testing

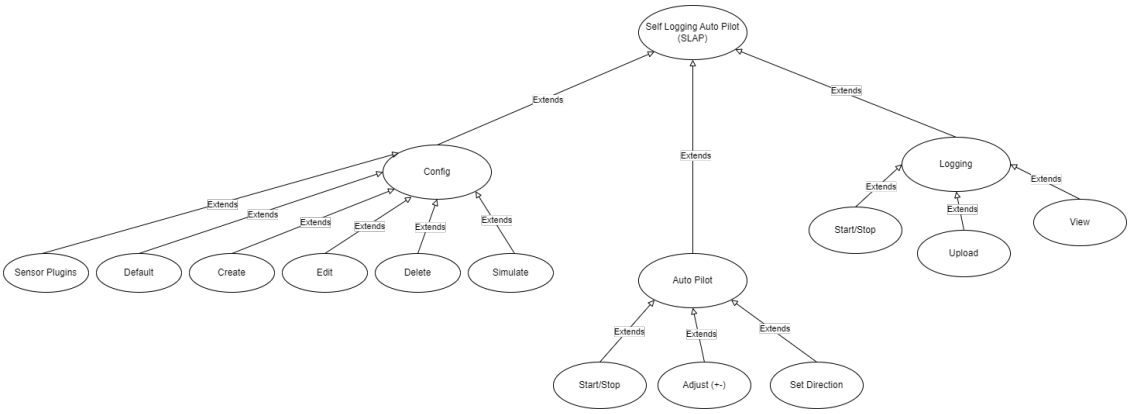
March 23, 2025

1 Testing

1.1 Introduction

The testings sections of this report details of how the SLAP project was tested. My test plan includes *manual* test which involved using the web interfaces and checking the results. Where appropriate, the testing also includes *unit testing* which is a code based test to verify that the code works as expected.

The test are organised to cover all the different module in the system. These modules are shown in the use-case diagram below.



1.2 Index of Testing

The following index shows all the modules and the use cases in the diagram and the tests for each one completed. The tables are an index to the manual test and the associated unit tests.

1.2.1 Config

The following tests verify the systems configuration functions. The configuration system allows different settings to be saved for different sea conditions.

Test Number	Use Case	Summary	Type	Result
1	Default	The system behaviour when no config is present	Manual	PASS
1.1		Function to correctly create default config	Unittest	PASS
2	Create	User enters config values for a new control configuration	Manual	PASS
2.1		Function to correctly create custom config	unittest	PASS
3	Edit	User changes a configuration's values	Manual	PASS
3.1		Function to correctly edit existing config	unittest	PASS
4	Delete	User deletes a configuration	Manual	PASS
4.1		Function to correctly delete existing config	unittest	PASS
5	Simulate	User enables simulator mode for the selected config	Manual	PASS
6	Add Plugin	Adds a sensor definition and plugin code	Manual	PASS

1.2.2 Auto Pilot

The following tests verify the auto pilot functions. The auto pilot funtion controls the boat using the tiller automatically. The system can work in a simulated (SIM) mode or in a real control (REAL) mode.

Test Number	Use Case	Summary	Type	Result
1	Start/Stop	User starts or stops the autopilot	manual	PASS

Test Number	Use Case	Summary	Type	Result
1.1		Function to correctly start/stop autopilot	unittest	PASS
2	Adjust (+-)	User adjusts the autopilot settings	manual	PASS
2.1		Function to correctly adjust target angle	unittest	PASS
3	Set Direction	User sets the direction for the autopilot	manual	PASS
3.1		Function to correctly set target direction	unittest	PASS

1.2.3 Logging

The following test verfit the logging functions the system. These function create a log of the geographic position in logditude and latitude at requlars time intervals. The log is stored in the database and can be uploaded to a cloud based mapping system.

Test Number	Use Case	Summary	Type	Result
1	Start/Stop	User starts or stops the logging	manual	PASS
1.1		Function to correctly start or stop the logging	unit test	PASS
2	Upload	User uploads the log data	manual	PASS
3	View	User views the log data	manual	PASS

1.3 Evidence

This section shows the relevant evidence for each test in the index. It includes a testing procedure and a description of the test. There are two types of test used:

- A *manual* test procedure where the application is driven by the user and the outcome is seen in the screenshots.
- A *unit test* is code which tests the relavant functions to the user case.

Unit tests are created using a Python testing framework called PyTest. All the test in the system are run and a report is generated. The comments in the unit test code detail the process and prints the results. It is important to read the code for details of these tests.

