Projektdokumentation Drei

Embedded Software Development, Prof. Dr. Jochen Hertle

Stephanie Ehrenberg, Markus Hornung, Simon Jahreiß, Luis Morales, Maximilian Pachl

09.07.2015

Zusammenfassung

Viele Studenten haben ein Anliegen an die Fachschaft unserer Faktultät. Leider passiert es häufiger, dass die überarbeiteten und verkaterten Studenten umsonst in die Fachschaft laufen und feststellen müssen, dass niemand da ist. Deswegen soll die Präsenz der Fachschaftsmitglieder in Echtzeit auf einer Webseite und durch Leuchtsignale neben der Tür visualisiert werden. So können die Studenten ihre Zeit effektiver planen und die Noten werden automatisch besser.

Inhaltsverzeichnis

Allgemein **3**

Anforderungen . . . 3

Idee . . . 3

Hardware . . . 3

Software Design **4**

Wifi Crawler . . . 5

Hotspot . . . 5

Crawler . . . 5

Statusanzeige GPIO . . . 6

Logging . . . 6

Manager . . . 6

Das Datenbank-Modul mit SQLite3 . . . 7

Genereller Aufbau . . . 7

Datenbankentwurf . . . 7

Das SQLiteWrapper.py Modul . . . 8

Webserver: Flask . . . 10

Webinterface: AngularJS . . . 11

Peripheriesteuerung . . . 12

SoundController . . . 12

DMXHandler . . . 12

Farbverwaltung . . . 12

Testing **13**

Unittests mit PyUnit . . . 13

Generelles Setup . . . 13

Datenbank-Wrapper . . . 13

Systemtest . . . 14

Webinterface . . . 14

Peripherie . . . 14

Crawler . . . 14

Lasttest . . . 14

Anhänge **15**

Arbeitsaufteilung . . . 15

Sprint Protokolle . . . 15

Allgemein

Anforderungen

- Automatische Präsenzerkennung (ohne Interaktion von Fachschaftsmitglied)
- Keine zusätzliche Software/Hardware für Fachschaftler von Nöten
- Webinterface zur Verwaltung
- Zeitnahe Visualisierung der Präsenz (Webseite wie Leuchtsignale)
- individuelle Willkommensmelodie
- Audio-Visuelle Eindeutigkeit der Fachschaftler (Farbe der Leuchtsignale, Melodie beim Eintritt)
- Systemstatus von Außen ersichtlich

Idee

Da eigentlich jedes Mitglied der Fachschaft ein Smartphone hat, soll die Präsenzerkennung über die Wifi Schnitstelle realisiert werden. Dazu wird ein dedizierter Hotspot, der nur die Fachschaftsräume abdeckt, installiert. Um zu vermeiden, dass auf den Smartphones zusätzliche Software installiert werden muss, soll anhand der MAC-Adresse erkannt werden, ob das Telefon in das spezielle Wifi eingebucht ist und somit der Fachschaftler präsent ist oder nicht.

Sobald ein Telefon sich ins Wifi einbucht, soll über die in den Fachschaftsräumen installierte Tonanlage die Willkommenmelodie des jeweiligen Telefoninhabers/Fachschaftlers abgespielt werden.

Zusätzlich wird neben der Fachschaftstür ein Leuchtsignal (PixelTube, ein Pixel) mit der vom User spezifizierten Farbe eingeschaltet. Beim Verlassen der Fachschaftsräume soll dieses Signal ausgeschaltet werden.

Durch eine Websocketverbindung auf der Fachschaftswebseite kann nahezu in Echtzeit die Präsenz der Mitglieder von überall eingesehen werden.

Hardware

- Raspberry Pi 2
- WLAN Stick für Hotspotfunktionalität
- ENTTEC DMX USB Pro (DMX Interface)
- Eurolite PixelTube 16 (Leuchtsignal)
- Laustprecher / Kopfhörer
- LEDs zur Visualisierung des Systemstatuses

Software Design

Um möglichst effiziente Softwareentwicklungszyklen zu gewährleisten, wurde großer Wert auf eine maximal modulare Entwicklung der einzelnen Komponenten gelegt. Die folgende Grafik veranschaulicht den Architekturentwurf:

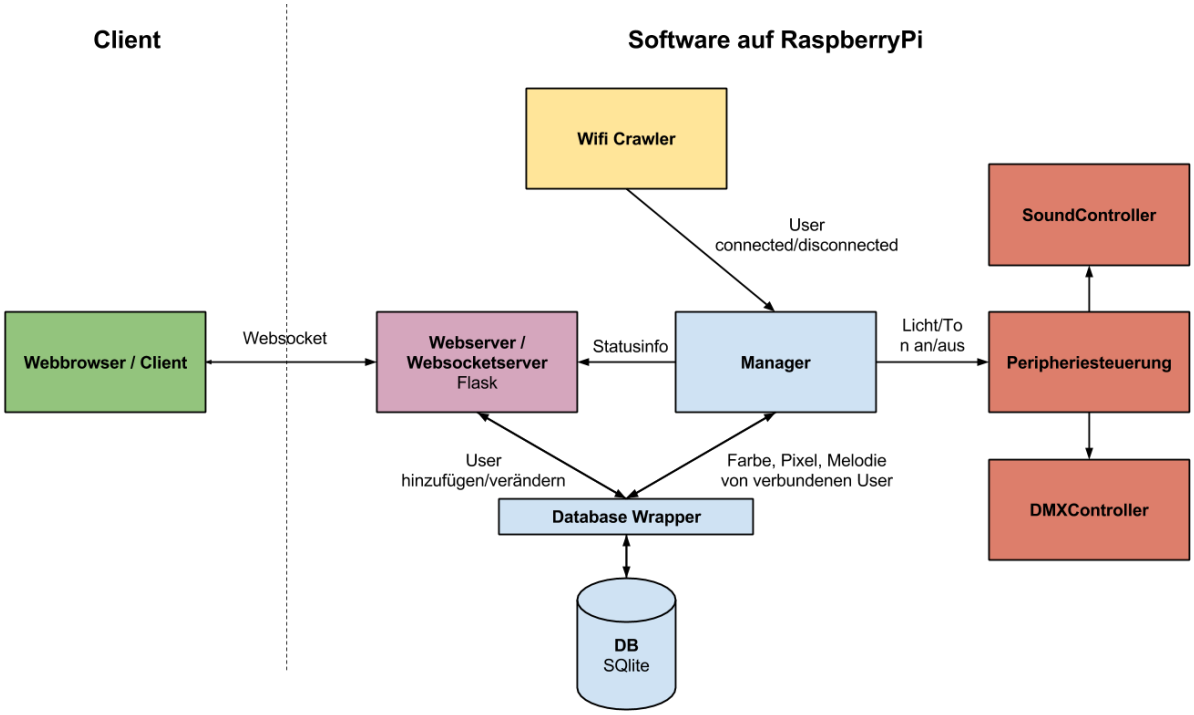

Abbildung 1: System Überblick

Der **Webserver**, der **Manager** und der **Wifi Crawler** kommunizieren mittels bi- und unidirektionaler **Queues**. Die **Peripherie-Steuerung wurde als Library** ausgelegt und wird direkt vom Manager verwendet. Zwischen Client und Webserver besteht eine bidirektionale Websocketverbindung, damit Präsenzänderungen in Fast-Echtzeit auf der Webseite der Fachschaft angezeigt werden können. Zusätzlich verwendet das **Verwaltungsinterface eine REST-Schnittstelle**, um User hinzufügen und abändern zu können. Zugriffe auf die **Datenbank wurden ebenfalls über eine Library gekapselt**.

Wifi Crawler

Hotspot

Die Hotspotfunktionalität wird mit dem Tool **hostapd** und einem zweiten WLAN Stick realisiert. Zusätzlich wurde ein DHCP Server auf dem RaspbberyPi installiert, der den Clients eigene IP Adressen gibt.

```
# Installation der benötigten Tools
$: apt-get install hostapd dhcpd

# Konfiguration von hostapd
$: cat /etc/hostapd/hostapd.conf
interface=wlan1
driver=rtl871xdrv
ssid=Horst
hw_mode=g
channel=6
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=goto_fail
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
```

Mit der oben angegebenen Konfiguration stellt das RaspberryPi einen Hotspot mit der SSID “Horst” und der WPA2 Passphrase “goto_fail” zur Verfügung.

Im Produktivsystem müsste noch eine **Netzwerkbrücke** zwischen dem Wifi- und dem Ethernetinterface erstellt werden, damit Geräte, die mit dem Raspberry Hotspot verbunden sind, auch weiterhin Internetzugriff haben.

Crawler

Um zu erkennen, welcher User sich verbunden hat, kommt das Tool **iwevent** zum Einsatz. Dieses Tool zeigt alle Ereignisse, die im Hotspot passieren auf der Kommandozeile an. Diese Ausgaben werden von einem Python Programm **geparsed** und an den Manager weiter gegeben. Beispielhafte Ausgabe von iwevent:

```
pi@192.168.188.26$: iwevent
Waiting for Wireless Events from interfaces...
07:33:44.994303   wlan1    Registered node:8C:3A:E3:17:DF:6C
07:33:48.866077   wlan1    Expired node:8C:3A:E3:17:DF:6C
```

Glücklicherweise kann iwevent auch als **unprevilegierter Benutzer** verwendet werden, somit ist es nicht nötig dieses Teil der Anwendung mit Root Rechten laufen zu lassen.

Statusanzeige GPIO

Zur Anzeige des Systemstatuses wurden zwei LEDs an das GPIO Interface des Raspberry Pis angeschlossen. Eine der beiden Leds zeigt den Start des Managers an. Sie erlischt beim Beenden des Managers. Die zweite LED visualisiert den Zustand des Wifi Crawlers. Zusätzlich blinkt die LED kurz, wenn ein User sich ins WLAN eingeloggt oder es verlassen hat.

