No account found for fail.com."  
   
 # Log the response and expectations  
 logging.info(f"Control method fetch\_account\_by\_website returned: '{result}'")  
 logging.info("Expected message: 'No account found for fail.com.'")  
   
 # Assert the failure message  
 assert result == expected\_message, "The failure message should match expected output"  
 logging.info("Test control\_fetch\_account\_fail passed")  
  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_]) # Run pytest directly

--- unitTest\_fetch\_all\_accounts.py ---

import pytest, os, sys  
from unittest.mock import MagicMock  
from test\_init import setup\_logging, base\_test\_case, save\_test\_results\_to\_file, log\_test\_start\_end, logging  
  
  
setup\_logging() # Initialize logging if needed  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestAccountDAO:  
 @pytest.fixture  
 def account\_dao(self, base\_test\_case, mocker):  
 mocker.patch('psycopg2.connect')  
 account\_dao = base\_test\_case.account\_dao  
 account\_dao.connection = MagicMock()  
 account\_dao.cursor = MagicMock()  
 logging.info("Fake database connection established")  
 return account\_dao  
  
 def test\_entity\_fetch\_all\_accounts\_success(self, account\_dao):  
 # Mock successful fetch operation  
 mock\_accounts = [(1, "test\_user", "password123", "example.com"), (2, "test\_user2", "password456", "example2.com")]  
 account\_dao.cursor.fetchall.return\_value = mock\_accounts  
   
 # Test fetch\_all\_accounts method  
 result = account\_dao.fetch\_all\_accounts()  
   
 logging.info(f"AccountDAO.fetch\_all\_accounts returned {result}")  
 logging.info("Expected result: a list of accounts")  
   
 # Assert and log the final outcome  
 assert result == mock\_accounts, "Should return a list of accounts"  
 logging.info("Test fetch\_all\_accounts\_success passed")  
  
 def test\_entity\_fetch\_all\_accounts\_fail(self, account\_dao):  
 # Mock failed fetch operation  
 account\_dao.cursor.fetchall.side\_effect = Exception("Database error")  
   
 # Test fetch\_all\_accounts method  
 result = account\_dao.fetch\_all\_accounts()  
   
 logging.info(f"AccountDAO.fetch\_all\_accounts returned {result}")  
 logging.info("Expected result: an empty list due to failure")  
   
 # Assert and log the final outcome  
 assert result == [], "Should return an empty list due to failure"  
 logging.info("Test fetch\_all\_accounts\_fail passed")  
  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestAccountControl:  
 @pytest.fixture  
 def account\_control(self, base\_test\_case, mocker):  
 account\_control = base\_test\_case.account\_control  
 account\_control.account\_dao = MagicMock(spec=base\_test\_case.account\_dao)  
   
 # Mock methods used in the control layer's fetch\_all\_accounts  
 mocker.patch.object(account\_control.account\_dao, 'connect')  
 mocker.patch.object(account\_control.account\_dao, 'close')  
 logging.info("Mocked AccountDAO connection and close methods")  
 return account\_control  
  
 def test\_control\_fetch\_all\_accounts\_success(self, account\_control):  
 # Mock successful fetch in the DAO layer  
 mock\_accounts = [(1, "test\_user", "password123", "example.com"), (2, "test\_user2", "password456", "example2.com")]  
 account\_control.account\_dao.fetch\_all\_accounts.return\_value = mock\_accounts  
   
 # Call the control method and check the response  
 result = account\_control.fetch\_all\_accounts()  
   
 expected\_message = "Accounts:\nID: 1, Username: test\_user, Password: password123, Website: example.com\nID: 2, Username: test\_user2, Password: password456, Website: example2.com"  
   
 logging.info(f"Control method fetch\_all\_accounts returned: '{result}'")  
 logging.info(f"Expected message: '{expected\_message}'")  
   
 # Assert and log the final outcome  
 assert result == expected\_message, "The fetched accounts list should match expected output"  
 logging.info("Test control\_fetch\_all\_accounts\_success passed")  
  
 def test\_control\_fetch\_all\_accounts\_fail(self, account\_control):  
 # Mock failed fetch in the DAO layer  
 account\_control.account\_dao.fetch\_all\_accounts.return\_value = []  
   
