

Alles klar — ich schreibe dir jetzt eine **Schritt-für-Schritt ToDo-Liste**, so dass du **offline** immer zu 100 % weißt, was als Nächstes kommt und in welcher Datei du es umsetzen musst.
Ich gehe streng nach der Verantwortungskette vor, damit du nicht durcheinanderkommst.

◆ Schritt-für-Schritt ToDo – Account-System

1 models.py – Datengerüste bauen

Ziel: Nur die Strukturen der Daten definieren. Keine Logik, kein Laden, kein Speichern.

ToDo:

1. Import:

```
from dataclasses import dataclass
```

2. Drei dataclass-Definitionen erstellen:

- **AccountMeta** → id, name, path
- **AccountData** → Alle inhaltlichen Accountdaten (SMTP, Schlüssel, Filter, etc.)
- **AccountFiles** → Pfade innerhalb eines Accounts (Pfad zu accountData.json, Pfad zu Logs, etc.)

3. **Achtung:** Hier niemals open() oder json.load() verwenden.
-

2 exceptions.py – Eigene Fehlerarten

Ziel: Eindeutige Fehlermeldungen, damit du beim Debuggen sofort weißt, was schief lief.

ToDo:

1. Jede Exception ist eine eigene Klasse, z. B.:

```
class AccountNotFoundError(Exception):  
    pass
```

2. Benötigte Exceptions:

- Account nicht gefunden
- Account existiert bereits
- Account-Limit erreicht
- Ungültige Account-Daten

3. Keine Logik in diesen Klassen – nur Namen und evtl. Custom-Nachricht.

3 paths_manager.py – Pfadverwaltung

Ziel: Einziger Ort, der accountPaths.json liest/schreibt.

ToDo:

1. Methoden erstellen:

- `load_paths()` → gibt alle Accountpfade zurück
- `add_path(account_meta: AccountMeta)` → neuen Account eintragen
- `remove_path(account_id)` → Account austragen
- `get_path(account_id)` → Pfad eines Accounts holen
- `list_paths()` → alle Accounts auflisten

2. **Achtung:**

- Hier **keine** AccountData laden!
 - Nur accountPaths.json bearbeiten.
-

4 repository.py – Daten für einen Account

Ziel: Arbeitet mit **einem einzelnen Account** und verwandelt JSON <-> Dataclasses.

ToDo (Laden):

1. Pfad vom paths_manager.py holen (`get_path`)
2. accountData.json öffnen (`open(path) + json.load()`)
3. Dictionary-Daten in AccountData umwandeln:

```
from dataclasses import asdict
account_data = AccountData(**data_dict)
```

4. Falls nötig: AccountMeta + AccountFiles ebenfalls instanziieren
5. Ergebnis an manager.py zurückgeben

ToDo (Speichern):

1. AccountData-Instanz von manager.py bekommen
 2. Mit `asdict()` in dict umwandeln
 3. Mit `json.dump()` in Datei schreiben
-

5 interface.py – Schnittstelle festlegen

Ziel: manager . py arbeitet nur gegen ein festes Interface, nicht direkt gegen die Implementierung.

ToDo:

1. from typing import Protocol importieren

2. Klasse definieren:

```
class AccountRepository(Protocol):
    def get_account(self, account_id: int) -> AccountData: ...
    def list_accounts(self) -> list[AccountMeta]: ...
    def create_account(self, data: AccountData) -> None: ...
    def update_account(self, account_id: int, data: AccountData) -> None: ...
```

3. Keine Implementierungen in dieser Datei – nur Methodensignaturen.

6 manager.py – Zentrale Steuerung

Ziel: Entgegennahme der Befehle von main . py und Koordination der restlichen Module.

ToDo:

1. Methoden definieren:

- create_account() → fragt Daten vom Benutzer ab, ruft repository.create_account() auf, trägt Pfad in paths_manager ein
- load_account(account_id) → ruft repository.get_account() auf
- update_account() → ruft repository.update_account() auf
- remove_account() → entfernt Account aus Pfaden und löscht Ordner

2. Fehlerbehandlung mit exceptions . py

3. Keine JSON-Zugriffe direkt hier – immer über repository . py oder paths_manager . py.

7 main.py – Einstiegspunkt

Ziel: Verbindung zwischen Benutzer (CLI/GUI) und manager . py.

ToDo:

1. Argumente per argparse einlesen

2. Entsprechende Methoden im manager . py aufrufen

3. Keine Business-Logik, kein JSON-Handling.

Reihenfolge zum Implementieren

1. **models.py** – Datengerüste fertigstellen
 2. **exceptions.py** – Fehlerklassen anlegen
 3. **paths_manager.py** – Pfadverwaltung umsetzen
 4. **repository.py** – Laden/Speichern eines Accounts implementieren
 5. **interface.py** – Schnittstelle definieren
 6. **manager.py** – Zentrale Steuerung umsetzen
 7. **main.py** – CLI-Anbindung bauen
-

 Damit hättest du eine **klare Offline-Roadmap**, bei der jede Datei nur genau ihre Aufgabe erfüllt und keine Logik vermischt wird.

Wenn du willst, kann ich dir diese Liste noch **als nummerierte Mindmap mit Pfeilen** umsetzen, damit du auch visuell Schritt für Schritt folgen kannst.

Möchtest du, dass ich das in einer Mindmap darstelle?

So hättest du eine visuelle Offline-Version deines ToDo-Plans.