

Todo App

Entwicklerdokumentation



17. März 2024

Berufscschule

Moltkestraße 24

# Inhaltsverzeichnis

[Inhaltsverzeichnis 1](#_Toc1021934154)

[Projektzweck 2](#_Toc249248615)

[Architekturübersicht 3](#_Toc868920656)

[Benutzeroberfläche 3](#_Toc302738186)

[Datenverwaltung 3](#_Toc1109261400)

[Netzwerkkommunikation 3](#_Toc832650319)

[Abhängigkeiten 4](#_Toc1762872877)

[Schlüsselkomponenten 4](#_Toc189231216)

[ToDoListItem 5](#_Toc1738825040)

[Methoden 5](#_Toc3654847)

[Eigenschaften 6](#_Toc1017923188)

[MainScreen & SettingScreen Screens 6](#_Toc1118740724)

[TaskStorageHandler 6](#_Toc610917042)

[Methoden 6](#_Toc1649953131)

[Attribute 7](#_Toc523176989)

[Externe Abhängigkeiten 7](#_Toc1214180271)

[MqttHandler 8](#_Toc874674335)

[Methoden 8](#_Toc1354239118)

[Attribute 9](#_Toc98866987)

[Externe Abhängigkeiten 9](#_Toc1179096565)

[MqttConfig 9](#_Toc2053909786)

[Methoden 10](#_Toc1754475375)

[Attribute 10](#_Toc308152625)

[Externe Abhängigkeiten 10](#_Toc346775046)

# Projektzweck

Erschaffen einer Todo-App, die von mehreren Nutzern zur selben Zeit verwendet werden kann. Diese App kann von Familien für Ihren Einkauf genutzt werden oder für Aufgaben, die erledigt werden müssen. Damit kann jeder, der mit der gleichen Schnittstelle verbunden ist, die gemeinsame Liste bearbeiten.

# Architekturübersicht

Die Anwendung ist mit der Kivy-Bibliothek für die grafische Benutzeroberfläche und der paho-mqtt-Bibliothek für die Netzwerkkommunikation entwickelt worden.Die Architektur des Systems kann in drei Hauptkomponenten unterteilt werden: die Benutzeroberfläche, die Datenverwaltung und die Netzwerkkommunikation.

## Benutzeroberfläche

Die Benutzeroberfläche ist mit Kivy und KivyMD erstellt.Sie besteht aus mehreren Bildschirmen und Widgets, die eine einfache Navigation und Interaktion ermöglichen. Dazu gehören die “MainToDoList” und “LoginScreen” Bildschirme, welche die Hauptinteraktionspunkte für die Benutzer darstellen. Die Oberfläche ermöglicht es Benutzern, Aufgaben hinzuzufügen, zu bearbeiten, zu löschen und deren Status zu aktualisieren.

## Datenverwaltung

Die Verwaltung der To-Do-Listen erfolgt durch die “TaskStorageHandler-Klasse”, die für das Lesen und Schreiben von Aufgaben in einer lokalen JSON-Datei verantwortlich ist. Diese Komponente sorgt für die Persistenz der Benutzerdaten und ermöglicht eine einfache Synchronisation mit dem MQTT-Server. Die Datenstruktur umfasst die UUID der Aufgabe, den Zustand und die Beschreibung.

## Netzwerkkommunikation

Die Netzwerkkommunikation wird durch die “MqttHandler-Klasse” realisiert, die eine Verbindung zu einem MQTT-Broker herstellt. Diese Komponente ist verantwortlich für das Publizieren von Änderungen an den To-Do-Listen und das Abonnieren von Updates, wodurch eine Echtzeit-Synchronisation der Daten zwischen verschiedenen Instanzen der Anwendung ermöglicht wird. Die MQTT-Konfiguration, einschließlich Broker-Adresse und Zugangsdaten, wird über die “MqttConfig-Klasse” verwaltet.

## Abhängigkeiten

Um das Projekt in einer Entwicklungsumgebung zum Laufen zu bringen, müssen folgende Bibliotheken mit pip installiert werden:  
  
KivyMD (pip install kivymd)  
paho-mqtt 1.6.1 (pip install paho-mqtt<2.0.0)

yaml (pip install yaml)

Konfiguration

Damit zur Laufzeit eine Verbindung zum Mqtt-Broker hergestellt werden kann müssen folgende Parameter entweder im Vorraus oder innerhalb der Anwendung auf der Seite der Einstellungen eingetragen werden:  
  
Broker-Adresse: Domäne des Mqtt-Broker.