1.3.1 Config

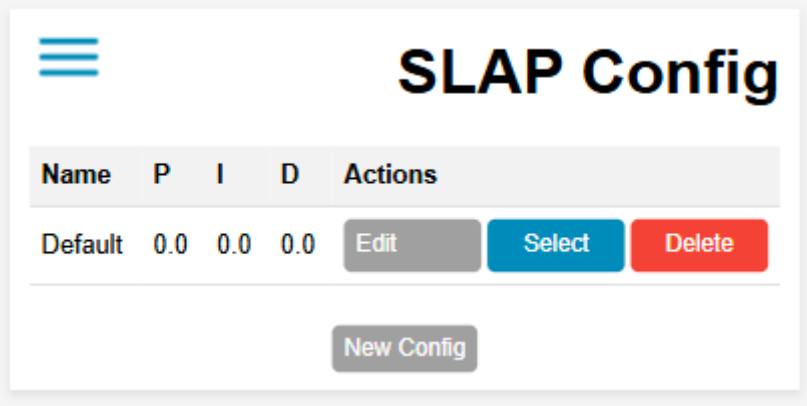
Test 1: Default

Description	Expected Outcome	Test Type
When the system is first started, no user config has been create and so the database will be empty. The system detects this, it creates a default config which is added to the database	Created default config	Manual

Procedure:

1. Clear database
2. Start application
3. Verify a default config has been added

Result: The screenshot shows a default config has been created when no config was present before.



1.3.2 Unit Test

```
[ ]: # %load C:
      ↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_defaultConfig.
      ↪py
     def test_getCurrentConfig_creates_default_when_none_exists(self):
             """Test that getCurrentConfig creates a default config when none␣
      ↪exists"""
             # Get current config (should create default)
             config = self.store.getCurrentConfig()

             # Verify default config was created
             self.assertEqual(config.name, 'Default')

             # Verify config was saved to database
```

```
        self.store.cursor.execute("SELECT * FROM CONFIGS WHERE isDefault =␣
↪True")
        saved_config = self.store.cursor.fetchone()
        self.assertIsNotNone(saved_config)
        self.assertEqual(saved_config['name'], 'Default')
        self.assertEqual(saved_config['proportional'], 0)
        self.assertEqual(saved_config['integral'], 0)
        self.assertEqual(saved_config['differential'], 0)
```

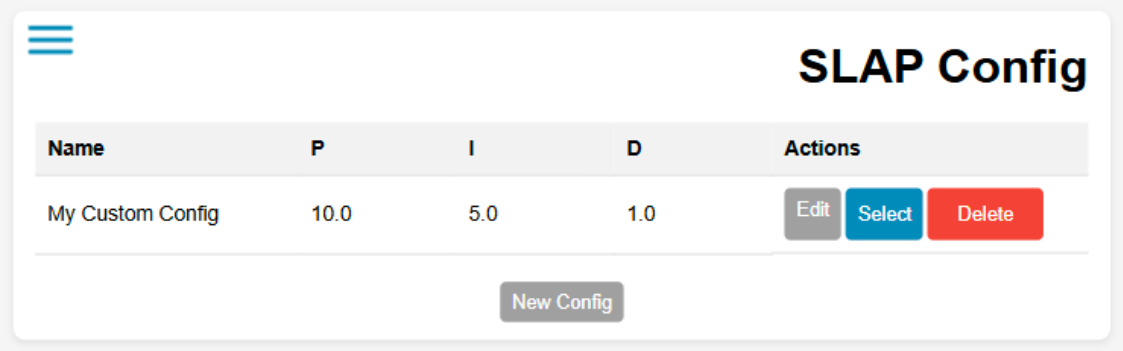
Test 2: Create

Description	Expected Outcome	Test Type
Slap provides the facility to create custom configs, the user can create a config and save it to the database to be selected for later use	Normal	Config is saved to database

Procedure:

1. Visit config page
2. Press create Config
3. Enter all needed values in the form
4. Press save
5. View saved config in database

Result: The screenshot shows a custom config has been created



Unit Test

```
[ ]: # %load C:
↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_createConfig.
↪py
from services.slapStore import SlapStore, Config

def test_createConfig_creates_new_config(self):
```

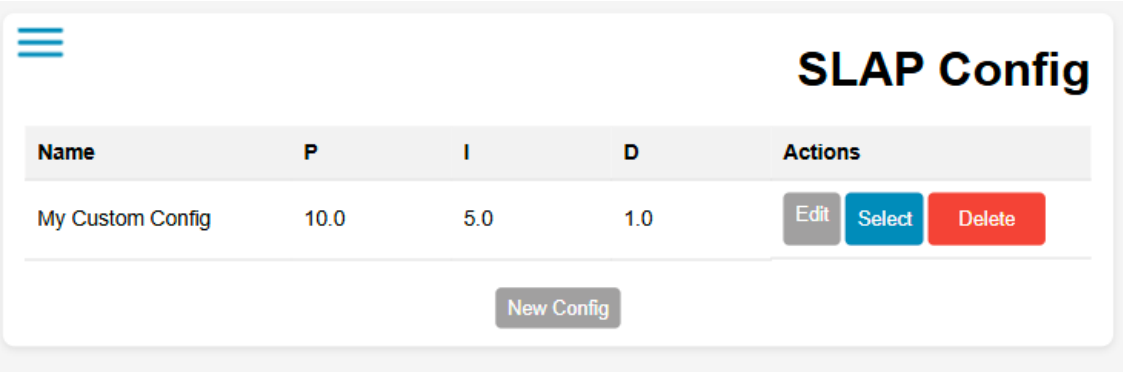
```
        """Test that createConfig creates a config"""
        print("--------------------------------")
        print("")
        print("UNIT TEST: Config_Create")
        print("\nTesting creation of new config...")

        # Get current config (should create default)
        config = Config(0, "My Custom Config", 10, 5, 1)

        config = self.store.newConfig(config)

        self.assertEqual(config.name, 'My Custom Config')

        # Verify config was saved to database
        self.store.cursor.execute(f"SELECT * FROM CONFIGS WHERE configId =␣
↪'{config.configId}'")
        saved_config = self.store.cursor.fetchone()
        self.assertIsNotNone(saved_config)

        print("Verifying saved values match input values...")
        self.assertEqual(saved_config['name'], 'My Custom Config')
        self.assertEqual(saved_config['proportional'], 10)
        self.assertEqual(saved_config['integral'], 5)
        self.assertEqual(saved_config['differential'], 1)
        print("All values verified successfully")
```