Um zu vermeiden, dass die einzelnen Dienste mit Rootrechten laufen müssen, wurde zur Ansteuerung der GPIOs das **sysfs** Interface verwendet. Dank des Filesystem Mappings ist es möglich, über simple Dateisystemberechtigungen auch unprivilegierten Nutzern die Verwendung der GPIOs zu erlauben.

Logging

Um Überblick über das System zu bekommen und etwaiges Fehlverhalten leichter aufdecken zu können, werden alle Statusmeldungen der diversen Komponenten an einer zentralen Stelle gesammelt und in eine Logdatei geschrieben. Die Log-Nachrichten bestehen dabei aus Datum, Uhrzeit, Art der Nachricht, Dateiname der aufrufenden Systemkomponente, Codezeile, Nachricht. Nachfolgend ein exemplarischer Auszug des Logfiles:

```
2015-05-28 11:36:05,172 INFO: Drei.py (17) Starting Drei
2015-05-28 11:36:05,172 INFO: InitTables.py (24) Creating table Sounds...
2015-05-28 11:36:05,173 INFO: InitTables.py (30) Creating table Users...
2015-05-28 11:36:05,173 INFO: InitTables.py (40) Tables successfully created.
2015-05-28 11:36:05,265 INFO: Crawler.py (35) Running
2015-05-28 11:36:05,346 INFO: Manager.py (21) Running
2015-05-28 11:36:05,348 INFO: SoundController.py (29) Started
2015-05-28 11:36:05,349 INFO: Periphery.py (24) LightController is connected
2015-05-28 11:36:05,351 INFO: Webserver.py (31) Webserver: Running on http://127.0.0.1:8080
2015-05-28 11:36:07,268 INFO: Crawler.py (22) New peer connected: 00:80:41:ae:fd:7e
2015-05-28 11:36:08,273 INFO: Manager.py (52) user 00:80:41:ae:fd:7e added
2015-05-28 11:36:08,274 INFO: Periphery.py (39) light 0 turned on
2015-05-28 11:36:08,277 INFO: Periphery.py (58) sound at Knight-Rider-Theme-Song.mp3 played
2015-05-28 11:36:08,278 INFO: SoundController.py (32) Loading file Knight-Rider-Theme-Song.mp3
```

Manager

Der Manager ist in erster Linie für die Abarbeitung der vom Crawler durch die entsprechende Queue gemledeten Ereignisse, d.h. das An- bzw. Abmelden von Usern, zuständig. Hierbei werden bei jeder Änderung die aktuellen Benutzer aus der Datenbank ausgelesen. Anschließend wird, falls es sich um eine Anmeldung handelt, das entsprechende Licht in der vom Benutzer spezifizierten Farbe eingeschaltet und dessen ausgewählter Sound abgespielt. Meldet sich ein Benutzer ab, so wird das dementsprechende Licht aus gemacht. Hierzu werden die entsprechende Funktionen der Peripherie Bibliothek aufgerufen. Eine weitere Aufgabe des Managers ist die Abarbeitung der Ereignisse, welche aus dem Webserver durch die entsprechende Queue gesendet werden. Dies betrifft zum einen den Latenz-Test und zum anderen den Last-Test, welche direkt aus dem Browser aus gesteuert werden. Die dort gesetzten Werte werden direkt an die Peripherie weiter gegeben, um die Farbe des dafür vorgesehenen Lichts zu ändern.

Das Datenbank-Modul mit SQLite3

Im Bereich der Softwareentwicklung auf dem Raspberry Pi findet man häufig SQLite-Datenbanken eingesetzt. Dies hat vor allem Performance-Gründe, da eine SQLite-DB nur durch eine Datei repräsentiert wird, in die geschrieben wird - es wird kein MySQL-Server o.ä. benötigt, der zusätzliche Ressourcen verbrauchen würde.

In unserer Anwendung haben wir uns ebenfalls für die Verwendung einer SQLite-Datenbank entschieden - zum einen aufgrund der bereits erwähnten Performance-Vorteile, zum anderen, weil wir für unsere Datenhaltung mit gerade einmal zwei Tabellen und rudimentären DB-Operationen auskommen - der Betrieb eines kompletten MySQL-Servers hätte daher unnötiger Overhead dargestellt. Ein weiterer Vorteil von SQLite ist, dass eine Python-Library sqlite3 existiert, die über einen Import einfach eingebunden werden kann - es muss **keine zusätzliche Software** installiert werden. Die Zugriffsverwaltung der DB geschieht hierbei über simple Dateisystemberechtigungen des zugrunde liegenden Betriebssystems. Dabei haben die .db-Files standardmäßig die Rechtezuweisung rwxr–r–‘.