 # Call the control method and check the response  
 result = account\_control.fetch\_all\_accounts()  
   
 expected\_message = "No accounts found."  
   
 logging.info(f"Control method fetch\_all\_accounts returned: '{result}'")  
 logging.info(f"Expected message: '{expected\_message}'")  
   
 # Assert and log the final outcome  
 assert result == expected\_message, "The message should indicate no accounts found"  
 logging.info("Test control\_fetch\_all\_accounts\_fail passed")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_]) # Run pytest directly

--- unitTest\_get\_price.py ---

import pytest, logging  
from unittest.mock import patch  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
async def test\_get\_price\_success(base\_test\_case):  
 # Simulate a successful price retrieval  
 with patch('entity.PriceEntity.PriceEntity.get\_price\_from\_page') as mock\_get\_price:  
 url = "https://example.com/product"  
 mock\_get\_price.return\_value = "$199.99"  
 expected\_entity\_result = "$199.99"  
 expected\_control\_result = "$199.99"  
  
 # Execute the command  
 result = await base\_test\_case.price\_control.receive\_command("get\_price", url)  
  
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Expected: {expected\_entity\_result}")  
 logging.info(f"Entity Layer Received: {mock\_get\_price.return\_value}")  
 assert mock\_get\_price.return\_value == expected\_entity\_result, "Entity layer assertion failed."  
 logging.info("Unit Test Passed for entity layer.\n")  
  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
async def test\_get\_price\_invalid\_url(base\_test\_case):  
 # Simulate an invalid URL case  
 with patch('entity.PriceEntity.PriceEntity.get\_price\_from\_page') as mock\_get\_price:  
 invalid\_url = "invalid\_url"  
 mock\_get\_price.return\_value = "Error fetching price: Invalid URL"  
 expected\_control\_result = "Error fetching price: Invalid URL"  
  
 # Execute the command  
 result = await base\_test\_case.price\_control.receive\_command("get\_price", invalid\_url)  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer invalid URL handling.\n")  
  
async def test\_get\_price\_failure\_entity(base\_test\_case):  
 # Simulate an entity layer failure when fetching the price  
 with patch('entity.PriceEntity.PriceEntity.get\_price\_from\_page', side\_effect=Exception("Failed to fetch price")) as mock\_get\_price:  
 url = "https://example.com/product"  
 expected\_control\_result = "Failed to fetch price: Failed to fetch price"  
  
 # Execute the command  
 result = await base\_test\_case.price\_control.receive\_command("get\_price", url)  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer failed to handle entity error correctly."  
 logging.info("Unit Test Passed for entity layer error handling.")  
  
async def test\_get\_price\_failure\_control(base\_test\_case):  
 # Simulate a control layer failure  
 with patch('control.PriceControl.PriceControl.receive\_command', side\_effect=Exception("Control Layer Failed")) as mock\_control:  
 url = "https://example.com/product"  
 expected\_control\_result = "Control Layer Exception: Control Layer Failed"  
  
 # Execute the command and catch the raised exception  
 try:  
 result = await base\_test\_case.price\_control.receive\_command("get\_price", url)  
 except Exception as e:  
 result = f"Control Layer Exception: {str(e)}"  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer failure.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_launch\_browser.py ---

import pytest, logging  
from unittest.mock import patch  
from test\_init import base\_test\_case, log\_test\_start\_end, setup\_logging  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
async def test\_launch\_browser\_success(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.launch\_browser') as mock\_launch:  
 # Setup mock return and expected outcomes  
 mock\_launch.return\_value = "Browser launched."  
 expected\_entity\_result = "Browser launched."  
 expected\_control\_result = "Control Object Result: Browser launched."  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("launch\_browser")  
   
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Expected: {expected\_entity\_result}")  
 logging.info(f"Entity Layer Received: {mock\_launch.return\_value}")  
 assert mock\_launch.return\_value == expected\_entity\_result, "Entity layer assertion failed."  
 logging.info("Unit Test Passed for entity layer.\n")  
   
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
async def test\_launch\_browser\_already\_running(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.launch\_browser', return\_value="Browser is already running.") as mock\_launch:  
 expected\_entity\_result = "Browser is already running."  
 expected\_control\_result = "Control Object Result: Browser is already running."  
   
