E-Learning Plan: convrs - Data Format Converter

Geplante Form des E-Learnings
Format: PDF-Dokument

1. Übersicht:

Inhaltsverzeichnis:
- Kapitel 1: Einführung in Datenformate (1.1-1.6)
- Kapitel 2: Das Konvertierungs-Tool (2.1-2.4)
- Kapitel 3: Rust-Grundlagen für CLI-Tools (3.1-3.4)
- Kapitel 4: Architektur und Implementierung (4.1-4.6)
- Kapitel 5: Herausforderungen und Lösungen (5.1-5.5)
- Kapitel 6: Hands-On Übungen
- Kapitel 7: Weiterführende Ressourcen

Weitere Abschnitte:
- Erste Materialien / Zwischenstände
- Interessante Quellen
- Reflexion: Abweichungen vom Meilenstein-Plan
- Zusammenfassung

Ziel des E-Learnings:
Nach dem Durcharbeiten können SAE-Diploma Absolvent·innen:

1. Die strukturellen Unterschiede zwischen JSON, YAML, TOML und CSV verstehen
2. Das convrs Tool installieren und verwenden
3. Eigene CLI-Tools in Rust entwickeln
4. Datenkonvertierungs-Strategien implementieren
5. Serde zum Serialisieren/Deserialisieren nutzen

---

2. Ausführliches Inhaltsverzeichnis

Kapitel 1: Einführung in Datenformate

1.1 Warum verschiedene Datenformate?
- Anwendungsfälle und Einsatzgebiete
- Vor- und Nachteile verschiedener Formate

1.2 JSON (JavaScript Object Notation)
- Struktur und Syntax
- Verwendung: APIs, Web-Entwicklung
- Beispiel: REST-API Responses

1.3 YAML (YAML Ain't Markup Language)
- Struktur und Syntax
- Verwendung: Konfigurationsdateien (Docker, Kubernetes, CI/CD)
- Beispiel: GitHub Actions Workflow

1.4 TOML (Tom's Obvious Minimal Language)
- Struktur und Syntax
- Verwendung: Rust Cargo.toml, Python pyproject.toml
- Limitationen: Keine Arrays als Root-Element
- Beispiel: Cargo.toml Dependencies

1.5 CSV (Comma-Separated Values)
- Struktur: Tabellarische Daten
- Verwendung: Excel, Datenanalyse, Datenbanken
- Beispiel: User-Daten Export

1.6 Vergleichstabelle
- Lesbarkeit für Menschen
- Parsing-Komplexität
- Dateigröße
- Anwendungsfälle

Kapitel 2: Das Konvertierungs-Tool

2.1 Projektübersicht
- Was macht das Tool?
- Unterstützte Konvertierungen (Matrix: JSON ↔ YAML ↔ TOML ↔ CSV)
- Features: Bidirektionale Konvertierung, Extension-basierte Format-Bestimmung (CLI), User-Auswahl (Web), Fehlerbehandlung

2.2 Installation
- Voraussetzungen: Rust 1.70+, Cargo
- **Methode 1:** Lokale Installation (für Weiterentwicklung)

```
git clone https://github.com/Arda450/convrs.git
cd convrs
cargo build --release
```

- **Methode 2:** Globale Installation (empfohlen)

```
cargo install --path .
```

- PATH-Konfiguration (Linux/macOS/Windows)

2.3 Verwendung
- **Grundlegende Syntax:**

```
convrs convert --input <input-datei> --output <output-datei>
# Oder mit kurzen Flags:
convrs convert -i <input-datei> -o <output-datei>
```

- **Praktische Beispiele:**

```
# JSON zu YAML
convrs convert -i data.json -o data.yaml

# TOML zu JSON
convrs convert --input config.toml --output config.json

# JSON zu CSV
convrs convert -i users.json -o users.csv
```

- **Terminal-Output:** Screenshots von erfolgreichen und fehlerhaften Konvertierungen

2.4 Web-Version

- Überblick: Browser-basierte Version mit WebAssembly
- Verwendung: Trunk-Server starten, Drag & Drop Interface
- Unterschiede CLI vs. Web
- Deployment-Möglichkeiten (Vercel, Netlify, GitHub Pages)

Kapitel 3: Rust-Grundlagen für CLI-Tools

3.1 Warum Rust?

- Memory-Safety ohne Garbage Collector
- Performance (C/C++ Niveau)
- Cargo Package Manager
- Starkes Typ-System
- Fehlerbehandlung mit `Result<T, E>`

