```
--- temporary.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__])



--- 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__))))

#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----------------------------------------------------")

# Import your control classes
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

@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()
    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, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end, save_test_results_to_file

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_add_account_success(base_test_case):
        with patch('control.AccountControl.AccountControl.add_account', return_value="Account for example.com added successfully.") as mock_add_account:
        # Setup expected outcomes
        username = "test_user"
        password = "test_pass"
        website = "example.com"
        expected_entity_result = "Account for example.com added successfully."
        expected_control_result = "Account for example.com added successfully."

        # Execute the command
        result = base_test_case.account_control.add_account(username, password, website)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
```

```
    logging.info(f"Entity Layer Received: {mock_add_account.return_value}")
        assert mock_add_account.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_add_account_failure_invalid_data(base_test_case):
    with patch('control.AccountControl.AccountControl.add_account', return_value="Failed to add account for example.com.") as mock_add_account:
    # Setup expected outcomes for invalid data scenario
    username = ""  # Invalid username
    password = ""  # Invalid password
    website = "example.com"
    expected_control_result = "Failed to add account for example.com."

    # Execute the command
    result = base_test_case.account_control.add_account(username, password, website)

    # 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 data handling.\n")
```

```
async def test_add_account_failure_entity_error(base_test_case):
                                    with          patch('control.AccountControl.AccountControl.add_account',
side_effect=Exception("Database Error")) as mock_add_account:
        # Setup expected outcomes
        username = "test_user"
        password = "test_pass"
        website = "example.com"
        expected_control_result = "Control Layer Exception: Database Error"

        # Execute the command
        try:
            result = base_test_case.account_control.add_account(username, password, website)
        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 failed to handle entity error correctly."
        logging.info("Unit Test Passed for control layer error handling.")

async def test_add_account_already_exists(base_test_case):
    # This simulates a scenario where an account for the website already exists
        with  patch('control.AccountControl.AccountControl.add_account',  return_value="Failed  to  add
account for example.com. Account already exists.") as mock_add_account:
        # Setup expected outcomes
```

```
        username = "test_user"
        password = "test_pass"
        website = "example.com"
        expected_control_result = "Failed to add account for example.com. Account already exists."

        # Execute the command
        result = base_test_case.account_control.add_account(username, password, website)

        # 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 when account already exists.")

if __name__ == "__main__":
    pytest.main([__file__])



--- 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, 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_delete_account_success(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.delete_account') as mock_delete:
        # Setup mock return and expected outcomes
        account_id = 1
        mock_delete.return_value = True
        expected_entity_result = "Account with ID 1 deleted successfully."
        expected_control_result = "Account with ID 1 deleted successfully."

        # Execute the command
        result = base_test_case.account_control.delete_account(account_id)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_delete.return_value}")
        assert mock_delete.return_value == True, "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_delete_account_not_found(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.delete_account') as mock_delete:
        # Setup mock return and expected outcomes
        account_id = 999
        mock_delete.return_value = False
        expected_control_result = "Failed to delete account with ID 999."

        # Execute the command
        result = base_test_case.account_control.delete_account(account_id)

        # 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 account not found.\n")

async def test_delete_account_failure_entity(base_test_case):
                                with patch('DataObjects.AccountDAO.AccountDAO.delete_account',
side_effect=Exception("Failed to delete account in DAO")) as mock_delete:
        # Setup expected outcomes
```

```
    account_id = 1
    expected_control_result = "Error deleting account."

    # Execute the command
    result = base_test_case.account_control.delete_account(account_id)

    # 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_delete_account_failure_control(base_test_case):
    # This simulates a failure within the control layer
                                with patch('control.AccountControl.AccountControl.delete_account',
side_effect=Exception("Control Layer Failed")) as mock_control:

    # Setup expected outcomes
    account_id = 1
    expected_control_result = "Control Layer Exception: Control Layer Failed"

    # Execute the command and catch the raised exception
    try:
        result = base_test_case.account_control.delete_account(account_id)
    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_fetch_account_by_website.py ---
import pytest
import 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_fetch_account_by_website_success(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website') as mock_fetch:
        # Setup mock return and expected outcomes
        website = "example.com"
        mock_fetch.return_value = ("sample_username", "sample_password")
        expected_entity_result = ("sample_username", "sample_password")
        expected_control_result = ("sample_username", "sample_password")
```

```
        # Execute the command
        result = base_test_case.account_control.fetch_account_by_website(website)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_fetch.return_value}")
        assert mock_fetch.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_fetch_account_by_website_no_account(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website') as mock_fetch:
        # Setup mock return and expected outcomes
        website = "nonexistent.com"
        mock_fetch.return_value = None
        expected_control_result = "No account found for nonexistent.com."

        # Execute the command
        result = base_test_case.account_control.fetch_account_by_website(website)

        # 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 no account found.\n")


async def test_fetch_account_by_website_failure_entity(base_test_case):
                with patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website',
side_effect=Exception("Database Error")) as mock_fetch:
    # Setup expected outcomes
    website = "example.com"
    expected_control_result = "Error: Database Error"

    # Execute the command
    result = base_test_case.account_control.fetch_account_by_website(website)

    # 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_fetch_account_by_website_failure_control(base_test_case):
                with patch('control.AccountControl.AccountControl.fetch_account_by_website',
side_effect=Exception("Control Layer Error")) as mock_control:
    # Setup expected outcomes
```

```
        website = "example.com"
        expected_control_result = "Control Layer Exception: Control Layer Error"

        # Execute the command and catch the raised exception
        try:
            result = base_test_case.account_control.fetch_account_by_website(website)
        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 failed to handle its own error correctly."
        logging.info("Unit Test Passed for control layer error handling.")


if __name__ == "__main__":
    pytest.main([__file__])


--- unitTest_fetch_all_accounts.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_fetch_all_accounts_success(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts') as mock_fetch_all:
        # Setup mock return and expected outcomes
            mock_fetch_all.return_value = [(1, "user1", "pass1", "example.com"), (2, "user2", "pass2",
"test.com")]
            expected_entity_result = "Accounts:\nID: 1, Username: user1, Password: pass1, Website:
example.com\nID: 2, Username: user2, Password: pass2, Website: test.com"
        expected_control_result = expected_entity_result

        # Execute the command
        result = base_test_case.account_control.receive_command("fetch_all_accounts")

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_fetch_all.return_value}")
            assert mock_fetch_all.return_value == [(1, "user1", "pass1", "example.com"), (2, "user2",
"pass2", "test.com")], "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_fetch_all_accounts_no_accounts(base_test_case):
```

```
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts') as mock_fetch_all:
        # Setup mock return and expected outcomes
        mock_fetch_all.return_value = []
        expected_control_result = "No accounts found."

        # Execute the command
        result = base_test_case.account_control.receive_command("fetch_all_accounts")

        # 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 no accounts found.\n")

async def test_fetch_all_accounts_failure_entity(base_test_case):
                                with patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts',
side_effect=Exception("Database Error")) as mock_fetch_all:
        # Setup expected outcomes
        expected_control_result = "Error fetching accounts."

        # Execute the command
        result = base_test_case.account_control.receive_command("fetch_all_accounts")

        # 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.")


if __name__ == "__main__":
    pytest.main([__file__])



--- 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_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__])
```