Port: entsprechender Port zur Adresse.

Username: Name des Klienten

Password: Passwort des Klienten

# Schlüsselkomponenten

## ToDoListItem

Beschreibung: Ein benutzerdefiniertes Widget, das eine einzelne Aufgabe in der To-Do-Liste darstellt. Es enthält eine Checkbox, um den Status der Aufgabe (erledigt/nicht erledigt) zu markieren, sowie einen Textbereich für die Aufgabenbeschreibung.

Interaktion: Benutzer können den Status einer Aufgabe durch das Aktivieren der Checkbox ändern. Durch langes Drücken auf eine Aufgabe werden Bearbeiten- und Löschen-Buttons angezeigt, die weitere Interaktionen ermöglichen.

Es scheint, dass Sie eine Entwicklerdokumentation für eine Klasse namens `ToDoListItem` schreiben müssen, die Teil einer Anwendung ist, die mit Kivy entwickelt wurde. Die `ToDoListItem`-Klasse stellt ein Listenelement dar, das Checkboxen und Schaltflächen zum Bearbeiten und Löschen von Aufgaben enthält. Hier ist eine Beispielstruktur für die Entwicklerdokumentation:

### Methoden

`\_\_init\_\_(self, \*\*kwargs)`

Konstruktor für das `ToDoListItem`-Objekt. Initialisiert die Eigenschaften und Event-Handler.

`refresh\_view\_attrs(self, rv, index, data)`

Aktualisiert die Ansichtsattribute des Widgets basierend auf den bereitgestellten Daten.

`on\_touch\_down(self, touch)`

Handler für Touch-Events, um Schaltflächen nach einer Verzögerung anzuzeigen.

`on\_touch\_up(self, touch)`

Handler für das Loslassen des Touch-Events, um geplante Ereignisse zu stornieren.

`show\_buttons(self, \*args)`

Zeigt die Bearbeitungs- und Löschschaltflächen an.

`hide\_buttons(self)`

Versteckt die Bearbeitungs- und Löschschaltflächen.

`hide\_buttons\_other\_items(self)`

Versteckt die Schaltflächen anderer Listenelemente.

`on\_checkbox\_change(self, checkbox, value)`

Handler für die Änderung des Checkbox-Zustands.

`on\_data(self, \*args)`

Handler für Datenänderungen, um den Zustand der Checkbox zu aktualisieren.

### Eigenschaften

`checkbox`

Objekteigenschaft für die Checkbox im Listenelement.

`id`

Zeichenfolgeneigenschaft, die die eindeutige ID des Listenelements enthält.

`text`

Zeichenfolgeneigenschaft, die den Text der Aufgabe enthält.

`state`

Boolesche Eigenschaft, die den Zustand der Aufgabe (erledigt/nicht erledigt) angibt.

## MainScreen

Die `MainScreen`-Klasse ist eine Bildschirmklasse, die die Hauptoberfläche der Anwendung darstellt. Sie ermöglicht das Laden, Hinzufügen, Synchronisieren, Bearbeiten und Löschen von Aufgaben.

### Methoden

`\_\_init\_\_(self, mqtt\_client, \*\*kw)`

Konstruktor für das `MainScreen`-Objekt. Initialisiert den MQTT-Client für die Kommunikation und ruft den Konstruktor der übergeordneten Klasse auf.

`load\_tasks\_in\_local\_list(self, tasks)`

Lädt Aufgaben in die lokale Liste des Bildschirms. Konvertiert Aufgaben in das erforderliche Datenformat und aktualisiert die Ansicht.

`add\_item(self, uuid="")`

Fügt ein neues Aufgabenelement hinzu. Generiert eine UUID, falls keine angegeben ist, und aktualisiert die Datenansicht.

`sync\_items(self)`

Synchronisiert Aufgaben mit den behaltenen Nachrichten des MQTT-Clients. Aktualisiert die lokale Aufgabenliste basierend auf den empfangenen Daten.

`delete\_item(self, task\_uuid)`

Löscht ein Aufgabenelement basierend auf seiner UUID. Aktualisiert die lokale Aufgabenliste und veröffentlicht die Änderungen über MQTT.

`edit\_item(self, item\_widget)`

Öffnet den Bearbeitungsmodus für ein Aufgabenelement. Zeigt ein globales Texteingabefeld an, um den Text der Aufgabe zu bearbeiten.