Genereller Aufbau

Zum komfortableren Zugriff auf die Datenbank haben wir das Wrapper-Modul `SQLiteWrapper.py` geschrieben, welches der Anwendung u.a. Funktionen zum Hinzufügen, Updaten und Löschen eines Users bereitstellt und somit die SQL-Statements wegkapselt. Dem vorgeschaltet ist ein Interface `Database.py, das die wichtigsten Funktionen definiert. Die Interface-Methoden sind  Ferner haben wir gerade für die anfängliche Projektphase die Klasse MockDatabse.py` geschrieben, um die Modularisierung zu erhöhen und die Entwicklung der DB-Schnittstelle unabhängig vom restlichen System vorantreiben zu können.

Datenbankentwurf

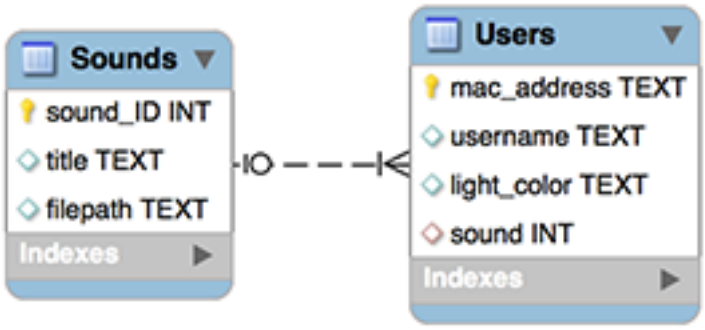

Abbildung 2: Entity-Releationship Modell

Das SQLiteWrapper.py Modul

Initialisierung des Moduls und der Datenbank

Beim Start unserer Anwendung wird im Runfile Drei.py die Datenbank mit ihren beiden Tabellen `Users` und `Sounds` angelegt:

```
# Initializing database with tables (if necessary)
InitTables.main()
```

Die Initialisierung der Datenbank geschieht dabei konkret über das Skript InitTables.py:

```
import sqlite3 as sql

def main():
    db_connect = None
    db = 'test.db'

    with sql.connect(db) as db_connect:
        # foreign keys are disabled by default for backwards compatibility
        db_connect.execute("""PRAGMA foreign_keys = ON;""")
        cursor = db_connect.cursor()
        cursor.execute("""PRAGMA foreign_keys;""")
        logger.log(Logger.INFO, "Creating table Sounds...")
        cursor.execute("""CREATE TABLE IF NOT EXISTS Sounds(
            sound_ID INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT,
            filepath TEXT UNIQUE
            );""")
        logger.log(Logger.INFO, "Creating table Users...")
        cursor.execute("""CREATE TABLE IF NOT EXISTS Users(
            mac_address TEXT PRIMARY KEY NOT NULL,
            username TEXT,
            light_color TEXT,
            sound INTEGER REFERENCES Sounds(sound_ID)
            );""")
        cursor.execute("""SELECT name FROM sqlite_master WHERE type='table' AND
            name='Sounds' OR name='Users';""")
        created_tables = flatten(cursor.fetchall())
        if len(created_tables) == 2:
            logger.log(Logger.INFO, "Tables successfully created.")
```

Der Wrapper selbst wird im Manager.py genutzt. Die Einbindung geschieht dabei über die Headerzeile `from lib.database.SQLiteWrapper import SQLiteWrapper`

Funktionen

Der Wrapper implementiert die folgenden Funktionen, die im Interface `Database.py` definiert sind:

```
- add_user(user)
- get_user(mac_address)
- retrieve_users()
- update_user(user_mac, user)
- delete_user(user_mac)
- list_sounds()
```

add_user(user)

Erhält als Parameter ein User-Objekt als DTO. Hierbei ist zu beachten, dass das User-Object zwar ein Attribut light_ID besitzt, dieses aber zunächst `None` ist. Die Light-ID wird erst beim Eintrag des Users in die Datenbank generiert und ist effektiv die ROWID des Tabelleneintrags.

get_user(mac_address)

Liefert zur gegebenen Mac-Adresse ein Userobjekt mit den Userdaten aus der Datenbank. Ist der User nicht vorhanden (kurzum - die Mac-Adresse kann in der DB nicht gefunden werden), wird `null` bzw `None` zurückgegeben.

retrieve_users()

Gibt eine Liste aller in der Datenbank eingetragenen User zurück. Ist die Datenbank leer (weil noch keine User eingetragen wurden), liefert retrieve_users() eine leere Liste []. Diese Funktion macht dabei intern nichts anderes, als eine Liste aller eingetragenen Mac-Adressen aus der DB abzufragen, für jede Adresse `get_user()`aufzurufen und die jeweiligen User-Objekte zu einer Liste zusammenzubauen.

update_user(user_mac, user)

Aktualisiert den User-Eintrag in der DB zu einer gegebenen Mac-Adresse. Es können nur die Lichtfarbe und der ausgewählte Sound bei einem bestehenden Nutzereintrag geändert werden, Name und Mac-Adresse sind unveränderlich. Existiert zu dem ausgewählten Sound noch kein Eintrag in der DB, wird ein neuer Eintrag in der Tabelle für die Sounds generiert. Die Funktion gibt `True`zurück, wenn ein Update ausgeführt wurde, und `False`, wenn nichts geändert wurde.

delete_user(user_mac)

Löscht den User-Eintrag in der DB zu einer gegebenen Mac-Adresse und liefert `True` / `False` zurück, um anzuzeigen, ob der Löschvorgang erfolgreich war oder nicht - existiert die angegebene Mac-Adresse nicht, wird auch nichts gelöscht und folglich `False` zurückgegeben.

list_sounds()

Gibt eine Liste aller in der DB eingetragenen Sounds zurück. Sind keine Sounds eingetragen, ist die gelieferte Liste leer.