 result = await base\_test\_case.browser\_control.receive\_command("launch\_browser")  
   
 logging.info(f"Entity Layer Expected: {expected\_entity\_result}")  
 logging.info(f"Entity Layer Received: {mock\_launch.return\_value}")  
 assert mock\_launch.return\_value == expected\_entity\_result, "Entity layer assertion failed."  
 logging.info("Unit Test Passed for entity layer.\n")  
   
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
async def test\_launch\_browser\_failure\_control(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.launch\_browser', side\_effect=Exception("Internal error")) as mock\_launch:  
 expected\_result = "Control Layer Exception: Internal error"  
   
 result = await base\_test\_case.browser\_control.receive\_command("launch\_browser")  
   
 logging.info(f"Control Layer Expected to Report: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer failed to handle or report the entity error correctly."  
 logging.info("Unit Test Passed for control layer error handling.")  
  
  
async def test\_launch\_browser\_failure\_entity(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.launch\_browser', side\_effect=Exception("Failed to launch browser: Internal error")) as mock\_launch:  
 expected\_control\_result = "Control Layer Exception: Failed to launch browser: Internal error"  
   
 result = await base\_test\_case.browser\_control.receive\_command("launch\_browser")  
   
 logging.info(f"Entity Layer Expected Failure: Failed to launch browser: Internal error")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer failed to report entity error correctly."  
 logging.info("Unit Test Passed for entity layer error handling.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_login.py ---

import pytest  
import logging  
from unittest.mock import patch, MagicMock  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
  
setup\_logging()  
  
async def test\_login\_success(base\_test\_case):  
 """Test that the login is successful when valid credentials are provided."""  
 # Patch methods  
 with patch('entity.BrowserEntity.BrowserEntity.login') as mock\_login:  
 with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:  
 # Setup mock return values  
 mock\_login.return\_value = "Logged in to http://example.com successfully with username: sample\_username"  
 mock\_fetch\_account.return\_value = ("sample\_username", "sample\_password")  
   
 expected\_entity\_result = "Logged in to http://example.com successfully with username: sample\_username"  
 expected\_control\_result = f"Control Object Result: {expected\_entity\_result}"  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")  
   
 # Assert results and logging  
 logging.info(f"Entity Layer Expected: {expected\_entity\_result}")  
 logging.info(f"Entity Layer Received: {mock\_login.return\_value}")  
 assert mock\_login.return\_value == expected\_entity\_result, "Entity layer assertion failed."  
 logging.info("Unit Test Passed for entity layer.\n")  
   
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
async def test\_login\_no\_account(base\_test\_case):  
 """Test that the control layer handles the scenario where no account is found for the website."""  
 with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:  
 # Setup mock to return no account  
 mock\_fetch\_account.return\_value = None  
   
 expected\_result = "No account found for example.com"  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")  
   
 # Assert results and logging  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer failed to handle missing account correctly."  
 logging.info("Unit Test Passed for missing account handling.")  
  
async def test\_login\_entity\_layer\_failure(base\_test\_case):  
 """Test that the control layer handles an exception raised in the entity layer."""  
 with patch('entity.BrowserEntity.BrowserEntity.login') as mock\_login:  
 with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:  
 # Setup mocks  
 mock\_login.side\_effect = Exception("BrowserEntity\_Failed to log in to http://example.com: Internal error")  
 mock\_fetch\_account.return\_value = ("sample\_username", "sample\_password")  
   
 expected\_result = "Control Layer Exception: BrowserEntity\_Failed to log in to http://example.com: Internal error"  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")  
   
 # Assert results and logging  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer failed to handle entity layer exception."  
 logging.info("Unit Test Passed for entity layer failure.")  
  
  
async def test\_login\_control\_layer\_failure(base\_test\_case):  
 """Test that the control layer handles an unexpected failure or exception."""  
 with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:  
 # Simulate an exception being raised in the control layer  
 mock\_fetch\_account.side\_effect = Exception("Control layer failure during account fetch.")  
   
 expected\_result = "Control Layer Exception: Control layer failure during account fetch."  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")  
   
 # Assert results and logging  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer failed to handle control layer exception."  
 logging.info("Unit Test Passed for control layer failure handling.")  
  
async def test\_login\_invalid\_url(base\_test\_case):  
 """Test that the control layer handles the scenario where the URL or selectors are not found."""  
 with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:  
 with patch('utils.css\_selectors.Selectors.get\_selectors\_for\_url') as mock\_get\_selectors:  
 # Setup mocks  
 mock\_fetch\_account.return\_value = ("sample\_username", "sample\_password")  
 mock\_get\_selectors.return\_value = {'url': None} # Simulate missing URL  
   
 expected\_result = "URL for example not found."  
   