`apply\_global\_edit(self)`

Übernimmt die globalen Bearbeitungen für ein Aufgabenelement. Aktualisiert den Text der Aufgabe, veröffentlicht die Änderungen über MQTT und beendet den Bearbeitungsmodus.

### Attribute

`mqtt\_client`

Der MQTT-Client für die Kommunikation.

### Externe Abhängigkeiten

- `time`: Zur Erzeugung von Zeitverzögerungen.

- `json`: Zur Verarbeitung von JSON-Daten.

- `kivy.uix.screenmanager.Screen`: Kivy-Bildschirmklasse für die Oberfläche.

- `logic.storage.TaskStorageHandler`: Klasse zur Handhabung der Aufgabenspeicherung.

- `uuid`: Zur Generierung eindeutiger Identifikatoren für Aufgaben.

## SettingScreen

SettingScreen: Ein Bildschirm zur Eingabe der Konfigurationsdaten für die MQTT-Verbindung. Benutzer können hier Server-Details und Authentifizierungsdaten eingeben, um die Synchronisation über MQTT zu ermöglichen.

### Methoden

`\_\_init\_\_(self, mqtt\_client, \*\*kw)`

Konstruktor für das `SettingScreen`-Objekt. Initialisiert die MQTT-Client-Instanz.

`is\_valid(self, username, password, broker\_adress, port, topic)`

Überprüft die Gültigkeit der eingegebenen MQTT-Konfigurationseinstellungen.

`update\_config\_yaml(self, username\_input, password\_input, broker\_adress\_input, port\_input, topic\_input)`

Aktualisiert die MQTT-Konfigurationsdatei mit den neuen Einstellungen.

`show\_shutdown\_dialog(self)`

Zeigt einen Dialog an, der den Benutzer über den Neustart der App informiert.

`shutdown\_app(self, \*args)`

Beendet die App nach Bestätigung durch den Benutzer.

### Externe Abhängigkeiten

- `kivy.uix.screenmanager.Screen`: Kivy-Klasse für die Verwaltung von Bildschirmen.

- `logic.mqtt.MqttConfig`: Klasse zur Verwaltung der MQTT-Konfiguration.

- `kivymd.uix.dialog.MDDialog`: KivyMD-Klasse für Dialoge.

- `kivymd.uix.button.MDFlatButton`: KivyMD-Klasse für flache Schaltflächen.

- `re`: Modul für reguläre Ausdrücke.

- `sys`: Modul für System-spezifische Funktionalitäten.

## TaskStorageHandler

Die `TaskStorageHandler`-Klasse ist verantwortlich für das Hinzufügen, Lesen, Aktualisieren und Löschen von Aufgaben in einer persistenten Datenspeicherung sowie für die Veröffentlichung von Änderungen über MQTT.

### Methoden

`\_add\_task(client\_id, message)`

Statische Methode zum Hinzufügen einer neuen Aufgabe zur Datenspeicherung. Erhält die Client-ID und die Aufgabenmeldung als Parameter.

`\_set\_task\_state(uuid, state: bool)`

Statische Methode zum Aktualisieren des Status einer Aufgabe. Erhält die UUID der Aufgabe und den neuen Status als Parameter.

`\_read\_data()`

Statische Methode zum Lesen aller Aufgabendaten aus der Datenspeicherung.

`\_write\_data(data)`

Statische Methode zum Schreiben von Daten in die Datenspeicherung.

`\_delete\_task(task\_uuid)`

Statische Methode zum Löschen einer Aufgabe aus der Datenspeicherung. Erhält die UUID der zu löschenden Aufgabe als Parameter.

`\_set\_task\_text(task\_uuid, text)`

Statische Methode zum Aktualisieren des Texts einer Aufgabe. Erhält die UUID der Aufgabe und den neuen Text als Parameter.

### Attribute

`tasks\_data\_path`

Pfad zur Datei, in der Aufgabendaten gespeichert sind.

`mqtt\_config`

Konfigurationsobjekt für MQTT-Kommunikation.

`mqtt\_client`

Clientobjekt für MQTT-Kommunikation.

### Externe Abhängigkeiten

- `json`: Zur Verarbeitung von JSON-Daten.

- `uuid`: Zur Generierung eindeutiger Identifikatoren für Aufgaben.

- `logic.mqtt.MqttConfig`: Konfigurationsklasse für MQTT.