Webserver: Flask

Der Webserver wurde mit Hilfe von Flask implementiert. Er ist für die Bereitstellung des Webinterfaces, der REST-Schnittstellen und Websockets verantwortlich. Der Webserver ist standardmäßig unter der Adresse **http://localhost:8080** verfügbar.

Zur Kommunikation mit dem Manager wurden zwei Queues eingerichtet. Eine Queue dient zur Nachrichtenübermittlung an den Manager und eine zum Nachrichtenempfang vom Manager. Erstere dient zur Umstellung der über den Websocket empfangenen Lichtfarbe. Diese wird vom Webserver über den Websocket empfangen und an den Manager weitergeleitet, der daraufhin das Licht in der entsprechenden Farbe anschaltet. Die Queue zum Nachrichtenempfang vom Manager wird genutzt, um Änderungen an den aktiven Benutzern zu empfangen. Als Datenquelle verwendet der Webserver direkt die SQLite Datenbank mit Hilfe des **SQLiteWrapper**s, welcher bereits zuvor vorgestellt wurde.

Die Bereitstellung der Dateien des Webinterfaces wird durch den in Flask intergrierten Webserver übernommen. Dieser stellt alle Dateien unter **static** zur Verfügung. Details zum Webinterface können dem nachfolgenden Punkt entnommen werden.

Die REST-Schnittstelle wurde mit Hilfe der Standardfunktionalität von Flask umgesetzt. Nachfolgende REST-Endpunkte existieren:

URL des Endpunkts	HTTP-Methode	Beschreibung
/api/users	GET	Gibt eine Liste aller Benutzer zurück
/api/Sounds	GET	Liefert eine Liste aller verfügbaren Sounds zurück
/api/users	POST	Legt den als Parameter übergebenen Benutzer an
/api/users/<string:user_id>	PUT	Aktualisiert den Benutzer mit der User-Id **<string:user_id>**
/api/users/<string:user_id>	DELETE	Löscht den Benutzer mit der User-Id **<string:user_id>**

Der zur Verfügung gestellte Websocket-Endpunkt wurde mit der Flask-Erweiterung **flask.ext.socketio** implementiert. Sobald ein Client die Verbindung zu diesem Websocket aufbaut, wird ein Event namens **Connected** ausgelöst. Der Client kann daraufhin folgende zwei Events an den Server senden.

Event	Beschreibung
GetActiveUsersEvent	Liefert eine Liste aller aktiven Benutzer zurück
LatencyColorEvent	Nimmt eine Farbe entgegen und setzt reicht diese an den Manager weiter, damit dieser ein Licht mit der entsprechenden Farbe anschaltet. Des weiteren sendet der Server bei jeder Änderung der Liste mit aktiven Usern ein Event namens **ActiveUsersNotification**, welches zusätzlich eine Liste der aktiven User enthält. Sobald sich also ein Benutzer in das WiFi einloggt oder das WiFi verlässt, wird dieses Event vom Server an alle Clients gebroadcasted.

Webinterface: AngularJS

Das Webinterface wird über den Webserver bereitgestellt und kann unter **http://localhost:8080/static/drei.html** abgerufen werden. Es handelt sich dabei um eine mit AngularJS entwickelte JavaScript-App, welche die zuvor vorgestellten REST- und Websocket-Endpunkte nutzt.

Die Entwicklung des Webinterfaces geschieht mit der von **yo-angular** bereitsgestellten Build-Umgebung in einem Verzeichnis außerhalb des Python-Entwicklungspfads. Nachdem ein Entwicklungspunkt erreicht wurde, welcher auf dem Webserver bereitgestellt werden soll, so muss zunächst das Verzeichnis **static** im Python-Entwicklungspfad geleert werden. Daraufhin wird im Pfad der Webapp mit **grunt build** der Build gestartet, welcher die Webapp wiederum in das **static** Verzeichnis baut. Daraufhin kann das aktualisierte Webinterface vom Server abgerufen werden.

Die Webapp enthält mehrere Services. Der **DataService** ist verantwortlich für die Kommunikation mit der REST-Schnittstelle des Servers. Dafür werden für die benötigten Endpunkte Methoden zur Verfügung gestellt. Dort auftretende Fehler werden durch den **ErrorHandler** behandelt, welcher Fehler in einem Fehlerdialog darstellt. Zusätzlich steht ein **WebsocketService** zur Verfügung, welcher für die Websocket-Kommunikation mit dem Server verantwortlich ist. Dieser bietet für die vom Server unterstützten Events entsprechende Methoden. Sobald Events empfangen werden, werden diese vom **WebsocketService** in die App gebroadcasted und an entsprechender Stelle in den Controllern behandelt.