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("login", site="example")  
   
 # Assert results and logging  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer failed to handle missing URL or selectors."  
 logging.info("Unit Test Passed for missing URL/selector handling.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_navigate\_to\_website.py ---

import pytest, logging  
from unittest.mock import patch  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
  
async def test\_navigate\_to\_website\_success(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.navigate\_to\_website') as mock\_navigate:  
 # Setup mock return and expected outcomes  
 url = "https://example.com"  
 mock\_navigate.return\_value = f"Navigated to {url}"  
 expected\_entity\_result = f"Navigated to {url}"  
 expected\_control\_result = f"Control Object Result: Navigated to {url}"  
  
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=url)  
  
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Expected: {expected\_entity\_result}")  
 logging.info(f"Entity Layer Received: {mock\_navigate.return\_value}")  
 assert mock\_navigate.return\_value == expected\_entity\_result, "Entity layer assertion failed."  
 logging.info("Unit Test Passed for entity layer.\n")  
  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
  
async def test\_navigate\_to\_website\_invalid\_url(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.navigate\_to\_website') as mock\_navigate:  
 # Setup mock return and expected outcomes  
 invalid\_site = "invalid\_site"  
 mock\_navigate.return\_value = f"URL for {invalid\_site} not found."  
 expected\_control\_result = f"URL for {invalid\_site} not found."  
  
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=invalid\_site)  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer invalid URL handling.\n")  
  
  
async def test\_navigate\_to\_website\_failure\_entity(base\_test\_case):  
 with patch('entity.BrowserEntity.BrowserEntity.navigate\_to\_website', side\_effect=Exception("Failed to navigate")) as mock\_navigate:  
 # Setup expected outcomes  
 url = "https://example.com"  
 expected\_control\_result = "Control Layer Exception: Failed to navigate"  
  
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=url)  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer failed to handle entity error correctly."  
 logging.info("Unit Test Passed for entity layer error handling.")  
  
  
async def test\_navigate\_to\_website\_launch\_browser\_on\_failure(base\_test\_case):  
 # This test simulates a scenario where the browser is not open and needs to be launched first.  
 with patch('entity.BrowserEntity.BrowserEntity.is\_browser\_open', return\_value=False), \  
 patch('entity.BrowserEntity.BrowserEntity.launch\_browser', return\_value="Browser launched."), \  
 patch('entity.BrowserEntity.BrowserEntity.navigate\_to\_website') as mock\_navigate:  
   
 # Setup expected outcomes  
 url = "https://example.com"  
 mock\_navigate.return\_value = f"Navigated to {url}"  
 expected\_control\_result = f"Control Object Result: Navigated to {url}"  
  
 # Execute the command  
 result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=url)  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer with browser launch.\n")  
  
  
async def test\_navigate\_to\_website\_failure\_control(base\_test\_case):  
 # This simulates a failure within the control layer  
 with patch('control.BrowserControl.BrowserControl.receive\_command', side\_effect=Exception("Control Layer Failed")) as mock\_control:  
   
 # Setup expected outcomes  
 url = "https://example.com"  
 expected\_control\_result = "Control Layer Exception: Control Layer Failed"  
  
 # Execute the command and catch the raised exception  
 try:  
 result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=url)  
 except Exception as e:  
 result = f"Control Layer Exception: {str(e)}"  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer failure.")  
   
