Perfekt – das ist die richtige Frage! Jetzt erkläre ich dir **jede Datei und ihre genaue Aufgabe in deinem konkreten Projekt** `ReMailD`.

Du bekommst für jede Datei:

- 🧠 **Klaren Zweck im Projekt**
- ⚙️ **Was sie mit den JSON-Dateien macht oder nicht**
- 🛠️ **Wie sie mit anderen Dateien zusammenspielt**
- 👨‍💼 **Einfaches Beispiel aus dem Alltag, damit du's dir vorstellen kannst**

📁 `account/` (Feature-Folder für Account-Verwaltung)

🧩 `models.py`

🧠 **Zweck:**

Enthält **nur die Datenstruktur** für Accounts – also wie du Accountdaten **im RAM hältst** (nicht wie du sie lädst/speicherst!).

⚙️ **Inhalt:**

- `AccountMeta`: enthält ID, Name, Pfad
- `AccountData`: enthält SMTP-Konfiguration, Schlüssel, Filter usw.

📦 **Wichtig für dein Projekt:**

Diese Klassen werden **von der Repository geladen** und dann in der Service-Schicht verwendet.

👨‍💼 **Alltag-Beispiel:**

Stell dir vor, das ist das **Formular**, das ein Kunde bei der Konto-Erstellung ausfüllt. Es liegt später **im Speicher** (nicht auf Papier).

📂 `repository.py`

🧠 **Zweck:**

Diese Datei enthält die konkrete **Implementierung zum Laden/Speichern der Accounts in `.json`-Dateien**.

⚙️ **Inhalt:**

- `JSONAccountRepository`: implementiert die Methoden aus `interface.py`
  z. B.:
  - `get_account(account_id)` → lädt `account.json` aus Pfad

- `list_accounts()` → liest `accountPaths.json`
- `add_account(...)` → schreibt Account in neuen JSON-Ordner

📦 **Wichtig für dein Projekt:**

Diese Datei ist die **einzige**, die weiß, wie deine JSON-Dateien funktionieren.
Sie weiß z. B., dass `accountPaths.json` eine Liste von gespeicherten Accounts enthält.

👨‍💼 **Alltag-Beispiel:**

Wie ein Archivar, der weiß, wie deine Ordner aussehen und wie man sie einsortiert.

🗒️ `interface.py`

🧠 **Zweck:**

Definiert, was ein Repository können **muss**, damit es vom Service genutzt werden darf. Du verwendest hier `Protocol` (aus `typing`).

⚙️ **Inhalt:**

```
class AccountRepository(Protocol):
    def get_account(self, id): ...
    def list_accounts(self): ...
    ...
```

📦 **Wichtig für dein Projekt:**

Dadurch ist deine `account_service.py` **nicht an JSON gebunden** – später kannst du z. B. `DatabaseAccountRepository` schreiben, ohne `service.py` zu ändern.

👨‍💼 **Alltag-Beispiel:**

Wie ein Vertrag: „Jeder Mitarbeiter im Lager muss Akten finden, speichern, löschen können.“

🛠️ `service.py`

🧠 **Zweck:**

Hier liegt die **Hauptlogik**, die du im CLI aufrufst:

- Konto erstellen
- Konto aktualisieren
- Konto löschen
- Konto auswählen

⚙️ **Inhalt:**

- Redet mit `repository.py`, um Accountdaten zu speichern/laden.

- Nutzt `models.py`, um mit `AccountMeta` und `AccountData` zu arbeiten.
- Löst Exceptions aus `exceptions.py` aus, wenn etwas schiefgeht.
- Nutzt `logging.debug/info/warning`, um Status zu melden.

📦 **Wichtig für dein Projekt:**

Hier entscheidest du z. B.:

- Wenn mehr als 20 Accounts → `AccountLimitReachedError`
- Welcher Account ist aktiv?
- Neue Accounts erstellen mit Name + zufälligem ID

👨‍💼 **Alltag-Beispiel:**

Das ist dein „Sachbearbeiter“, der sagt: „Erstelle ein neues Konto“, ruft den Archivar (`repository.py`) und sagt, wie das Formular (`models.py`) auszufüllen ist.

🚨 `exceptions.py`

🧠 **Zweck:**

Definiert **alle Fehlerfälle** rund um Accounts – **damit du sie gezielt behandeln kannst**.

⚙️ **Inhalt:**

- `AccountNotFoundError`
- `AccountLimitReachedError`
- `AccountAlreadyExistsError`
- ...

📦 **Wichtig für dein Projekt:**

- Wenn z. B. ein Account nicht geladen werden kann → wirfst du `AccountNotFoundError`, nicht `FileNotFoundError`.
- Bessere Lesbarkeit & Testbarkeit deines Codes.