Zusätzlich wurden mehrere Controller umgesetzt. Der **DashboardCtrl** ist für das Dashboard verantwortlich und lauscht auf den Broadcast des **WebsocketService**s, um die Liste der aktiven Nutzer aktuell zu halten. Bei der Initialisierung wird zudem ein **GetActiveUsersEvent** an den Server geschickt, damit nicht erst auf eine Änderung gewartet werden muss, um die aktiven Benutzer anzeigen zu können. Der **UsersCtrl** ist verantwortlich für die Ansicht zur Benutzerverwaltung. Er kommuniziert mit Hilfe des **DataServices** mit dem Webserver und empfängt, updated und löscht mit diesem System-User. Ein weiterer Controller ist der **LatencyCtrl**, dieser ist verantwortlich für die Testseite, auf der ein Latenz- und ein Last-Test durchgeführt werden können. Bei ersterem handelt es sich um einen Test der Zeit, die benötigt wird für die Kommunikation von Webserver bis hin zur Peripherie. Dabei kann man auf einem Schieberegler zwischen fünf verschiedenen Farben wählen, was die Farbe des dafür vorgesehenen Lichts ändert. Beim Last-Test werden automatisch 4048 Farbänderungen, in etwa wie beim Latenz-Test, gesendet, allerdings hier in kürzester Zeit. So wird das Verhalten des Systems und der Auswirkungen auf die Kommunikationsgeschwindigkeit unter Volllast getestet. Der entsprechende Controller kommuniziert mit dem **WebsocketService** um die entsprechenden Befehle an den Server zu übermitteln. Zusätzlich existieren noch ein **CreateCtrl** und ein **UpdateCtrl**, welche für das Anlegen bzw. Aktualisieren eines Users verantwortlich sind. Hierfür nutzen sie den **DataService**. Der **ErrorCtrl** dient zur Darstellung eines Fehlers.

Peripheriesteuerung

Wie bereits mehrfach erwähnt wurde die Ansteuerung der Peripherie als **Bibliothek** ausgelegt um den Zugriff auf die direkte Hardware zu abstrahieren. Zu den zur Verfügung gestellten Funktionen zählt die Ausgabe über die Audioschnittstelle und das Ansteuern der Pixeltube via DMX Bus. Zusätzlich werden einige **Hilfsfunktionen** mit dieser Bibliothek bereitgestellt, wie beispielweise das Umrechnen von Hex-Farb-Strings in RGB Werte.

Um auch innerhalb der Bibliothek klare Sturkturen einzuhalten wurde die Ansteuerung der Lampen in die Klasse DMXHandler und die Ansteuerung der Audioausgabe in die Klasse SoundController ausgelagert.

SoundController

Der **SoundController** nutzt die Bibiothek pygame zur Wiedergabe. Der Vorteil dieser Bibliothek ist das Abspielen von verschiedenen Musik-Formaten (bspw. .wav, .mp3, . . .).

DMXHandler

Die Kontrolle von DMX-Geräten erfolgt durch den **DMXHandler**. Er hält eine Liste von der Oberklasse **DMXDevice** abgeleiteter Klassen, sodass neue DMX Geräte schnell hinzugefügt werden können. Der **DMXHandler** verwaltet die Geräte und setzt mit Hilfe der **DMXConnection** in den entsprechenden Geräten die Farbwerte. **DMXConnection** sorgt für die Verbindung zum *ENTTEC DMX USB Pro* DMX Interface.

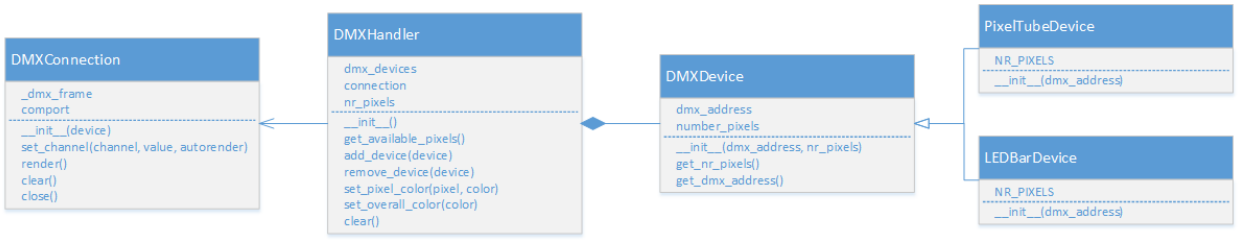

Abbildung 3: DMX Verwaltung

Farbverwaltung

Die Farbverwaltung für die DMX Geräte wird mit Hilfe der Klasse **RGBColor** und der Factory **ColorFactory** realisiert.