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_project\_help.py ---

import pytest, logging  
from unittest.mock import patch  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
async def test\_project\_help\_success(base\_test\_case):  
 with patch('control.BotControl.BotControl.receive\_command') as mock\_help:  
 # Setup mock return and expected outcomes  
 mock\_help.return\_value = (  
 "Here are the available commands:\n"  
 "!project\_help - Get help on available commands.\n"  
 "!fetch\_all\_accounts - Fetch all stored accounts.\n"  
 "!add\_account 'username' 'password' 'website' - Add a new account to the database.\n"  
 "!fetch\_account\_by\_website 'website' - Fetch account details by website.\n"  
 "!delete\_account 'account\_id' - Delete an account by its ID.\n"  
 "!launch\_browser - Launch the browser.\n"  
 "!close\_browser - Close the browser.\n"  
 "!navigate\_to\_website 'url' - Navigate to a specified website.\n"  
 "!login 'website' - Log in to a website (e.g., !login bestbuy).\n"  
 "!get\_price 'url' - Check the price of a product on a specified website.\n"  
 "!start\_monitoring\_price 'url' 'frequency' - Start monitoring a product's price at a specific interval (frequency in minutes).\n"  
 "!stop\_monitoring\_price - Stop monitoring the product's price.\n"  
 "!check\_availability 'url' - Check availability for a restaurant or service.\n"  
 "!start\_monitoring\_availability 'url' 'frequency' - Monitor availability at a specific interval.\n"  
 "!stop\_monitoring\_availability - Stop monitoring availability.\n"  
 "!stop\_bot - Stop the bot.\n"  
 )  
 expected\_result = mock\_help.return\_value  
   
 # Execute the command  
 result = await base\_test\_case.bot\_control.receive\_command("project\_help")  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for project help.\n")  
  
  
async def test\_project\_help\_failure(base\_test\_case):  
 with patch('control.BotControl.BotControl.receive\_command', side\_effect=Exception("Error handling help command")) as mock\_help:  
 expected\_result = "Error handling help command: Error handling help command"  
   
 # Execute the command and catch the raised exception  
 try:  
 result = await base\_test\_case.bot\_control.receive\_command("project\_help")  
 except Exception as e:  
 result = f"Error handling help command: {str(e)}"  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer failed to handle error correctly."  
 logging.info("Unit Test Passed for error handling in project help.\n")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_receive\_email.py ---

import pytest  
from unittest.mock import MagicMock  
from test\_init import setup\_logging, base\_test\_case, save\_test\_results\_to\_file, log\_test\_start\_end, logging  
  
setup\_logging()  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestEmailDAO:  
   
 @pytest.fixture  
 def email\_dao(self, base\_test\_case, mocker):  
 # Use the send\_email\_with\_attachments from base\_test\_case  
 email\_dao = base\_test\_case.email\_dao  
 mocker.patch('smtplib.SMTP')  
 logging.info("Mocked EmailDAO with send\_email\_with\_attachments method")  
 return email\_dao  
  
 def test\_entity\_send\_email\_success(self, email\_dao):  
 # Mock successful email sending  
 email\_dao.return\_value = "Email with file 'monitor\_price.html' sent successfully!"  
   
 # Perform the test  
 result = email\_dao('monitor\_price.html')  
   
 # Log and assert the result  
 assert result == "Email with file 'monitor\_price.html' sent successfully!"  
 logging.info("Test send\_email\_success passed")  
  
 def test\_entity\_send\_email\_fail(self, email\_dao):  
 # Mock failure in email sending  
 email\_dao.return\_value = "File 'non\_existent\_file.html' not found."  
   
 # Perform the test  
 result = email\_dao('non\_existent\_file.html')  
   
 # Log and assert the result  
 assert result == "File 'non\_existent\_file.html' not found in either excelFiles or htmlFiles."  
 logging.info("Test send\_email\_fail passed")  
  
  
@pytest.mark.usefixtures("base\_test\_case")  
class TestEmailControl:  
  
 @pytest.fixture  
 def email\_control(self, base\_test\_case, mocker):  
 # Get the bot control from base\_test\_case, which should handle the receive\_command method  
 email\_control = base\_test\_case.bot\_control  
 email\_control.receive\_command = MagicMock() # Mock the receive\_command method  
 logging.info("Mocked EmailControl (BotControl) for control layer")  
 return email\_control  
  
 def test\_control\_send\_email\_success(self, email\_control):  
 # Mock successful email sending  
 email\_control.receive\_command.return\_value = "Email with file 'monitor\_price.html' sent successfully!"  
   