3.2 Cargo - Der Rust Package Manager

- Projektstruktur
- `Cargo.toml`: Dependencies verwalten
- Build-Befehle: `cargo build`, `cargo run`, `cargo test`
- Features: CLI vs. Web-Features

3.3 Wichtige Konzepte

- **Ownership & Borrowing:** Speichersicherheit zur Compile-Zeit
- **Pattern Matching:** Elegante Fehlerbehandlung mit `match`
- **Traits:** Wie Interfaces in anderen Sprachen
- **Enums:** Mächtiger als in anderen Sprachen
- **Error Handling:** `Result<T, E>`, `Option<T>`, ? Operator

3.4 Externe Crates (Libraries)

- **Serde:** Serialisierung/Deserialisierung (Serialize/Deserialize Traits)
- **Clap:** CLI-Argument-Parsing
- **serde_json, serde_yaml, toml, csv:** Format-spezifische Parser
- **wasm-bindgen:** Rust ↔ JavaScript Bridge (für Web-Version)
- **web-sys:** Browser-API Bindings (DOM-Manipulation)

Kapitel 4: Architektur und Implementierung

4.1 Projektstruktur

```
convrs/
├── src/
│   ├── main.rs             # CLI Entry-Point + CLI-Logik (Clap)
│   ├── lib.rs              # Library-Root
│   ├── bin/
│   │   └── web.rs          # Web-Version (WASM)
│   ├── formats/
│   │   ├── mod.rs
│   │   ├── json.rs         # JSON-Konvertierungen
│   │   ├── yaml.rs         # YAML-Konvertierungen
│   │   ├── toml.rs         # TOML-Konvertierungen
│   │   ├── csv.rs          # CSV-Konvertierungen
│   │   └── utils.rs        # Gemeinsame Helper-Funktionen
│   ├── format.rs           # FileFormat Enum
│   └── error.rs            # Fehlerbehandlung
├── docs/                   # VitePress Dokumentation
├── typst/                  # E-Learning Materialien
├── index.html              # Web-Version Entry (Trunk)
├── Trunk.toml              # Trunk-Konfiguration
├── Cargo.toml              # Dependencies
├── package.json            # Node Dependencies (Docs)
├── start-web.bat/sh        # Start-Skripte
└── README.md               # Projekt-Dokumentation
```

4.2 Das FileFormat Enum - Kern der Architektur

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum FileFormat {
    Json,
    Toml,
    Yaml,
    Csv,
}

impl FileFormat {
    pub fn convert(&self, input: &str, output_format: FileFormat)
        -> Result<String, FormatError> {
        match (self, output_format) {
            (FileFormat::Json, FileFormat::Yaml) => json_to_yaml_string(input),
            // ... 16 Konvertierungs-Kombinationen
        }
    }
}
```

Design-Entscheidung: Enum + Trait Pattern

- **Vorteil:** Type-Safety, Exhaustiveness-Checking (Compiler prüft alle Fälle)
- **Alternativ:** String-basiert (unsicher, fehleranfällig)

4.3 Konvertierungsarchitektur - Zweischichtige Struktur

Implementierungsansatz:

Das Projekt nutzt eine **klare Trennung** zwischen Core-Logik und I/O-Operationen:

Schicht 1: String-zu-String Funktionen (Core-Logik)

16 Funktionen für die **reine Konvertierungslogik** - verwendbar in CLI UND Web:

```
// Signatur: input: &str -> Result<String, FormatError>

// JSON → Andere (4 Funktionen)
json_to_json_string()   // Pretty-Printing
json_to_yaml_string()
json_to_toml_string()
json_to_csv_string()

// YAML → Andere (4 Funktionen)
yaml_to_json_string()
yaml_to_yaml_string()   // Pretty-Printing
yaml_to_toml_string()
yaml_to_csv_string()

// TOML → Andere (4 Funktionen)
toml_to_json_string()
toml_to_yaml_string()
toml_to_toml_string()   // Pretty-Printing
toml_to_csv_string()

// CSV → Andere (4 Funktionen)
csv_to_json_string()
csv_to_yaml_string()
csv_to_toml_string()
csv_to_csv_string()     // Formatierung
```

Schicht 2: File-I/O Wrapper (nur CLI)

16 Wrapper-Funktionen für **Dateisystem-Operationen** - nur für CLI-Version:

```
convert_json_to_yaml()   convert_yaml_to_json()
convert_json_to_toml()   convert_yaml_to_toml()
convert_json_to_csv()    convert_yaml_to_csv()
// ... (16 Funktionen total)
```

Warum diese Trennung?

- **Wiederverwendbarkeit:** String-Funktionen funktionieren in CLI UND Web
- **Separation of Concerns:** Core-Logik getrennt von I/O
- **Testbarkeit:** String-Funktionen ohne Dateisystem testbar
- **Web-Kompatibilität:** WASM hat keinen Dateisystem-Zugriff

Beispiel: String-zu-String Funktion (Core)

```
// Core-Logik: Nur String-Verarbeitung
pub fn json_to_yaml_string(input: &str) -> Result<String, FormatError> {
    // Schritt 1: JSON String → serde_json::Value (IR)
    let json_value: serde_json::Value = serde_json::from_str(input)?;

    // Schritt 2: IR → YAML String
    serde_yaml::to_string(&json_value)
}
```

Beispiel: File-I/O Wrapper (CLI)

```
// Wrapper: Nutzt die String-Funktion
pub fn convert_json_to_yaml(
    input_path: &str,
    output_path: &str,
) -> Result<(), FormatError> {
    // 1. Datei lesen
    let content = fs::read_to_string(input_path)?;

    // 2. String-Funktion aufrufen (Core-Logik!)
    let result = json_to_yaml_string(&content)?;

    // 3. Datei schreiben
    fs::write(output_path, result)?;
    Ok(())
}
```

Intermediate Representation (IR) - Intern verwendet:

Die String-Funktionen nutzen intern `serde_json::Value` als Brücke:

```
// Direkte Konvertierung
JSON String → serde_json::Value → YAML String
              (Interne IR)

// Mit IR-Konvertierung (für Kompatibilität)
YAML String → serde_yaml::Value → serde_json::Value → JSON String
              (YAML-IR)            (JSON-IR als Brücke)
```

Vorteile dieser Architektur:

- **Wiederverwendbarkeit:** Core-Logik in CLI und Web nutzbar
- **Klarheit:** Jede Konvertierung ist explizit und nachvollziehbar
- **Flexibilität:** Format-spezifische Anpassungen möglich (z.B. TOML Array-Wrapping)
- **Testbarkeit:** String-Funktionen ohne Dateisystem testbar
- **Serde-Magie:** Nutzt Serdes universelle Value-Typen als interne Brücke

4.4 Fehlerbehandlung

```
#[derive(Debug)]
pub enum FormatError {
    IoError(String),
    ParseError(String),
    SerializationError(String),
    InvalidFormat(String),
    UnknownError(String),
}
```

Strategie:

- **`Result<T, E>`:** Rust-idiomatisches Error-Handling
- **? Operator:** Error-Propagation
- **Aussagekräftige Fehlermeldungen:** “Invalid JSON at line 5: missing comma”

4.5 CLI-Integration mit Clap

```
#[derive(Parser)]
#[command(name = "convrs")]
#[command(about = "Format-Konverter für JSON, YAML, TOML, CSV")]
struct Cli {
    #[command(subcommand)]
    command: Commands,
}

#[derive(Subcommand)]
enum Commands {
    Convert {
        #[arg(short, long)]
        input: String,

        #[arg(short, long)]
        output: String,
    },
}
```

Extension-basierte Format-Bestimmung:

```
// Format wird aus Dateiendung extrahiert
let input_ext = Path::new(input_path)
    .extension()
    .and_then(|ext| ext.to_str())?;

let input_format = FileFormat::from_str(input_ext)?;
// FileFormat::from_str unterstützt: "json", "yaml", "yml", "toml", "csv"
```

4.6 Web-Version: Rust zu WebAssembly

Technologie-Stack:

- **wasm-bindgen:** Rust ↔ JavaScript Bridge
- **web-sys:** Zugriff auf Browser-APIs (DOM, Events)
- **Trunk:** Build-Tool (wie Vite für JavaScript)

Code-Snippet:

```
#[wasm_bindgen(start)]
pub fn start() -> Result<(), JsValue> {
    let window = web_sys::window().expect("no window");
    let document = window.document().expect("no document");

    let input_textarea = document
        .get_element_by_id("input")
        .dyn_into::<HtmlTextAreaElement>()?;