Test 3: Edit

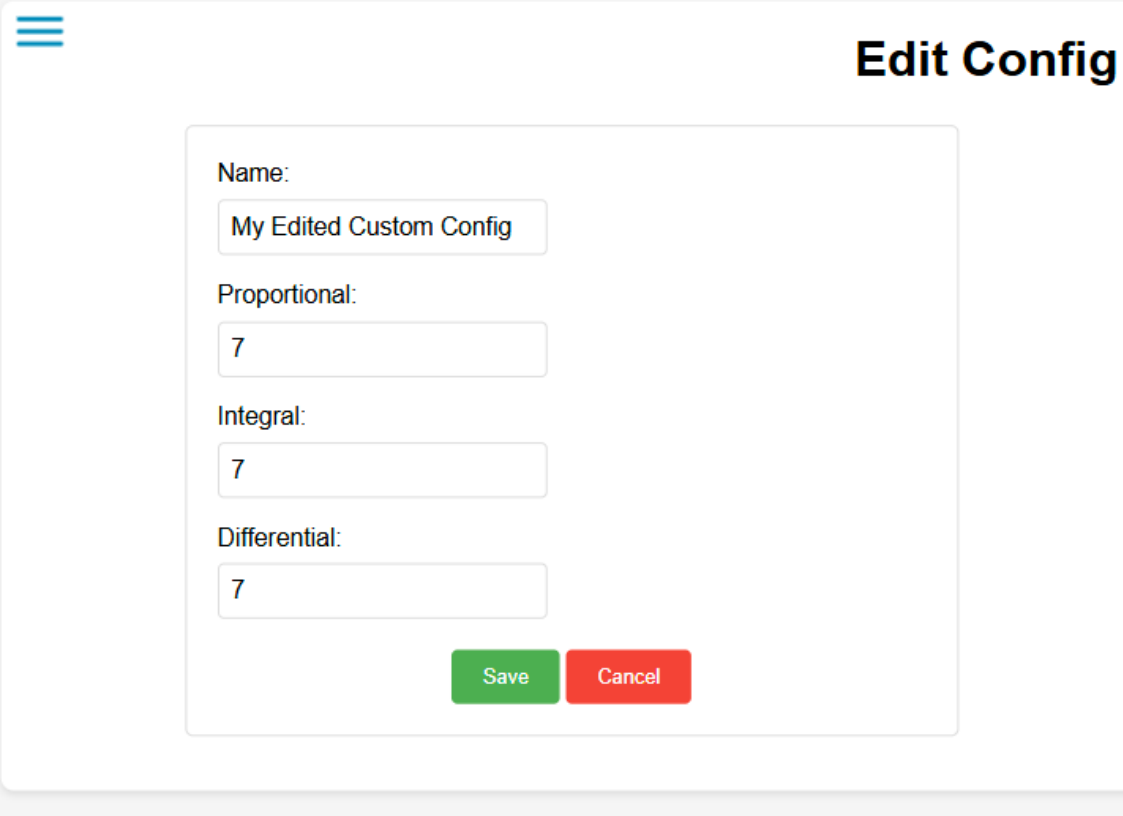
Description	Data Type	Expected Outcome	Test Type
User changes a configuration's values	Normal	Edited config is saved to database	Manual

Procedure:

1. Visit config page

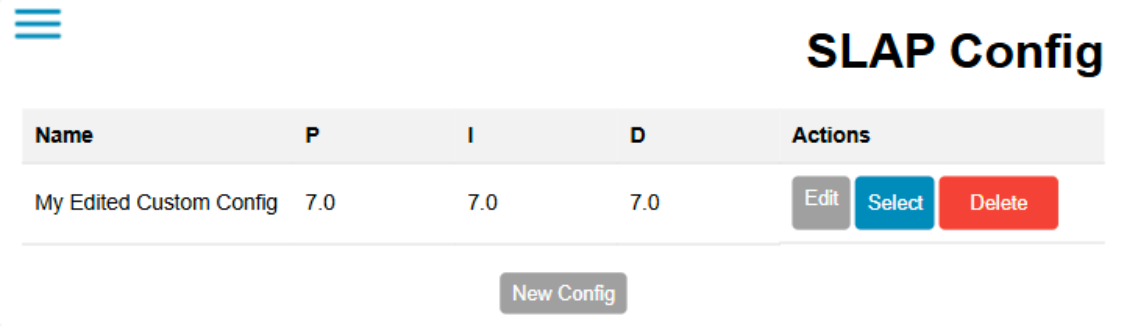


2. Press edit Config



3. Adjust needed values in the form
4. Press save
5. View saved config in database

Result



Unit Test

```
[ ]: # %load C:
     ↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_editConfig.
     ↪py
     from services.slapStore import SlapStore, Config
```

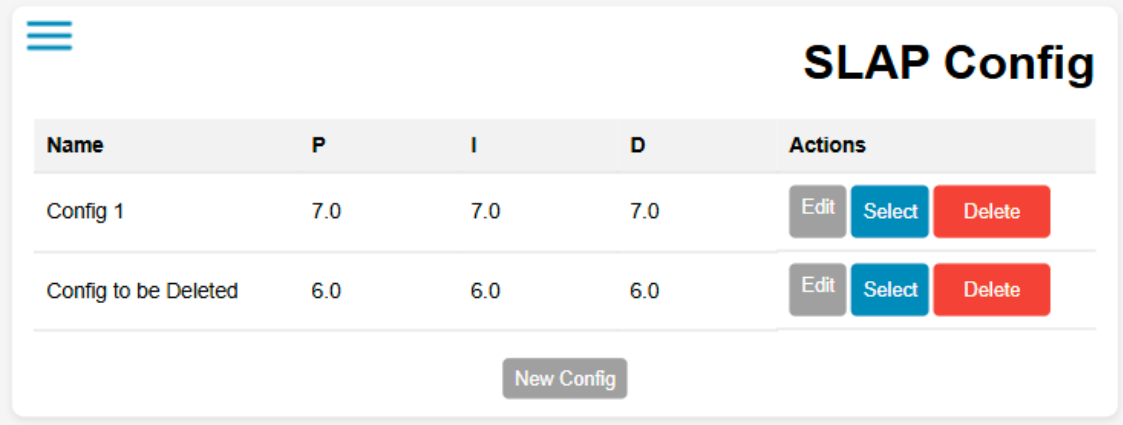
```
def test_editConfig_updates_existing_config(self):
        """Test that editConfig updates an existing config"""
        print("--------------------------------")
        print("")
        print("UNIT TEST: Config_Edit")
        print("\nTesting editing of existing config...")

        # Create initial config
        initial_config = Config(0, "Test Config", 1, 2, 3)
        initial_config = self.store.newConfig(initial_config)

        # Edit the config
        edited_config = Config(initial_config.configId, "Edited Config", 10,␣
↪20, 30)
        self.store.updateConfig(edited_config)

        # Verify config was updated in database
        self.store.cursor.execute(f"SELECT * FROM CONFIGS WHERE configId =␣
↪'{initial_config.configId}'")
        saved_config = self.store.cursor.fetchone()
        self.assertIsNotNone(saved_config)

        print("Verifying saved values match edited values...")
        self.assertEqual(saved_config['name'], 'Edited Config')
        self.assertEqual(saved_config['proportional'], 10)
        self.assertEqual(saved_config['integral'], 20)
        self.assertEqual(saved_config['differential'], 30)
        print("All values verified successfully")
```

Test 4: Delete

Description	Data Type	Expected Outcome	Test Type
User deletes a configuration	Normal	Config is removed from database	Manual

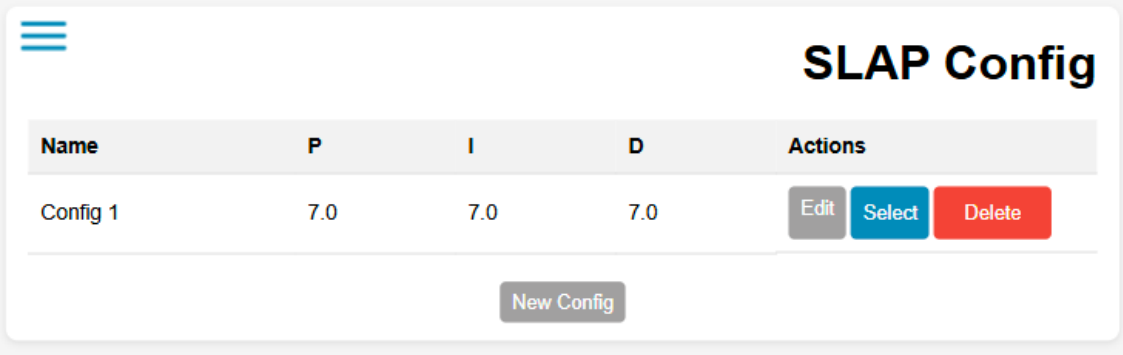
Procedure:

1. Visit config page



2. Press delete on a config row
3. View configs in database to verify deletion

Result



Unit Test

[ ]:
```
# %load C:
  ↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_deleteConfig.
  ↪py
from services.slapStore import SlapStore, Config

def test_deleteConfig_removes_existing_config(self):
        """Test that deleteConfig removes an existing config"""
        print("--------------------------------")
        print("")
        print("UNIT TEST: Config_Delete")
        print("\nTesting deletion of existing config...")

        # Create initial config
        initial_config = Config(0, "Test Config", 1, 2, 3)
        initial_config = self.store.newConfig(initial_config)

        # Delete the config
        self.store.deleteConfig(initial_config.configId)
```

```
        # Verify config was deleted from database
        self.store.cursor.execute(f"SELECT * FROM CONFIGS WHERE configId =␣
↪'{initial_config.configId}'")
        deleted_config = self.store.cursor.fetchone()

        print("Verifying config was deleted...")
        self.assertIsNone(deleted_config)
        print("Config deletion verified successfully")
```

Test 5: Simulate

Description	Data Type	Expected Outcome	Test Type
User enables simulator mode for the selected config	Normal	Simulator is started	Manual

Procedure:

1. Visit config page
2. Press simulate on a selected config row
3. Verify simulator starts with displays config name

[Screenshot of config in database]

Test 6: Add Plugin

Description	Data Type	Expected Outcome	Test Type
Adds a sensor definition and plugin code	Normal	Plugin is saved to database	Manual

[Screenshot of config in database]

Auto Pilot Test 1: Start/Stop

Description	Data Type	Expected Outcome	Test Type
User starts or stops the autopilot	Normal	Autopilot starts or stops	Manual

Procedure:

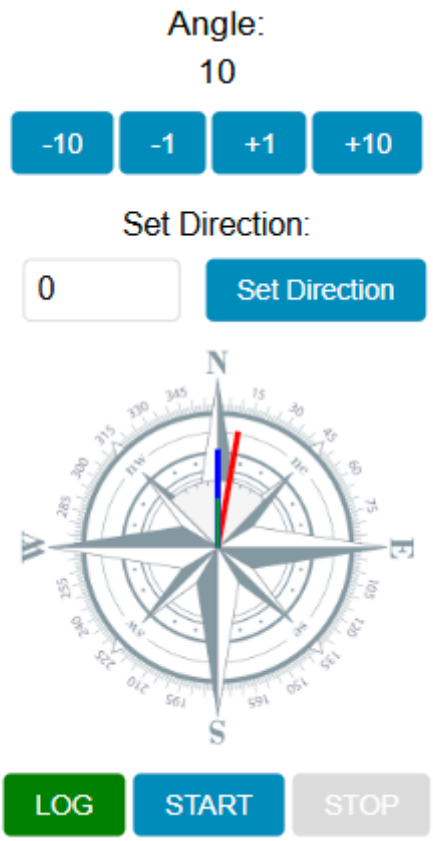
1. Press start/stop button
2. Verify autopilot status changes

Test 2: Adjust (+-)

Description	Data Type	Expected Outcome	Test Type
User adjusts the autopilot settings	Normal	Settings are adjusted	Manual

Procedure:

2. Press +/- buttons to adjust settings
3. Verify target heading changes correctly



Unit Test

```
[ ]: # %load slap/src/iteration2/tests/test_adjustAutoPilot.py
from control.autoPilot import AutoPilot

def test_adjust_target_angle(self):
    """Test that the +/-10 buttons correctly adjust the target angle"""
    # Create test autopilot with initial target angle of 0
    auto_pilot = AutoPilot()
    auto_pilot.setHeading(0)

    # Test +10 button
    heading = auto_pilot.setHeading(auto_pilot.getHeadings()['target'] + 10)
    assert heading == 10

    # Test -10 button
    heading = auto_pilot.setHeading(auto_pilot.getHeadings()['target'] - 10)
    assert heading == 0
```

Test 3: Set Direction

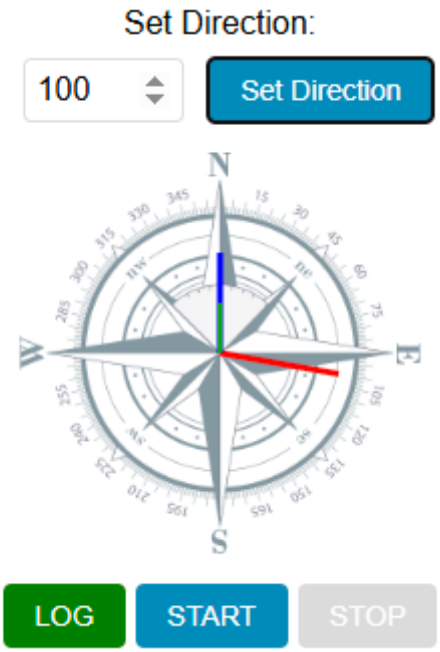
Description	Data Type	Expected Outcome	Test Type
User sets the direction for the autopilot	Normal	Direction is set	Manual

