Testdokumentation

Die folgenden Seiten befassen sich mit dem Testen des Systems. Ziel ist es die Qualitaet des Systems nachzuweisen, sowie die Teststrukturen zu dokumentieren.

Inhaltsverzeichnis

1 Testkonzept

'Teststrategie'

In diesem Punkt wird analysiert, in welchen Systemkomponenten das Testen am wichtigsten ist. Diese Komponenten werden spaeter fuer die entstehenden Tests priorisiert betrachtet. Da es fuer uns nicht moeglich war, eine 100 prozentige Testabdeckung zu gewaehrleisten, haben wir uns fuer dieses Vorgehen entschieden. Um einen Ueberblick ueber die Testfaelle zu bekommen, wurde eng mit dem Entwicklerteam zusammengearbeitet. Es wurde das komplette System durchleuchtet und die Kernfunktionen bestimmt. Diese Kernfunktionen sollen durch die Tests abgedeckt werden koennen, um ein fehlerfreies Laufen des Systems sicherzustellen. Entstandene Testfaellen wurden in einer Liste festgehalten. Diese Testfaelle beschreiben:

- Einzuhaltende Vorbedingung
- Eingabedaten
- Erwartetes Ergebnis

Zudem wurden sie noch einmal zusaetzlich in manuelle- und automatische Tests unterteilt. Der Anwendungsbereich fuer die automatisierten Testfaelle liegt ausschliesslich im BackEnd und die manuellen Testfaelle im FrontEnd.

a) Testobjekte

Table 1. Testfaelle BackEnd

Testfall	Vorbedingung	Eingabedaten	Ergebnis
Django Rest-Schnittstelle	Backend mit Rest-API läuft	JSON-File mit den (Wetter)daten	Die Daten aus der JSON-File werden korrekt in die versch. Tabellen aufgeteilt. Jede Tabelle wurde mit neuen Werten erweitert.
QueryByime-Funktion	Der Server muss laufen & die Tabellen muessen leer sein	Drei Eintraege mit versch. Daten	Es werden immer nur die Eintraege erkannt, die in dem gewaehlten Zeitraum liegen.
Mapping der Sensordaten	Server muss laufen	JSON-File mit den (Wetter)daten	Die Kuerzel 'deg' & 'dir' werden zu den Begriffen 'degrees' & 'direction' und das Wort 'timestamp' wird zu 'measure time'
Rasberry Pi Skript	Backend mit Rest-API läuft	Sensor Daten im definierten JSON-Format	Datenbank Tabellen wurden mit den entsprechenden Datensätzen gefüllt

b) Testmethoden

'Testumgebung'

Die Testumgebung beschreibt, welche Werkzeuge fuer das Testen eingesetzt werden. Es wurde hauptsaechlich mit UnitTests in Python gearbeitet. Dazu kamen noch weitere Integrationstests um das genutzte Django-Framework zu testen. Damit wurden alle Sachen aus dem Backend abgedeckt. Das FrontEnd wurde mit sogenannten User-Acceptance-Tests(End-User-Tests) bearbeitet. Um dies sinnvoll zu gestalten wurde eine Liste mit versch. Testfaellen erstellt um die Konsistenz zu sichern. Bei der Uebergabe gab es noch Abnahmetests zusammen mit dem Kunden (Ergebnisse im Punkt 'Testergebnisse'). Dazu wurden auch die Testfaelle aus den FrontEndTests genutzt.

'Testorganisation'

Wann und Wie wird getestet? Wir haben ein Concurrent Testing angestrebt, dass bedeutet es wurden die Tests simultan zur Codeerweiterung entwickelt. Damit sollte gewaehrleistet werden, dass immer der groesstmoeglichste Teil des Systems durch Tests abgedeckt ist. Ausserdem soll dies

auch ein Verzug der Tests verhindern.

