Detail-Extractor – Architektur & Funktionsweise

1. Ziel des Extractors

Der Detail-Extractor ist eine deterministische Parsing-Engine zur strukturierten Extraktion von Job-Detailinformationen.

Er erfüllt folgende Aufgaben:

- Wandelt JSON-, HTML- oder XML-Responses in das standardisierte ZIELSCHEMA (v0.1) um
- Arbeitet vollständig ohne LLM im Live-Run
- Nutzt ausschließlich hart definierte DOM-Strukturen oder JSON-Pfade
- Trennt Parsing-Logik sauber vom Company-Script und vom Schema-Builder

Der Extractor enthält keine Heuristiken und keine Überschriften-Erkennung.

---

2. Architekturprinzip

Der Extractor besteht aus zwei klar getrennten Teilen:

A) Mapping-Block (EDITIERBAR)

```
DETAIL_MAPPING_V01 = {...}
```

Dieser Block definiert:

- unterstützte Input-Typen ("json", "html", "xml")
- JSON-Pfade für einzelne Felder
- CSS-Selektoren für DOM-Strukturen
- Modus der Extraktion (text, html, list)
- optionale DOM-Elemente, die entfernt werden sollen

Dieser Block wird später per KI aus Beispielen generiert und manuell eingefügt.

B) Engine (NICHT ANFASSEN)

Die Engine übernimmt:

- JSON-Pfad-Auswertung
- DOM-Selektor-Auswertung
- HTML → Text-Bereinigung
- Listenextraktion
- Rückgabe strukturierter Sections

Die Engine ist generisch und bleibt für alle Firmen identisch.

---

3. Unterstützte Input-Typen

Der Extractor kann drei Typen verarbeiten:

JSON

- Verarbeitung über definierte Key-Pfade
- Optionales HTML-Cleaning einzelner Felder
- Unterstützt Listen-Arrays direkt aus JSON

HTML

- Verarbeitung über CSS-Selektoren
- Root-Selector optional
- List-Container + item_selector

XML

- Wird mit BeautifulSoup im XML-Modus geparst
- Funktional identisch zu HTML-DOM-Verarbeitung

---

4. Extrahierte Zielfelder

Der Extractor liefert folgendes strukturierte Ergebnis:

```
{
  "fulltext": str | None,
  "overview": str | None,
  "responsibilities": List[str],
  "requirements": List[str],
  "additional": List[str],
  "benefits": List[str],
  "process": str | None
}
```

Diese Felder werden anschließend in das Pydantic-Schema (`DetailJobV01`) geschrieben.

---

5. JSON-Mapping-Struktur

Beispiel:

```
"json": {
  "paths": {
    "fulltext": ["data","job","descriptionHtml"],
    "responsibilities_items": ["data","job","sections","responsibilities"],
```

```
  },
  "html_fields": ["fulltext"],
  "html_item_fields": ["responsibilities_items"]
}
```

- Pfade sind Listen von Keys
- Integer-Indizes möglich
- HTML-Felder werden automatisch bereinigt

---

6. DOM-Mapping-Struktur

Beispiel:

```
"dom": {
  "root_selector": "div.job-description",
  "fields": {
    "overview": {"selector": "div.overview", "mode": "text"}
  },
  "lists": {
    "responsibilities_items": {
        "selector": "div.responsibilities",
        "item_selector": "li",
        "mode": "text"
    }
  }
}
```

- root_selector begrenzt Extraktionsbereich
- fields = Einzeltexte
- lists = Listenextraktion
- mode = "text" oder "html"

---

7. Clean-Up-Mechanismen

Automatisch enthalten:

- HTML → Klartext Konvertierung
- Mehrfach-Whitespace-Reduktion
- Entfernen leerer Strings
- Null-Rückgabe bei fehlenden Feldern

Keine inhaltliche Interpretation.

---

8. Verantwortlichkeiten im Gesamtsystem

Komponente	Aufgabe
Company-Script	Orchestrierung & Speicherung
Extractor	Strukturierte Detail-Extraktion
Schema-Builder	Pydantic-Validierung
GCS-Upload	Persistenz

Der Extractor kennt keine DB-Logik und keine Cloud-Logik.

---

9. Erweiterbarkeit

Neue Firmen: - Neues Extractor-File - Eigener DETAIL_MAPPING_V01 Block

Neue Schema-Version: - Anpassung im Builder - Extractor unverändert

---

10. Zukunft: KI-gestützte Mapping-Generierung

Workflow:

1. 3–5 Detail-Responses bereitstellen
2. KI erzeugt ausschließlich DETAIL_MAPPING_V01
3. Block wird 1:1 in Extractor kopiert
4. Kein LLM im Produktionslauf

---

11. Design-Philosophie

- Deterministisch
- Wartbar
- Trennung von Parsing & Schema
- Kein implizites Verhalten
- Keine Überschriften-Heuristik
- Keine semantische Interpretation

Ende der Architektur-Zusammenfassung.