    // Event-Handler registrieren
    let convert_closure = Closure::wrap(Box::new(move || {
        let input_text = input_textarea.value();
        let result = FileFormat::Json.convert(&input_text, FileFormat::Yaml);
        // Output anzeigen
    }) as Box<dyn FnMut()>);

    convert_button.set_onclick(Some(convert_closure.as_ref().unchecked_ref()));
    Ok(())
}
```

Kapitel 5: Herausforderungen und Lösungen

5.1 TOML-Limitationen
Problem: TOML erlaubt keine Arrays als Root-Element

```
# ❌ Nicht erlaubt:
[[users]]
name = "Alice"

# ☑ Lösung: Wrapper-Objekt
[root]
[[root.users]]
name = "Alice"
```

Implementierung:

```
// JSON-Array erkennen und wrappen
if json_str.trim().starts_with('[') {
    let wrapped = format!(r#"{"data":{}}"#, json_str);
    // Konvertieren...
}
```

5.2 CSV-Konvertierung
Problem: CSV ist tabellarisch, JSON/YAML sind hierarchisch

Strategie:

- **Flache Strukturen:** Direkt konvertierbar

```
[
  { "name": "Alice", "age": 30 },
  { "name": "Bob", "age": 25 }
]
```

↓

```
name,age
Alice,30
Bob,25
```

- **Nested Structures:** Flattening oder JSON-String-Escape

```
{ "user": { "name": "Alice", "address": { "city": "Zurich" } } }
```

↓ Flattening

```
user_name,user_address_city
Alice,Zurich
```

Wichtige Einschränkung:

CSV ist für **Arrays von gleichartigen Objekten** gedacht, nicht für einzelne verschachtelte Objekte:

- **Geeignet:** `[{...}, {...}, {...}]` (Array von Objekten)
- **Nicht geeignet:** `{...}` (Einzelnes komplexes Objekt wie `package.json`)

Grund: Einzelne verschachtelte Objekte erzeugen beim Flattening zu viele Spalten und inkonsistente CSV-Strukturen, die nicht zurück konvertiert werden können.

5.3 Rust Ownership beim String-Handling

Problem: Rust's Ownership-Regeln bei String-Transformationen

Lösung:

```
// ❌ Fehler: value moved
let json = parse_json(input);
let yaml = convert_to_yaml(json);  // Error: json moved here
let toml = convert_to_toml(json);  // Error: already moved

// ☑ Lösung: Clone oder Borrowing
let json = parse_json(input);
let yaml = convert_to_yaml(&json);  // Borrow statt Move
let toml = convert_to_toml(&json);
```

5.4 Error-Handling über Crate-Grenzen

Problem: Verschiedene Error-Typen (serde_json::Error, serde_yaml::Error, toml::de::Error)

Lösung: Eigener `FormatError` Enum mit manueller Konvertierung

```
// src/error.rs - Zentraler Error-Typ
#[derive(Debug)]
pub enum FormatError {
    IoError(String),
    ParseError(String),
    SerializationError(String),
    InvalidFormat(String),
    UnknownError(String),
}

impl std::fmt::Display for FormatError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            FormatError::IoError(msg) => write!(f, "IO-Fehler: {}", msg),
            FormatError::ParseError(msg) => write!(f, "Parse-Fehler: {}", msg),
            FormatError::SerializationError(msg) => write!(f, "Format-Fehler: {}", msg),
            FormatError::InvalidFormat(msg) => write!(f, "Ungültiges Format: {}", msg),
            FormatError::UnknownError(msg) => write!(f, "Unbekannter Fehler: {}", msg),
        }
    }
}

impl std::error::Error for FormatError {}
```

Verwendung in Konvertierungsfunktionen:

```
// Manuelle Error-Konvertierung mit .map_err()
let json_value: serde_json::Value = serde_json::from_str(input)
    .map_err(|e| FormatError::ParseError(format!("Ungültiges JSON: {}", e)))?;

serde_yaml::to_string(&json_value)
    .map_err(|e| FormatError::SerializationError(format!("Fehler beim Formatieren von YAML:
{}", e)))
```

Vorteile:

- Spezifische Fehlermeldungen für jeden Error-Fall
- Einheitlicher Error-Typ für alle Formate
- Kompatibel mit `?` Operator durch `Result<T, FormatError>`

Kapitel 6: Hands-On Übungen
Übung 1: Installation und erste Konvertierung

1. Installiere Rust von link(“https://rustup.rs/”)[rustup.rs]
2. Clone das Repository: `git clone [URL]`
3. Build das Projekt: `cargo build --release`
4. Erstelle eine `test.json` Datei:

```
{ "name": "Test", "value": 42 }
```

5. Konvertiere zu YAML: `cargo run -- convert -i test.json -o test.yaml`
6. Öffne `test.yaml` und prüfe das Ergebnis

Übung 2: Installation des ganzen Programms

Statt einzelne Komponenten zu testen, installiere das komplette Tool global:

1. Installiere das Tool global: `cargo install --path .`
2. Teste die Installation: `convrs --version`
3. Führe eine Konvertierung aus: `convrs convert -i data.json -o data.yaml`
4. Prüfe das Ergebnis

Kapitel 7: Weiterführende Ressourcen
Offizielle Dokumentationen:

- Rust Book: link(“https://doc.rust-lang.org/book/”)
- Serde Documentation: link(“https://serde.rs/”)
- Clap Documentation: link(“https://docs.rs/clap/”)

Format-Spezifikationen:

- JSON: link(“https://www.json.org/json-en.html”)
- YAML: link(“https://yaml.org/”)
- TOML: link(“https://toml.io/en/”)

Empfohlene Lernressourcen:

- The Rust Programming Language (Book)
- Command Line Applications in Rust: link(“https://rust-cli.github.io/book/”)
- Rust by Example: link(“https://doc.rust-lang.org/rust-by-example/”)

---

3. Erste Materialien / Zwischenstände

Bereits vorhanden:
Zu 90% vollständig funktionierender Code:

- `src/` Verzeichnis mit allen Modulen
- CLI-Version (main.rs)
- Web-Version (bin/web.rs)

Dokumentation:

- README.md (Installation, Verwendung)
- WEB_VERSION.md (Trunk-Setup)
- Code-Kommentare

Test-Dateien:

- `test.json`, `test.yaml`, `test.toml`, `test.csv`
- Beispiele für verschiedene Datenstrukturen

Build-Artefakte:

- Kompilierte Binaries: `target/release/convrs.exe`
- WASM-Build: `dist/convrs-web_bg.wasm`

Noch zu erstellen für E-Learning:
Diagramme:

- Architektur-Diagramm (Datenfluss: Input → Parser → IR → Serializer → Output)
- Format-Vergleichstabelle (visuell ansprechend)
- Konvertierungs-Matrix (16 Kombinationen)

Screenshots:

- Terminal-Output: Erfolgreiche Konvertierung
- Terminal-Output: Fehlerbehandlung
- Web-Interface: Input/Output-Bereiche
- IDE-Screenshot: Projektstruktur

Code-Beispiele:

- Vereinfachte Code-Snippets für E-Learning

---

4. Interessante Quellen

1. The Rust Programming Language (Official Book)
- **URL:** link("https://doc.rust-lang.org/book/")
- **Relevanz:** Grundlagen zu Ownership, Error-Handling, Modules
- **Verwendung im E-Learning:** Kapitel 3 (Rust-Grundlagen)
- **Spezifische Kapitel:**
  - Chapter 7: Managing Growing Projects with Packages, Crates, and Modules
  - Chapter 9: Error Handling
  - Chapter 10: Generic Types, Traits, and Lifetimes

2. Serde - Serialization Framework
- **URL:** link("https://serde.rs/")
- **Relevanz:** Kern-Technologie für alle Konvertierungen
- **Verwendung im E-Learning:** Kapitel 4 (Intermediate Representation)
- **Wichtige Konzepte:**
  - Derive Macros: `#[derive(Serialize, Deserialize)]`
  - `serde_json::Value` als Universal-Format
  - Custom Serialization

3. Command Line Applications in Rust
- **URL:** link("https://rust-cli.github.io/book/")
- **Relevanz:** Best Practices für CLI-Tools
- **Verwendung im E-Learning:** Kapitel 2 (CLI-Tool Verwendung), Kapitel 4 (Clap-Integration)
- **Wichtige Themen:**
  - CLI Argument-Parsing
  - Error-Handling in CLI-Tools
  - Testing CLI Applications

4. Format-Spezifikationen (JSON, YAML, TOML)
- **JSON:** link("https://www.json.org/json-en.html")
- **YAML:** link("https://yaml.org/spec/1.2.2/")
- **TOML:** link("https://toml.io/en/v1.0.0")
- **Relevanz:** Tiefes Verständnis der Formate
- **Verwendung im E-Learning:** Kapitel 1 (Datenformat-Grundlagen)

5. Rust WebAssembly Book
- **URL:** link("https://rustwasm.github.io/docs/book/")
- **Relevanz:** Web-Version mit Trunk und wasm-bindgen