2 Testfaelle

a) Automatisierte UnitTests

Im ersten Testfall wird eine JSON-File an unsere Django-Schnittstelle gesendet. Diese ist dafuer da, die enthaltenden Daten auf die einzelnen Tabellen aufzuteilen. Um zu kontrollieren ob es tatsaechlich funktioniert hat, wird mit Hilfe von einzelnen 'assertEqual'-Abfragen geschaut, ob die Werte in den Tabellen gelangt sind. Moeglichkeiten waeren z.B. das Abfragen der Anzahl von Eintraegen in einer Tabelle, oder sogar nach bestimmten Werten in der Tabelle. Ausserdem wird der Statuscode abgefragt (HTTP_201_CREATED). Wenn dieser korrekt ist gibt das System eine Meldung aus.

```
TEST_DATA_DIR = 'testdata/1589990400.json'


def get_test_file():
    script_dir = os.path.dirname(__file__)  # <-- absolute dir the script is in
    abs_test_data_path = os.path.join(script_dir, TEST_DATA_DIR)
    test_file = open(abs_test_data_path)
    return test_file


class TestSensorDataPost(APITestCase):
    """
    Ensure that the data is spread correctly.
    """
    def test_sensor_data_was_posted(self):
        test_file = get_test_file()
        data = json.load(test_file)

        response = self.client.post('/api/sensor-data', data, format='json')

        self.assertEqual(response.status_code, status.HTTP_201_CREATED, 'Uploading the Json-File was successful.')
        self.assertEqual(Wind.objects.count(), 1)
        self.assertEqual(Temperature.objects.count(), 1)
        self.assertEqual(Battery.objects.count(), 1)
        self.assertEqual(SolarCell.objects.count(), 1)
        self.assertEqual(MeasurementSession.objects.count(), 1)

        self.assertEqual(Temperature.objects.get().degrees, 20.5)
        self.assertEqual(Wind.objects.get().speed, 1.5)
```

Figure 1. Test der Rest-Schnittstelle

Die QueryByTime-Funktion wird dadurch getestet, dass wir zuerst drei versch. Eintraege posten. Danach schauen wir ob diese drei Eintraege erfolgreich erstellt wurden, indem wir die Anzahl mit der Zahl '3' vergleichen. Wenn dies erfolgreich war, setzten wir nun ein festen Start- & Endpunkt fest. Nun folgen wieder assertEqual-Abfragen wo wir gezielt nur nach einzelnen Eintraegen suchen, dabei wird geprueft ob wirklich nur die zu erwarteten Datensaetze zurueckgegeben werden.

```python
def test_query_by_time(self):
    """Creating random Temperatures"""
    time_1 = datetime.now() - timedelta(days=5)
    temp_1 = Temperature(degrees=10, measurement_session=None, measure_time=time_1)
    temp_1.save()

    time_2 = datetime.now()
    temp_2 = Temperature(degrees=10, measurement_session=None, measure_time=time_2)
    temp_2.save()

    time_3 = datetime.now() - timedelta(days=100)
    temp_3 = Temperature(degrees=10, measurement_session=None, measure_time=time_3)
    temp_3.save()

    """
    Ensure we can filter by dates.
    """
    self.assertEqual(Temperature.objects.count(), 3)

    start = datetime.now() - timedelta(10)
    filtered_temps = self.client.get('/api/temps/?start=' + str(start))
    self.assertEqual(len(filtered_temps.data), 2)

    end = datetime.now() - timedelta(2)
    filtered_temps = self.client.get('/api/temps/?start=' + str(start) + '&end=' + str(end))
    self.assertEqual(len(filtered_temps.data), 1, 'The filtering was successful.')
```

Figure 2. Test der Datumsabfrage

Der dritte Testfall kontrolliert, ob die in der JSON-File enthaltenen Kuerzel ordentlich gemappt werden. Bei den drei assert-Abfragen ist das Vorgehen immer das gleiche. Mit dem Befehl 'assertIsNotNone' wird geschaut, dass das erwuenschte Wort vorhanden ist. Falls nicht kommt eine Fehlermeldung, die den Nutzer darauf hinweist.

```python
def test_senor_data_mapping(self):
    """
    Ensure that the acronyms are mapped correctly
    """
    data = json.load(get_test_file())
    session_id = int(data['session_id'])
    session = MeasurementSession(session_id=session_id)
    map_sensor_data(data, session)

    temp_dict = data['temp']
    with self.assertRaises(KeyError):
        temp_dict['deg']
    self.assertIsNotNone(temp_dict['degrees'], 'Temperature dict was not mapped correctly.')

    wind_dict = data['wind']
    with self.assertRaises(KeyError):
        wind_dict['dir']
    self.assertIsNotNone(wind_dict['direction'], 'Wind dict was not mapped correctly.')

    battery_dict = data['battery']
    with self.assertRaises(KeyError):
        battery_dict['timestamp']
    self.assertIsNotNone(wind_dict['measure time'], 'Battery dict was not mapped correctly.')
```