 # Call the control method and check the response  
 result = email\_control.receive\_command("receive\_email", "monitor\_price.html")  
   
 # Log and assert the result  
 assert result == "Email with file 'monitor\_price.html' sent successfully!"  
 logging.info("Test control\_send\_email\_success passed")  
  
 def test\_control\_send\_email\_fail(self, email\_control):  
 # Mock failure in email sending  
 email\_control.receive\_command.return\_value = "File 'non\_existent\_file.html' not found."  
   
 # Call the control method and check the response  
 result = email\_control.receive\_command("receive\_email", "non\_existent\_file.html")  
   
 # Log and assert the result  
 assert result == "File 'non\_existent\_file.html' not found."  
 logging.info("Test control\_send\_email\_fail passed")  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_]) # Run pytest directly

--- unitTest\_start\_monitoring\_availability.py ---

import pytest, logging  
from unittest.mock import patch  
from test\_init import base\_test\_case, setup\_logging, run\_monitoring\_loop, log\_test\_start\_end  
import asyncio  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
async def test\_start\_monitoring\_availability\_success(base\_test\_case):  
 with patch('entity.AvailabilityEntity.AvailabilityEntity.check\_availability') as mock\_check:  
 url = "https://example.com"  
 mock\_check.return\_value = "Selected or default date is available for booking."  
   
 expected\_control\_result = [  
 "Checked availability: Selected or default date is available for booking.",  
 "Monitoring stopped successfully!"  
 ]  
  
 # Run the monitoring loop once  
 actual\_control\_result = await run\_monitoring\_loop(  
 base\_test\_case.availability\_control,  
 base\_test\_case.availability\_control.check\_availability,  
 url,  
 "2024-10-01",  
 1  
 )  
  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {actual\_control\_result}")  
 assert actual\_control\_result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer.")  
  
  
async def test\_start\_monitoring\_availability\_failure\_entity(base\_test\_case):  
 with patch('entity.AvailabilityEntity.AvailabilityEntity.check\_availability', side\_effect=Exception("Failed to check availability")):  
 url = "https://example.com"  
 expected\_control\_result = [  
 "Failed to check availability: Failed to check availability",  
 "Monitoring stopped successfully!"  
 ]  
  
 # Run the monitoring loop once  
 actual\_control\_result = await run\_monitoring\_loop(  
 base\_test\_case.availability\_control,  
 base\_test\_case.availability\_control.check\_availability,  
 url,  
 "2024-10-01",  
 1  
 )  
  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {actual\_control\_result}")  
 assert actual\_control\_result == expected\_control\_result, "Control layer failed to handle entity error correctly."  
 logging.info("Unit Test Passed for entity layer error handling.")  
  
  
async def test\_start\_monitoring\_availability\_failure\_control(base\_test\_case):  
 with patch('control.AvailabilityControl.AvailabilityControl.receive\_command', side\_effect=Exception("Control Layer Failed")):  
 url = "https://example.com"  
 expected\_control\_result = "Control Layer Exception: Control Layer Failed"  
  
 try:  
 result = await base\_test\_case.availability\_control.receive\_command("start\_monitoring\_availability", url, "2024-10-01", 5)  
 except Exception as e:  
 result = f"Control Layer Exception: {str(e)}"  
  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer failure.")  
  
  
async def test\_start\_monitoring\_availability\_already\_running(base\_test\_case):  
 with patch('entity.AvailabilityEntity.AvailabilityEntity.check\_availability') as mock\_check:  
 url = "https://example.com"  
 base\_test\_case.availability\_control.is\_monitoring = True  
 expected\_control\_result = "Already monitoring availability."  
  
 result = await base\_test\_case.availability\_control.receive\_command("start\_monitoring\_availability", url, "2024-10-01", 5)  
  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer failed to handle already running condition."  
 logging.info("Unit Test Passed for control layer already running handling.\n")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_start\_monitoring\_price.py ---

import pytest  
import logging  
from unittest.mock import patch, AsyncMock  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
  
async def test\_start\_monitoring\_price\_success(base\_test\_case):  
 with patch('entity.PriceEntity.PriceEntity.get\_price\_from\_page', return\_value="100 USD") as mock\_get\_price:  
   
 # Setup expected outcomes  
 url = "https://example.com/product"  
 expected\_result = "Starting price monitoring. Current price: 100 USD"  
   