👨‍💼 **Alltag-Beispiel:**

Statt „Fehler 404“ sagst du: „Account nicht gefunden“ – direkt, spezifisch, verständlich.

🔧 `utils.py` (nur für account-spezifische Helfer)

🧠 **Zweck:**

Kleine Hilfsfunktionen, die **nicht zum Service gehören**, aber z. B. Pfade validieren, zufällige Account-IDs generieren, Namen prüfen usw.

⚙️ **Inhalt:**

- `generate_account_id()`
- `is_valid_account_name(name)`
- `get_account_folder_path(account_id)`

📦 **Wichtig für dein Projekt:**

Halte `service.py` schlank, indem du kleine technische Helfer auslagerst.

👨‍💼 **Alltag-Beispiel:**

Wie ein Taschenmesser – du holst es raus, wenn du mal kurz was zuschneiden musst, aber trägst es nicht ständig in der Hand.

📁 `config/`

📄 `settings.py`

🧠 **Zweck:**

Globale Konfiguration wie:

- Max. Account-Anzahl
- Standard-JSON-Dateipfade
- Logging-Level
- Config-Verzeichnis

⚙️ **Inhalt:**

```
MAX_ACCOUNTS = 20
CONFIG_DIR = Path.home() / ".remaild"
ACCOUNT_PATHS_FILE = CONFIG_DIR / "accountPaths.json"
LOG_LEVEL = "DEBUG"
```

📦 **Wichtig für dein Projekt:**

Von überall nutzbar – z. B. `repository.py` nutzt `ACCOUNT_PATHS_FILE` als Pfad.

👨‍💼 **Alltag-Beispiel:**

Wie ein Schaltkasten im Büro: einmal einstellen – dann läuft der Strom überall richtig.

📁 `utils/`

📄 `path_utils.py`

🧠 **Zweck:**

Allgemeine (nicht Account-spezifische) Helfer rund um Pfade & Dateisystem.

⚙️ **Inhalt:**

- `ensure_config_dir()` → stellt sicher, dass `.remaild/` existiert
- `get_json_path()` → gibt den vollen Pfad zur `accountPaths.json` zurück

📦 **Wichtig für dein Projekt:**

Wird z. B. vom Repository genutzt, bevor ein Account gespeichert wird.

👨‍💼 **Alltag-Beispiel:**

Wie ein Hausmeister: Er stellt sicher, dass der Raum existiert, bevor jemand versucht, etwas reinzulegen.

📁 `cli/`

📄 `main.py`

🧠 **Zweck:**

Der **Startpunkt für das CLI** – dort initialisierst du alles und führst aus, was der Benutzer will.

⚙️ **Inhalt:**

- Importiere `AccountService` und `JSONAccountRepository`
- Verbinde beides → `service = AccountService(repo)`
- Erzeuge Kommandos mit `argparse` oder `click`

📦 **Wichtig für dein Projekt:**

Dies ist dein **User Interface**, das z. B. sagt:

```
remaild account create --name "BenutzerX"
```

→ und ruft dann:

```
service.create_account("BenutzerX", Path(...))
```

👨‍💼 **Alltag-Beispiel:**

Wie der Tresen in einer Bank – Benutzer kommt rein und sagt, was er tun will.

✅ Fazit: Das perfekte System für ReMailD

Teil	Aufgabe im Projekt	Dateien
Datenmodelle	Struktur der Accounts im RAM	`models.py`
Datenzugriff	JSON-Dateien lesen/schreiben	`repository.py`, `interface.py`
Businesslogik	Accounts erstellen, löschen, auswählen	`service.py`
Fehlerbehandlung	Benutzerfreundliche Fehler	`exceptions.py`
Hilfsfunktionen	Validierung, IDs, Pfadfunktionen	`utils.py`, `path_utils.py`
Zentrale Config	Konstanten & globale Einstellungen	`settings.py`
Benutzeroberfläche	CLI-Tool zum Steuern des Programms	`main.py`

Wenn du dieses System so baust, kannst du:

- Accounts jederzeit austauschen, erweitern, testen
- Logging und Exceptions sauber handhaben
- Dein CLI erweitern ohne Chaos
- Alle Dateien logisch finden und erweitern

Du arbeitest dann **genau so**, wie es große Python-Projekte (z. B. `FastAPI`, `Typer`, `Pydantic`) auch machen – nur **einsteigerfreundlich heruntergebrochen**.

Wenn du jetzt mit diesem System arbeitest, bist du strukturell **absolut auf Profi-Niveau**.