Procedure:

2. Enter desired heading in degrees (0-359)

3. Press set button
4. Verify target heading updates to entered value

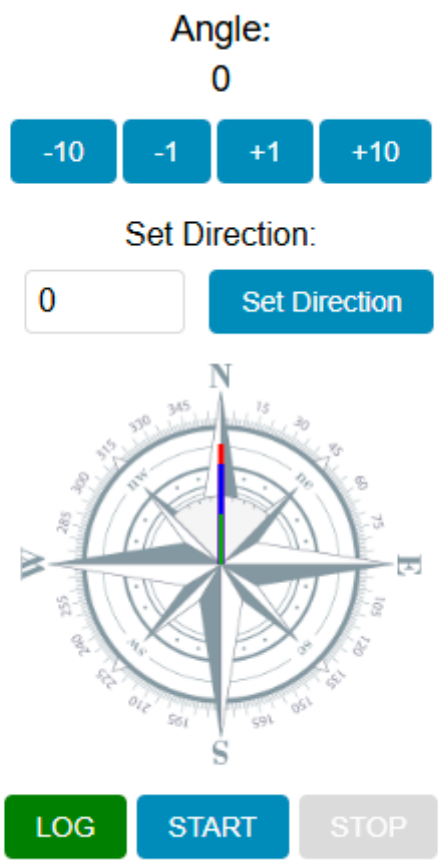
Result:



Description	Data Type	Expected Outcome	Test Type
User sets the direction for the autopilot	Errornous	Direction is set	Manual

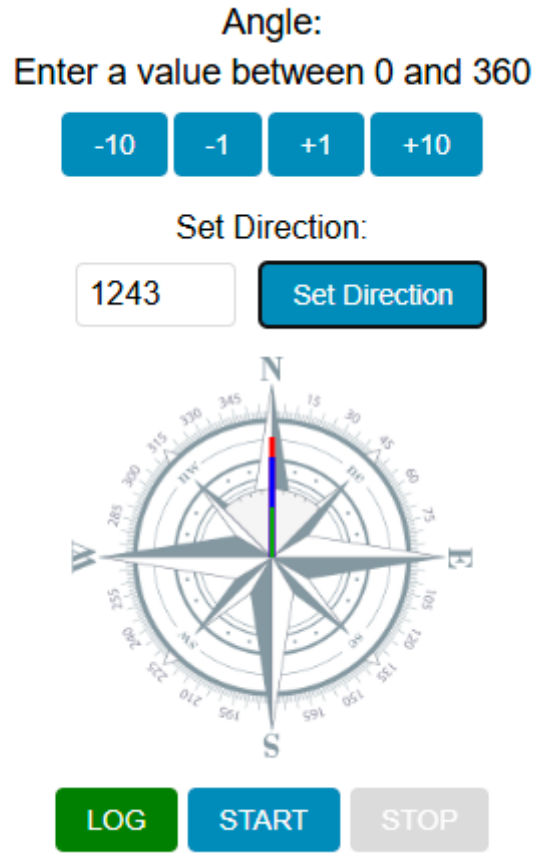
Procedure:

2. Enter errornous heading in degrees (<0 or >360)
3. Press set button



4. Verify that the system returns an error, prompting the user to enter another value

Result:



Unit Test

```
[ ]: # %load C:
      ↪\Users\franc\vscode\projects\slap\slap\src\iteration2\tests\test_setAutoPilot.
      ↪py
    from control.autoPilot import AutoPilot
    from web.app import WebServer
    from services.logger import Logger
    from services.mapManager import MapManager
    from transducers.gps import Gps
    from services.slapStore import SlapStore
    import unittest
    import json

    def test_set_direction(self):
        print("\nTesting setDirection endpoint...")

        # Create test client and make request
        self.auto_pilot = AutoPilot()
        self.logger = Logger(Gps(), MapManager())
        self.web_server = WebServer(self.auto_pilot, self.logger)
```