 # Mocking the sleep method to break out of the loop after the first iteration  
 with patch('asyncio.sleep', side\_effect=KeyboardInterrupt):  
 try:  
 # Execute the command  
 base\_test\_case.price\_control.is\_monitoring = False  
 result = await base\_test\_case.price\_control.receive\_command("start\_monitoring\_price", url, 1)  
 except KeyboardInterrupt:  
 # Force the loop to stop after the first iteration  
 base\_test\_case.price\_control.is\_monitoring = False  
   
 # Log and assert the outcomes  
 logging.info(f"Entity Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {base\_test\_case.price\_control.results[0]}")  
 assert expected\_result in base\_test\_case.price\_control.results[0], "Price monitoring did not start as expected."  
 logging.info("Unit Test Passed for start\_monitoring\_price success scenario.\n")  
  
  
async def test\_start\_monitoring\_price\_already\_running(base\_test\_case):  
 # Test when price monitoring is already running  
 base\_test\_case.price\_control.is\_monitoring = True  
 expected\_result = "Already monitoring prices."  
   
 # Execute the command  
 result = await base\_test\_case.price\_control.receive\_command("start\_monitoring\_price", "https://example.com/product", 1)  
   
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer did not detect that monitoring was already running."  
 logging.info("Unit Test Passed for already running scenario.\n")  
  
  
async def test\_start\_monitoring\_price\_failure\_in\_entity(base\_test\_case):  
 # Mock entity failure during price fetching  
 with patch('entity.PriceEntity.PriceEntity.get\_price\_from\_page', side\_effect=Exception("Error fetching price")) as mock\_get\_price:  
   
 # Setup expected outcomes  
 url = "https://example.com/product"  
 expected\_result = "Starting price monitoring. Current price: Failed to fetch price: Error fetching price"  
   
 # Mocking the sleep method to break out of the loop after the first iteration  
 with patch('asyncio.sleep', side\_effect=KeyboardInterrupt):  
 try:  
 # Execute the command  
 base\_test\_case.price\_control.is\_monitoring = False  
 await base\_test\_case.price\_control.receive\_command("start\_monitoring\_price", url, 1)  
 except KeyboardInterrupt:  
 # Force the loop to stop after the first iteration  
 base\_test\_case.price\_control.is\_monitoring = False  
   
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {base\_test\_case.price\_control.results[-1]}")  
 assert expected\_result in base\_test\_case.price\_control.results[-1], "Entity layer did not handle failure correctly."  
 logging.info("Unit Test Passed for entity layer failure scenario.\n")  
  
  
async def test\_start\_monitoring\_price\_failure\_in\_control(base\_test\_case):  
 # Mock control layer failure  
 with patch('control.PriceControl.PriceControl.start\_monitoring\_price', side\_effect=Exception("Control Layer Exception")) as mock\_start\_monitoring:  
   
 # Setup expected outcomes  
 expected\_result = "Control Layer Exception"  
   
 # Execute the command and catch the raised exception  
 try:  
 result = await base\_test\_case.price\_control.receive\_command("start\_monitoring\_price", "https://example.com/product", 1)  
 except Exception as e:  
 result = f"Control Layer Exception: {str(e)}"  
   
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert expected\_result in result, "Control layer did not handle the failure correctly."  
 logging.info("Unit Test Passed for control layer failure scenario.\n")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_stop\_bot.py ---

import pytest  
import logging  
from unittest.mock import MagicMock, patch  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
  
async def test\_stop\_bot\_success(base\_test\_case):  
 with patch('control.BotControl.BotControl.receive\_command') as mock\_stop\_bot:  
 # Setup mock return and expected outcomes  
 mock\_stop\_bot.return\_value = "Bot has been shut down."  
 expected\_entity\_result = "Bot has been shut down."  
 expected\_control\_result = "Bot has been shut down."  
  