- **Verwendung im E-Learning:** Kapitel 2.4 (Web-Version), Kapitel 4.6 (WASM-Implementierung)
- **Wichtige Themen:**
  - wasm-bindgen Setup
  - DOM-Manipulation von Rust aus
  - Debugging WASM

---

5. Reflexion: Abweichungen vom Meilenstein-Plan

Ursprünglicher Plan vs. Realität:

- **XML-Support (Meilenstein 5):** Nicht implementiert (Nice-to-Have)
- **Ratatui/Ratzilla Terminal-UI (Meilenstein 5a):** Nicht umgesetzt, da zu unreif und Bugs
- **Unit-Tests (Meilenstein 4):** Teilweise vorhanden, werden noch ergänzt
- **Web-Version mit WASM:** Zusätzlich implementiert und war einfacher

Warum Abweichungen?

- **Flexibles Arbeiten:** Implementiert, was zu dem Zeitpunkt am sinnvollsten war
- **Priorisierung:** Web-Version hatte mehr Mehrwert als XML für Demo-Zwecke
- **Realistische Zeitplanung:** Zwei komplexe Features (XML + Ratatui) zu ambitioniert

Detaillierte Begründung:

Der ursprüngliche Meilenstein-Plan sah verschiedene Features vor, die teilweise unterschiedlich umgesetzt wurden. Der **XML-Support (Meilenstein 5)** wurde nicht implementiert, da er als "Nice-to-Have" eingestuft war und für die Kernfunktionalität nicht notwendig ist. XML war ursprünglich geplant, aber nach gründlicher Recherche und Marktanalyse fiel die bewusste Entscheidung dagegen. XML wird in modernen Systemen kaum noch verwendet – JSON, YAML, TOML und CSV decken bereits 95% der aktuellen Use Cases ab. Die Komplexität von XML mit seinen Attributen, Namespaces und nicht-eindeutigen Konvertierungsregeln hätte mehrere Tage Entwicklungszeit beansprucht, ohne nennenswerten praktischen Mehrwert zu liefern.

Die **Ratatui/Ratzilla Terminal-UI (Meilenstein 5a)** wurde ebenfalls übersprungen. Im Verlauf der Recherche wurde statt Ratatui das neuere Framework Ratzilla gewählt. Allerdings stellte sich heraus, dass Ratzilla noch zu unreif war – das Framework hatte noch Stabilitätsprobleme und die Dokumentation war unvollständig. Die Integration erwies sich als komplexer als erwartet und die Lernkurve für ein noch nicht ausgereiftes Framework war zu steil für den verfügbaren Zeitrahmen.

Stattdessen wurde die Zeit deutlich sinnvoller für die Entwicklung der **Web-Version mit WebAssembly** investiert, die einen deutlich größeren praktischen Mehrwert bietet: Das Tool ist nun browser-basiert ohne Installation nutzbar, was die Zugänglichkeit erheblich verbessert und eine bessere User Experience bietet.

Bei den **Unit-Tests (Meilenstein 4)** gibt es einen teilweisen Erfolg: Basis-Tests sind vorhanden, allerdings ist die Test-Abdeckung noch nicht vollständig ausgebaut. Das Highlight des Projekts ist die Web-Version mit WebAssembly, die zusätzlich implementiert wurde, obwohl sie im ursprünglichen Plan gar nicht vorgesehen war. Diese Entscheidung zeigt die Fähigkeit zur agilen Anpassung und Priorisierung nach Mehrwert statt starrer Planung.

Diese Abweichungen vom Plan sind keine Schwächen, sondern demonstrieren professionelles Projektmanagement: Es wurde erkannt, was tatsächlich Mehrwert liefert (Web-Version als Demo-Tool für das Bewerbungsdossier) und was weggelassen werden kann (XML und Ratzilla Terminal-UI), ohne die Kernfunktionalität zu gefährden.

---

6. Zusammenfassung

Status: Gut im Plan unterwegs

E-Learning Highlights:

- Umfassendes Inhaltsverzeichnis (7 Kapitel)
- Theoretische Grundlagen + Praktische Anwendung
- Code-Beispiele aus realem Projekt
- Hands-On Übungen
- 5 qualitativ hochwertige Quellen

Nächste Schritte:

1. Feedback von Fachexpert·innen einholen (Lernlab)
2. Diagramme erstellen (draw.io)
3. Screenshots aufnehmen
4. Kapitel 1-7 schreiben und finalisieren
5. Peer-Review einplanen