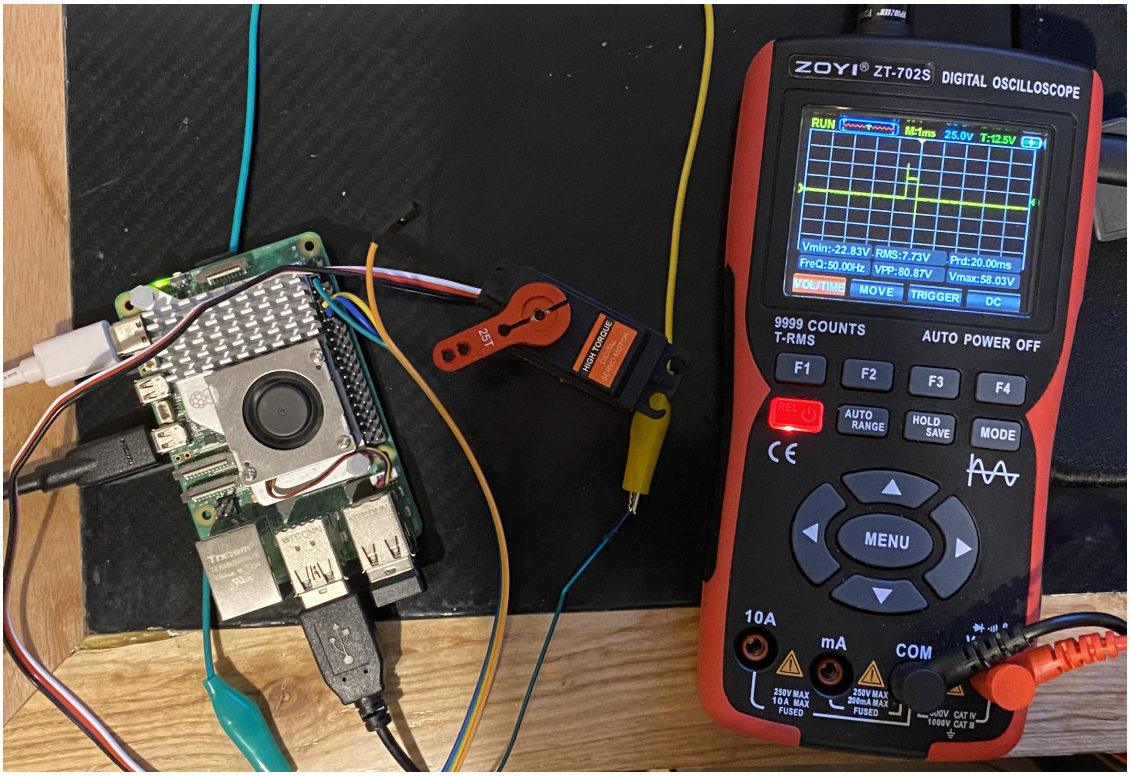
```
    app = self.web_server.create_server(self.store)
    self.client = app.test_client()
    response = self.client.put('/api/setDirection', data='180')
    self.assertEqual(response.status_code, 200)
    data = json.loads(response.data)
    self.assertEqual(data['angle'], '180')
    self.assertEqual(self.auto_pilot.getHeadings()['target'], 180)

    response = self.client.put('/api/setDirection', data='90')
    self.assertEqual(response.status_code, 200)
    data = json.loads(response.data)
    self.assertEqual(data['angle'], '90')
    self.assertEqual(self.auto_pilot.getHeadings()['target'], 90)

    print("Valid heading tests passed")
```

Test 3: Set Direction (Hardware)

Description	Data Type	Expected Outcome	Test Type
User sets the direction for the autopilot	Normal	Direction is set	Manual

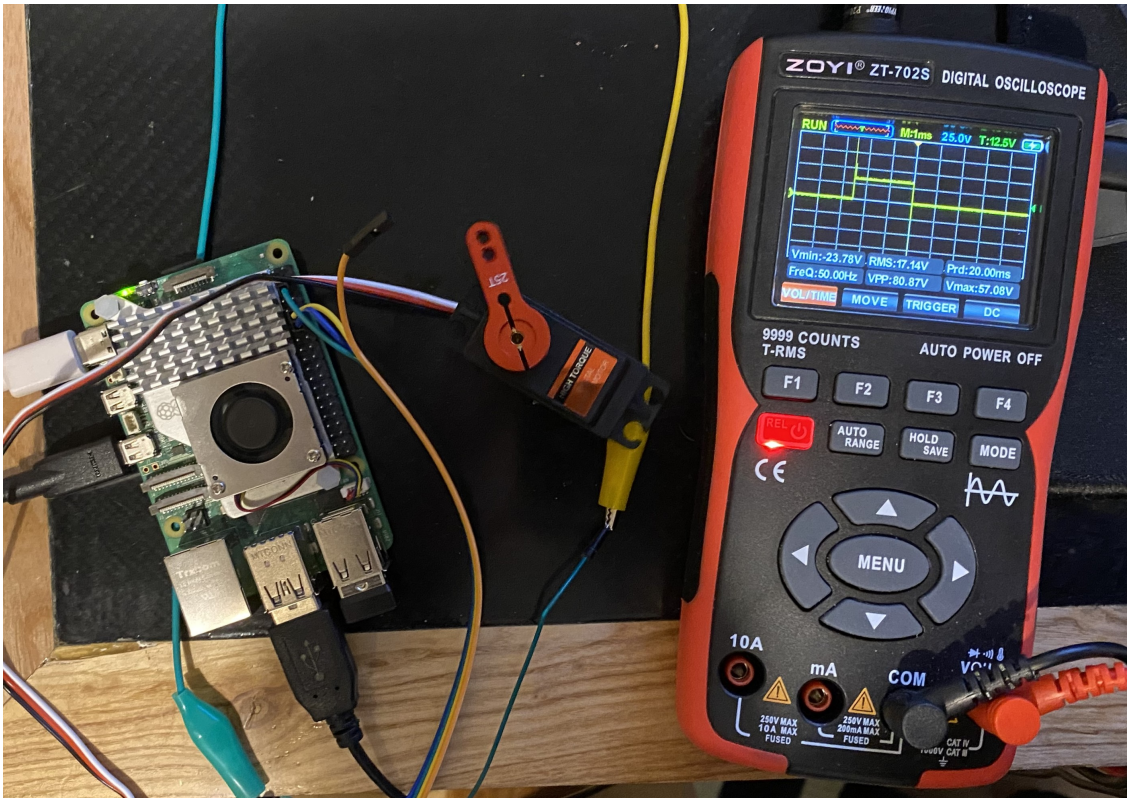
Procedure:

1. Enter Heading so tiller turns to left
2. Check PWM response and motor position



The image shows the servo motor in the left position, and the oscilloscope trace with a 0.5ms pulse.

3. Change Heading so tiller turns to right



The image shows the servo motor in the right position, and the oscilloscope trace with a 2.5ms pulse.

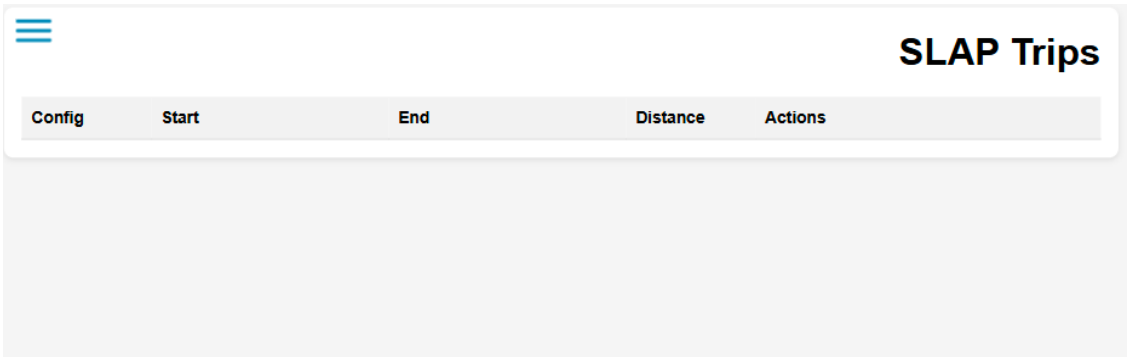
4. Check PWM response and motor position

Logging Test 1: Start/Stop

Description	Data Type	Expected Outcome	Test Type
User starts or stops the logging	Normal	Logging starts or stops	Manual

Procedure:

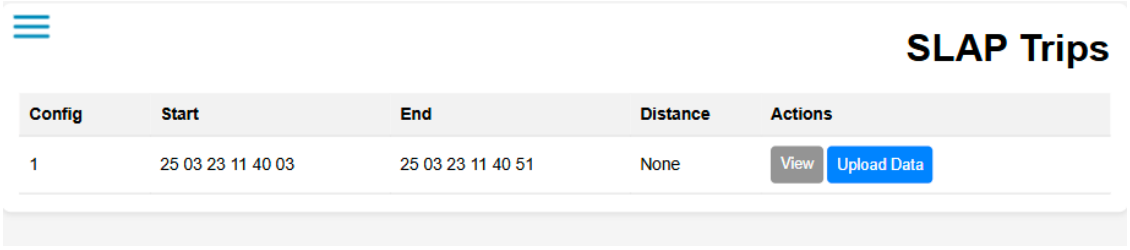
1. Verify no trip is present



2. Press start/stop button and check UI updates

3. Check trip is created/closed appropriately

Result:



Unit Test The unit test code below verifies the underlying methods for this functionality

[ ]:
```
# %load slap/src/iteration2/tests/test_startStopLogging.py
from services.logger import Logger
from transducers.gps import Gps
from services.mapManager import MapManager
from services.slapStore import Config

def test_startStopLogging_creates_new_trip(self):
    """Test that starting and stopping logging creates a new trip"""
    # Create test components
    gps = Gps()
    map_manager = MapManager()
    logger = Logger(gps, map_manager)
    logger.setStore(self.store)
```

```
    # Start logging
    logger.start(Config(0, 'default', 0, 0, 0))
    assert logger.running == True

    # Stop logging
    logger.stop()
    assert logger.running == False

    # Verify a new trip was created in the database
    self.store.cursor.execute("SELECT COUNT(*) FROM Trip")
    trip_count = self.store.cursor.fetchone()[0]
    assert trip_count == 1
```

Test 2: Upload

Description	Data Type	Expected Outcome	Test Type
User uploads the log data	Normal	Log data is uploaded	Manual

Procedure:

2. Press upload button
3. Verify data is sent to server
4. Verify data appears in cloud storage

Result:

Test 3: View

Description	Data Type	Expected Outcome	Test Type
User views the log data	Normal	Log data is displayed	Manual

Procedure:

2. Press view button
3. Display map view to show trip

[Screenshot of log data]