Figure 3. Test zum Mapping

Im folgenden Test wird die Korrektheit des Raspi-Skripts getestet, welches die Sensor-Daten per Post-Request an den Webserver uebermittelt. Dabei sammelt es die vorliegenden JSON Dateien ein und postet diese an den Server. Die Daten werden an den entsprechenden Rest-Endpunkt uebermittelt, welcher dann die Aufteilung der Daten in die einzelnen Datenbank-Tabellen vornimmt.

```python
def test_sensor_data_was_posted_by_skript(self):
    collect_and_post_data.SEARCH_DIR = './testdata/unmapped.json'
    collect_and_post_data.find_and_post_data()

    self.assertEqual(Wind.objects.count(), 1)
    self.assertEqual(Temperature.objects.count(), 1)
    self.assertEqual(Battery.objects.count(), 1)
    self.assertEqual(SolarCell.objects.count(), 1)
    self.assertEqual(Load.objects.count(), 1)
    self.assertEqual(MeasurementSession.objects.count(), 1)

    test_data = json.load(get_test_file('mapped'))
    self.assertEqual(Temperature.objects.get().degrees, test_data['temperatrure']['degrees'])

    self.assertEqual(Wind.objects.get().speed, test_data['wind']['speed'])
    self.assertEqual(Wind.objects.get().direction, test_data['wind']['dir'])

    self.assertEqual(Battery.objects.get().voltage, test_data['battery']['voltage'])
    self.assertEqual(Battery.objects.get().current, test_data['battery']['current'])

    self.assertEqual(SolarCell.objects.get().voltage, test_data['pv']['voltage'])
    self.assertEqual(SolarCell.objects.get().power, test_data['pv']['power'])

    self.assertEqual(Load.objects.get().voltage, test_data['load']['voltage'])
    self.assertEqual(Load.objects.get().current, test_data['load']['current'])
```

Figure 4. Test des Raspi-Skripts

b) manuelle Tests

Hierfuer wurde eine Liste erstellt, die zum Testen des FrontEnds genutzt wurde. Sie soll alle Moeglichkeiten abdecken und die Resultate dokumentieren. Sie wird auch gleichzeitig fuer den Abnahmetest am Ende verwendet.

Table 2. Testfaelle FrontEnd

Testfall	Ergebnis
Beim klicken auf das "Wetterstation-Logo" gelangt man auf die Startseite	ja [ ] nein [ ]
Graphen werden beim Aufruf der Startseite mit den Standardeinstellungen erstellt (1 Tag bei den Temperaturdaten / 4h bei den Winddaten) und dargestellt	ja [ ] nein [ ]
Die angezeigten Daten sind (logisch) korrekt	ja [ ] nein [ ]
vorgefertigte Zeitraume (Auswahlfelder) lassen sich auswaehlen und aktualisieren den Graphen	ja [ ] nein [ ]
Detaillierte Ansicht der einzelnen Datenpunkte ist mit dem Cursor moeglich	ja [ ] nein [ ]
Erweiterter Modus funktioniert (selbststaendig einen Zeitraum bestimmen)	ja [ ] nein [ ]

Testfall	Ergebnis
Wenn der angegebene Zeitraum logisch falsch ist, wird der Graph NICHT aktualisiert	ja [ ] nein [ ]
Tab "Webcam&Galerie": Weiterleitung in die Galerie. das aktuellste Bild wird gross dargestellt und die aelteren Bilder sind nach Datum in Reitern geordnet	ja [ ] nein [ ]
Bei der Auswahl eines Bildes oeffnet sich eine "LightBox"	ja [ ] nein [ ]
Login mit korrektem Benutzernamen/Passwort leitet zum Wartungsbereich weiter	ja [ ] nein [ ]
Bei falschem Benutzernamen/Passwort wird die Login-Seite neu geladen	ja [ ] nein [ ]
Die Zu-/Abwahl von Graphen ist durch Auswahlfelder moeglich (Wartungsbereich), der Graph wird aktualisiert	ja [ ] nein [ ]
Die Achsenbeschriftung aendert sich mit der Auswahl der dargestellten Daten Wartungsbereich)	ja [ ] nein [ ]

3 Testergebnisse

Die Ergebnisse der Test waren eigentlich immer erfolgreich. Sie entstanden trotz Concurrent Testing mit leichter Verzoegerung und dienten dadurch eher zur Sicherung des Systems. Nach System- updates konnte man die Tests durchlaufen lassen und schauen, ob die bereits bestehenden Komponenten trotzdem noch funktionierten. Da unser Entwicklerteam durchgehend gute Arbeit geleistet hatte und sowohl bei der Entwicklung des Systems, als auch beim Support der Testentwicklung stets behilflich war, haben die Tests durchgehend die positive Entwicklung des Systems bestaetigt. Deshalb kam es zu keinen signifikanten Abweichungen, auf die man haette eingehen muessen. Die vermeintlichen Probleme sind durch die gute Zusammenarbeit zwischen dem Team und der Auftraggeber schon bereits frueh erkannt worden und konnten somit schon bei der Implementierung vermieden/geloest werden.

Abnahmetestergebnis