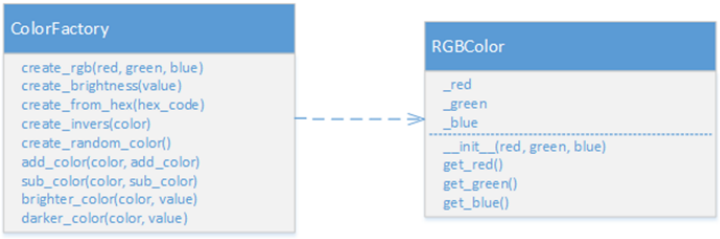

Abbildung 4: Farbverwaltung

Testing

Unittests mit PyUnit

Generelles Setup

1.) Einbinden der Unittest-Library:

```
import unittest
```

2.) Erstellen einer Testklasse durch Ableiten von unittest.TestCase:

```
class TestClass(unittest.TestCase):
```

3.) Falls gewünscht, kann eine testklassenweite setUp-Methode definiert werden, die zu Beginn der Ausführung der Testklasse *einmal* aufgerufen wird - wird über Annotation `@classmethod` realisiert:

```
@classmethod
def setUpClass(cls):
    // here be code
```

4.) Falls benötigt, kann eine tearDown-Methode spezifiziert werden, die nach jeder Test-Methode der Klasse ausgeführt wird:

```
def tearDown(self):
```

5.) Defintion der Testfälle

```
def test_simple_user_adding(self):
    self.assertTrue(self.db_wrapper.add_user(self.testuser))
```

Man kann auch andere assert-Methoden nutzen, wie beispielsweise den Klassiker `self.assertEqual(first, second, msg=None)`, der zwei Objekte `first` und `second` mit `==` vergleicht.

Datenbank-Wrapper

Innerhalb unseres Projekts wurden vor allem für den Datenbank-Wrapper SQLiteWrapper.py Unittests zur Absicherung und Dokumentation der einzelnen Interface-Methoden geschrieben. Zu Beginn werden diverse Testuser angelegt und als Objektvariablen definiert, um sie in den Tests später komfortabel nutzen zu können. In der tearDown() wird sicher gestellt, dass die DB nach der Ausführung einer Testmethode leer ist.

Systemtest

Webinterface

Die Funktion des Webinterfaces wurde durch ausführliche Oberflächentests verifiziert. Diese wurden manuell von unserem Team durchgeführt.

Testcase	i.o.	Bemerkung
Benutzer hinzufügen	✓	
Benutzer ändern	✓	Farbe, Musik wurden geaendert
Benutzer entfernen	✓	
Echtzeit Onlineanzeige	✓	Minimal merkbare Verzögerung bis Status aktualisiert

Peripherie

Die Peripherieansteuerung, bestehend aus Soundausgabe, LED-Bar und Pixeltube, wurde im gleichen Zuge getestet. Der einfache Positivtest lief erwartungsgemäß gut, das Mobilgerät eines Nutzers wurde vom System sofort bei Einschalten des WLANS am Mobilgerät erkannt, bevor das Smartphone selbst das WLAN des Pi erkannt hat. Die Verzögerung zwischen Systemreaktion und WLAN-Erkennung war nicht wahrnehmbar.

Beim gleichzeitigen Anmelden zweier User werden beide Lichtslots geschaltet, der Sound des ersten Users wird dann abgebrochen und der Sound des zweiten Users abgespielt.

Crawler

Um die Funtion des Crawlers zu Testen wurde mit einem Testtelefon eine Verbindung mit dem Hotspot des RaspberryPIs hergestellt. Danach wurde ausgewertet ob der Crawler das neue Telefon erkennt und ob die MAC Adresse korrekt ermittelt wird. Beides war mit 2 verschiedenen Testgeräten der Fall. Zusätzlich wurde auch getest ob beim Trennen der Verbindung alles richtig erkannt wird. Auch das hat anstandslos funktioniert. Der Crawler kann somit als funktionierend betrachtet werden. Diese Tests wurden auch manuell von einem Teammitglied durchgeführt.

Lasttest

Um zu simulieren wie sich das System unter Last verhält wurd eine **große Anzahl paraleller Websocket Anfragen** an den Webserver ausgelöst. Diese sollten einfach nur die Farbe eines Pixels der Pixeltube ändern. Nachdem alle Anfragen vom Webinterface gesendet wurden, **dauert es ein wenig** bis diese auch tatsälich auf der Pixeltube angezeigt werden. **Allerdings geht keine einzige Anfrage verlohren**. Da wir zwischen den einzelnen Komponenten Queues zur Kommunikation verwenden, welche einen **internen Puffer** haben und somit keine Anfragen wegschmeißen, ist es egal ob eine Komponente nicht mit der Verarbeitung hinter her kommt. Allerdings werden Statusänderungen von Fachschafftsmitglieder auch erst ausgeführt wenn alle Anfragen des Lasttests verarbeitet sind **(First-In-First-Out Prinzip)**.

