Perfekt – danke für deine Rückmeldung. Du bekommst jetzt die **überarbeitete To-Do-Liste**, **komplett offline umsetzbar**, **anfängerfreundlich erklärt**, mit **noch klarerer Rollenverteilung** der Dateien – speziell so, dass du **jedes Teil des Account-Systems sofort verstehst**, ohne irgendetwas googeln zu müssen.

✅ ZIEL DER ARBEIT

Du baust ein **Account-Verwaltungssystem**, bei dem:

- Jeder Account ein eigener **Ordner mit JSON-Dateien** ist.
- Es eine **zentrale accountsPaths.json** gibt, die **alle Accounts verwaltet** (wie ein Inhaltsverzeichnis).
- Du Accounts erstellen, löschen, laden, aktualisieren und auswählen kannst.
- Die Struktur **sauber getrennt** ist – jede Datei hat einen **klaren Zweck**.

📁 DEINE DATEIEN & IHRE AUFGABEN (jetzt ganz klar!)

`main.py`

Was sie tut:
→ Startpunkt. Fragt den Benutzer (z. B. per `argparse`) was passieren soll: Account erstellen, laden, etc.
→ Leitet das Kommando an `manager.py` weiter.

Stell dir vor:
Der Benutzer spricht mit `main.py`. `main.py` spricht **nicht** selbst mit Accounts, sondern **schickt Befehle** an den Manager.

`manager.py` (**Zentrale Steuerung = "Chef")

Was sie tut:
→ Diese Datei bekommt Befehle von `main.py`
→ Ruft dafür Funktionen auf in:

- `repository.py` (für **einen** Account)

- `paths_manager.py` (für **alle Accounts**)
  → Ruft ggf. Logger & Fehlerklassen auf
  → Kombiniert alles zu einem Ergebnis

Stell dir vor:
`manager.py` ist der **Chef**, der die **anderen Dateien koordiniert**, aber selbst **nichts direkt** tut. Er sagt nur, *"Lade diesen Account, prüfe das Passwort, aktualisiere den Account."*

`repository.py` (**Einzelaccount-Verwalter = "Buchhalter")

Was sie tut:
→ Arbeitet **mit einem einzelnen Account-Ordner**
→ Liest/schreibt die JSON-Dateien in diesem Ordner
→ Gibt z. B. zurück: `AccountData`, `AccountMeta`

Stell dir vor:
Du gibst dem Repository die ID des Accounts. Es weiß, wo der Ordner ist, und liest/schreibt dort die `.json`-Dateien.

Was rein gehört:

- Methoden wie:
  - `get_account(account_id: str) -> AccountData`
  - `update_account(...)`
  - `save_account(...)`
- Verwendung von `AccountFiles` (aus `models.py`), um Pfade zu finden
- Benutzt `AccountMeta` und `AccountData` (ebenfalls aus `models.py`)

`paths_manager.py` (**Zentrale Pfad-Verwaltung = "Verzeichnisführer")

Was sie tut:
→ Arbeitet **ausschließlich** mit `accountPaths.json`
→ Hat nichts mit den `.json`-Dateien innerhalb der Accounts zu tun
→ Verwaltet:

- Welche Accounts existieren
- Wo deren Ordner liegen
- Welche ID sie haben

Was rein gehört:

- Funktion `add_account_path(meta: AccountMeta)`
- Funktion `remove_account_path(account_id: str)`
- Funktion `get_account_path(account_id: str)`
- Funktion `list_account_paths() -> list[AccountMeta]`

Stell dir vor:
Ein Inhaltsverzeichnis in einem Buch. Diese Datei weiß, *wo* du suchen musst – sie liest aber **nicht den Inhalt**.

`models.py` (**Datenmodelle = "Baupläne")

Was sie tut:
→ Definiert, wie Daten aussehen.
→ Diese Klassen enthalten **keine Logik**, nur **Attribute**

Muss enthalten:

1. `AccountMeta`: ID, Name, Ordnerpfad
2. `AccountData`: Die Inhalte des Accounts (z. B. SMTP, API, Benutzername...)
3. `AccountFiles`: Speichert Pfade zu allen `.json`-Dateien **innerhalb eines Account-Ordners**

Stell dir vor:
Ein "Plan", wie ein Account aussieht. So wie ein Formular mit Feldern.

`interface.py` (**Interface/Vertrag = "Was ein Repository können MUSS")

Was sie tut:
→ Definiert mit `Protocol`, welche Methoden ein Repository **haben muss**
→ Keine Logik, nur Methodensignaturen

Warum das wichtig ist:
Wenn du später z. B. eine andere Version vom Repository machen willst (mit SQLite), kannst du sie **austauschen**, solange sie dasselbe Interface erfüllt.

Was rein gehört:

- Klasse `AccountRepository(Protocol):`
- Methoden:

- `get_account(...) -> AccountData`
- `add_account(...)`
- `remove_account(...)`
- `update_account(...)`
- `list_accounts() -> list[AccountMeta]`

Stell dir vor:
Ein Vertrag: *"Egal wie du es machst, du musst diese Funktionen anbieten."*

`exceptions.py` (**Eigene Fehlerarten = "Klarere Fehlermeldungen")

Was sie tut:
→ Enthält **eigene Fehlerklassen**, z. B. `AccountNotFoundError`
→ Diese Klassen **erben** von `Exception`
→ Sie helfen dir, **genauer zu sagen, was falsch gelaufen ist**

Beispiel:

```
class AccountNotFoundError(Exception):
    def __init__(self, account_id: str):
        super().__init__(f"Kein Account mit der ID '{account_id}' gefunden.")
```

Stell dir vor:
Statt `ValueError` zu werfen, sagst du genau *was* passiert ist – und kannst gezielt darauf reagieren.

✅ Technische Datenflüsse (einfach erklärt)

📦 Account erstellen (Ablauf)

1. `main.py` sagt zu `manager.py`: „Erstelle neuen Account.“
2. `manager.py`:
    - Fragt per Input Daten ab
    - Ruft `paths_manager.py.add_account_path()` auf → speichert in `accountPaths.json`
    - Ruft `repository.py.add_account(...)` auf → erstellt Ordner, JSON-Dateien

- Loggt ggf. per `logger`

3. Ergebnis: Account ist vollständig angelegt

📦 Account laden

1. `main.py` sagt zu `manager.py`: „Lade Account mit ID X“
2. `manager.py` ruft:
   - `paths_manager.py.get_account_path()` auf → bekommt Pfad
   - `repository.py.get_account(...)` auf → liest JSON-Dateien im Ordner
3. Account-Daten werden als `AccountData` zurückgegeben

📦 Account löschen

1. `main.py` sagt zu `manager.py`: „Lösche Account mit ID X“
2. `manager.py` ruft:
   - `paths_manager.py.remove_account_path(...)`
   - `repository.py.remove_account(...)` (löscht Ordner)
3. Falls Account nicht existiert → wirf `AccountNotFoundError`

🧠 Wichtige Hinweise für deine Offline-Arbeit

- Jede Datei hat **eine einzige Verantwortung**
- `manager.py` ist die **einzige Datei**, die mit der Außenwelt (Benutzer, CLI) redet
- `paths_manager.py` ist die **einzige Datei**, die `accountPaths.json` anfasst
- `repository.py` ist die **einzige Datei**, die mit den Dateien **innerhalb** eines Accounts arbeitet

---

Wenn du willst, kann ich dir noch eine Mindmap oder ein ausgedrucktes Ablaufdiagramm (PNG oder PDF) zur Übersicht bauen – sag einfach Bescheid.