- `logic.mqtt.MqttHandler`: Klasse zum Handhaben von MQTT-Kommunikation.

## MqttHandler

Funktionalität: Kümmert sich um die Kommunikation mit dem MQTT-Broker. Diese Klasse ermöglicht das Publizieren von Änderungen an der To-Do-Liste und das Abonnieren von Updates, um eine Synchronisation zwischen verschiedenen Instanzen der App zu ermöglichen.

Konfiguration: Die Konfigurationsdetails für die MQTT-Verbindung werden durch die “MqttConfig”-Klasse bereitgestellt und beinhalten Informationen wie Broker-Adresse, Port und Authentifizierungsdaten

### Methoden

`\_\_init\_\_(self, config: MqttConfig)`

Konstruktor für das `MqttHandler`-Objekt. Initialisiert die Konfiguration und den MQTT-Client.

`on\_connect(self, client, userdata, flags, rc, properties=None)`

Handler für die Verbindung zum MQTT-Broker.

`on\_publish(self, client, userdata, mid, properties=None)`

Handler für das erfolgreiche Veröffentlichen einer Nachricht.

`on\_subscribe(self, client, userdata, mid, granted\_qos, properties=None)`

Handler für das erfolgreiche Abonnieren eines Topics.

`on\_message(self, client, userdata, msg)`

Handler für eingehende Nachrichten vom MQTT-Broker.

`publish\_message(self, message, retain=False, qos=1)`

Veröffentlicht eine Nachricht an das angegebene Topic.

`start\_loop(self)`

Startet die MQTT-Schleife, um eingehende Nachrichten zu empfangen.

`stop\_loop(self)`

Stoppt die MQTT-Schleife.

`get\_retained\_messages(self)`

Ruft die zuletzt gespeicherten behaltenen Nachrichten ab.

`\_\_delete\_old\_retained\_messages(self)`

Löscht die alten behaltenen Nachrichten.

### Attribute

`config`

Konfigurationsobjekt für die MQTT-Kommunikation.

`topic`

Das MQTT-Topic, dem der Handler folgt.

`client`

Der Paho MQTT-Client, der für die Kommunikation verwendet wird.

`\_\_retainedMessages`

Eine Liste zum Speichern von behaltenen Nachrichten.

### Externe Abhängigkeiten

- `time`: Zur Erzeugung von Zeitverzögerungen.

- `paho.mqtt.client`: Bibliothek zur Implementierung von MQTT-Kommunikation.

- `logic.mqtt.MqttConfig`: Konfigurationsklasse für MQTT.

## MqttConfig

Funktionalität: Verwaltet die Konfigurationsdaten für die MQTT-Verbindung. Diese Klasse lädt die Konfiguration aus einer JSON-Datei und stellt diese Informationen anderen Teilen der Anwendung zur Verfügung.

Anpassung: Benutzer können die Konfigurationsdatei bearbeiten, um die Verbindungsdetails für ihren spezifischen MQTT-Broker anzupassen.

### Methoden

`\_\_init\_\_(self, client\_id, broker\_adress, port, topic, qos, username, password)`

Konstruktor für das `MqttConfig`-Objekt. Initialisiert die MQTT-Konfigurationsdaten.

`load\_from\_resource()`

Statische Methode zum Laden der MQTT-Konfiguration aus der Konfigurationsdatei. Wenn kein `client-id` in der Konfiguration angegeben ist, wird eine zufällige `client-id` generiert und die Konfiguration aktualisiert.

`read()`

Statische Methode zum Lesen der Konfigurationsdaten aus der Konfigurationsdatei.

`write(data)`

Statische Methode zum Schreiben von Konfigurationsdaten in die Konfigurationsdatei.

### Attribute

`client\_id`

Die Client-ID für die MQTT-Kommunikation.

`broker\_adress`

Die Adresse des MQTT-Brokers.

`port`

Der Port des MQTT-Brokers.

`topic`

Das MQTT-Topic, dem der Client folgt.

`qos`

Die Quality of Service (QoS) für die MQTT-Kommunikation.

`username`

Der Benutzername für die Authentifizierung beim MQTT-Broker.

`password`

Das Passwort für die Authentifizierung beim MQTT-Broker.

### Externe Abhängigkeiten

- `json`: Zur Verarbeitung von JSON-Daten.

- `sys`: Zur Beendigung des Programms bei Fehlern.

- `uuid`: Zur Generierung eindeutiger Identifikatoren für Clients.