Anhänge

Arbeitsaufteilung

Nachfolgend finden Sie eine Auflistung der Modulverantwortlichen, sowie den Bearbeitungstatus der einzelnen Module:

Modul	Verantwortlicher	Erledigt
Manager	Markus Hornung	✓
Datenbankanbindung	Stephanie Ehrenberg	✓
Webinterfaces / Webserver	Simon Jahreiss	✓
DMX Interface / Soundansteuerung	Luis Morales	✓
Hotspot / Crawler + GPIO	Maximilian Pachl	✓

Sprint Protokolle

19.03.2015

- Architekturkonzept
- Verteilung Aufgaben
- Recherche/Einrichtung Aufgabenverwaltung (Kanbanery)

26.03.2015

- Daily Scrum:
  - Simon: Socketrecherche abgeschlossen, Webserver liefert statische Ressourcen aus, App läuft clientseitig und verbindet sich mit Sockets
  - Max: Hotspot funktioniert
  - Stephanie: SQLite eignet sich optimal zum Einsatz im Projekt
  - Luis: Erste Recherche zu Sound/Lampen abgeschlossen
  - Markus: Möglichkeiten zu IPC recherchiert, Sockets und Pipes kommen in Frage -> Named Pipes ideal
- Verteilung neuer Aufgaben:
  - Kein Einsatz von Maven-ähnlichem Programm zur Verwaltung von Dependencies
  - DB Anforderungen geklärt

09.04.2015

- Daily Scrum:
  - Max: NIX
  - Luis: Lampen läuft, Sound läuft
  - Simon: bisschen REST angeschaut, IPC helfen
  - Stephanie: SQLite ausgecheckt
  - Markus: IPC
- TODO:
  - Max: Bau den Crawlers
  - Luis: Lampe testen, pybuilder anschaun

  - Simon: REST weiter machen
  - Stephanie: DB weitermachen
  - Markus: IPC (JSON parser)

- Planung Sprints:

  1. Bis 23.04.15 08:15: MockUp Crawler, Manager, DB, rudimentäres Webinterface (Liste von User anzeigen, neuer User hinzufügen)
  2. Crawler, Webinterface (Onlinestatus)
  3. Webinterface (User löschen, ändern)

16.04.2015

- Daily Scrum

  - Max: Access Point ausgecheckt, Crawler gebaut
  - Stephanie: User in DB einfügen, SVN kommt bald
  - Luis: Farben etc in Klasse, Ansteuerung inkl. hexadezimal etc., Anbindung Mockup Manager kommt
  - Simon: Webinterface weiter gemacht, Dependencies
  - Markus: Manager fertig, Dependencies

- TODO:

  - Luis+Markus: Controller Steuerung Peripherie (Manager)
  - Steph: DB fertig, ins SVN
  - Max: Hybrid-Crawler (Mockup+Normal) + Kommunikation Manager
  - Simon: Webinterface weiter machen
  - Simon+Markus: Dependencies aufm Raspi auschecken
  - Markus: Manager umbauen

23.04.2015

Erster Sprint erfolgreich abgeschlossen, alle geplanten Arbeiten abgeschlossen, DB noch nicht 100%ig fertig, Webinterface quasi schon fertig

- TODO:

  - Max: LEDs für Komponenten einbauen (status) -> Library
  - Steph: DB: Skript einbinden -> Tabellen anlegen, Rechtemanagement
  - Simon+Markus: Raspi pybuilder
  - Luis: Objektorientierung in periphery
  - Simon: Manager in webserver anbinden
  - Max: möglichkeiten webserver -> lokal starten?! (Flask)

- Planung Sprints:

  2. Bis 07.05.15 8:15: TODOs
  3. ???

07.05.2015

1. Systemtest: läuft bei uns Erfolgreiches Beenden des 2. Sprints
2. Sprint: Fertig stellen (Wireless Accesspoint, Sound Wiedergabe, Anzeige Sounds Webservice, Rechte abchecken) bis: 21.05.15

28.05.2015

- TODO:
  - Logging/error handler
  - Tests (wo sinnvoll/möglich)
  - Kommentare
- Erledigt:
  - Logging (Markus)
  - Kommentare in Webservices (Simon)
  - Webservices refactored (Simon)
  - Light IDs angepasst (Max)
  - MAC-Adressen case-insensitive (Markus)

11.06.2015

Erledigt:

- Sound Suche in eigenes Modul abgekapselt (Simon)
- Status Licht hinzugefügt (Luis)
- Manager-Webserver Queue eingeführt für dynamische Steuerung aus Browser (-> Status Licht) (Markus)
- Logging für DMX hinzugefügt (Luis)
- Kommentare hinzugefügt (Simon, Markus)
- Fehlerdialog in web-ui hinzugefügt (Simon)
- Bug in SQLiteWrapper behoben (Simon)
- Http Fehler codes hinzugefügt (Simon)
- Url korrigiert (Simon)
- web-ui aktualisiert (Simon)

25.06.2015

- Max: Doku erweitert
- Markus: Weitere sounds hinzugefügt
- Simon: Lasttest hinzugefügt