 # Execute the command  
 result = await base\_test\_case.bot\_control.receive\_command("stop\_bot", ctx=MagicMock())  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer stop bot.\n")  
  
  
  
async def test\_stop\_bot\_failure\_control(base\_test\_case):  
 with patch('control.BotControl.BotControl.receive\_command', side\_effect=Exception("Control Layer Failed")) as mock\_control:  
 # Setup expected outcomes  
 expected\_control\_result = "Control Layer Exception: Control Layer Failed"  
  
 # Execute the command and catch the raised exception  
 try:  
 result = await base\_test\_case.bot\_control.receive\_command("stop\_bot", ctx=MagicMock())  
 except Exception as e:  
 result = f"Control Layer Exception: {str(e)}"  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed."  
 logging.info("Unit Test Passed for control layer failure.\n")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_stop\_monitoring\_availability.py ---

import pytest, logging  
from unittest.mock import patch  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
import asyncio  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
async def test\_stop\_monitoring\_availability\_success(base\_test\_case):  
 # Simulate the case where monitoring is already running  
 base\_test\_case.availability\_control.is\_monitoring = True  
 base\_test\_case.availability\_control.results = ["Checked availability: Selected or default date is available for booking."]  
   
 # Expected message to be present in the result  
 expected\_control\_result\_contains = "Monitoring stopped successfully!"  
   
 # Execute the stop command  
 result = base\_test\_case.availability\_control.stop\_monitoring\_availability()  
   
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected to contain: {expected\_control\_result\_contains}")  
 logging.info(f"Control Layer Received: {result}")  
   
 assert expected\_control\_result\_contains in result, "Control layer assertion failed for stop monitoring."  
 logging.info("Unit Test Passed for stop monitoring availability.")  
  
async def test\_stop\_monitoring\_availability\_no\_active\_session(base\_test\_case):  
 # Simulate the case where no monitoring session is active  
 base\_test\_case.availability\_control.is\_monitoring = False  
 expected\_control\_result = "There was no active availability monitoring session. Nothing to stop."  
   
 # Execute the stop command  
 result = base\_test\_case.availability\_control.stop\_monitoring\_availability()  
   
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_control\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_control\_result, "Control layer assertion failed for no active session."  
 logging.info("Unit Test Passed for stop monitoring with no active session.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_stop\_monitoring\_price.py ---

import pytest  
import logging  
from unittest.mock import patch, AsyncMock  
from test\_init import base\_test\_case, setup\_logging, log\_test\_start\_end  
  
# Enable asyncio for all tests in this file  
pytestmark = pytest.mark.asyncio  
setup\_logging()  
  
async def test\_stop\_monitoring\_price\_success(base\_test\_case):  
 # Set up monitoring to be active  
 base\_test\_case.price\_control.is\_monitoring = True  
 base\_test\_case.price\_control.results = ["Price went up!", "Price went down!"]  
  
 # Expected result after stopping monitoring  
 expected\_result = "Results for price monitoring:\nPrice went up!\nPrice went down!\n\nPrice monitoring stopped successfully!"  
   
 # Execute the command  
 result = base\_test\_case.price\_control.stop\_monitoring\_price()  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer did not return the correct results for stopping monitoring."  
 logging.info("Unit Test Passed for stop\_monitoring\_price success scenario.\n")  
  
  
async def test\_stop\_monitoring\_price\_not\_active(base\_test\_case):  
 # Test the case where monitoring is not active  
 base\_test\_case.price\_control.is\_monitoring = False  
 expected\_result = "There was no active price monitoring session. Nothing to stop."  
  
 # Execute the command  
 result = base\_test\_case.price\_control.stop\_monitoring\_price()  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert result == expected\_result, "Control layer did not detect that monitoring was not active."  
 logging.info("Unit Test Passed for stop\_monitoring\_price when not active.\n")  
  
  
async def test\_stop\_monitoring\_price\_failure\_in\_control(base\_test\_case):  
 # Simulate failure in control layer during stopping of monitoring  
 with patch('control.PriceControl.PriceControl.stop\_monitoring\_price', side\_effect=Exception("Error stopping price monitoring")) as mock\_stop\_monitoring:  
  
 # Expected result when the control layer fails  
 expected\_result = "Error stopping price monitoring"  
   
 # Execute the command and handle exception  
 try:  
 result = base\_test\_case.price\_control.stop\_monitoring\_price()  
 except Exception as e:  
 result = str(e)  
  
 # Log and assert the outcomes  
 logging.info(f"Control Layer Expected: {expected\_result}")  
 logging.info(f"Control Layer Received: {result}")  
 assert expected\_result in result, "Control layer did not handle the failure correctly."  
 logging.info("Unit Test Passed for stop\_monitoring\_price failure scenario.\n")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])