Inhaltsverzeichnis

1. Lernziele dieses E-Learnings

1. Die Architektur eines plattformübergreifenden Tools nachvollziehen, bei dem eine gemeinsame Rust-Codebasis sowohl als Web-Anwendung (WebAssembly) im Browser als auch als CLI-Tool auf der Kommandozeile läuft.
2. Die strukturellen Unterschiede zwischen JSON, YAML, TOML und CSV verstehen und bewerten, welches Format in welchem Kontext am besten geeignet ist.
3. Praktische Konvertierungsszenarien kennen.
4. Verstehen, wie Serde zum Serialisieren und Deserialisieren verschiedener Formate genutzt wird.
5. Das convrs-Tool installieren und für die Konvertierung zwischen allen vier Formaten verwenden.
6. Die Web-Version von convrs mit Trunk starten und im Browser nutzen.

---

2. Einführung in Datenformate

2.1. Warum verschiedene Datenformate?

In der modernen Softwareentwicklung werden Daten ständig zwischen Systemen ausgetauscht. Ob APIs, Konfigurationsdateien, Datenbanken oder Automatisierungspipelines: Überall begegnen uns strukturierte Datenformate. Jedes Format wurde für bestimmte Anwendungsfälle entwickelt und bringt eigene Stärken und Schwächen mit.

Die vier wichtigsten Formate in der heutigen Praxis sind:

- **JSON** (JavaScript Object Notation): Standard für Web-APIs und Datenaustausch
- **YAML** (YAML Ain't Markup Language): Beliebt für Konfigurationsdateien
- **TOML** (Tom's Obvious Minimal Language): Einfache, explizite Konfigurationssprache
- **CSV** (Comma-Separated Values): Tabellarische Daten für Analyse und Export

Diese vier Formate decken in der Praxis den Grossteil der Anwendungsfälle ab (APIs, Config, Tabellen). APIs liefern JSON, Kubernetes erwartet YAML, Excel exportiert CSV. Wer in solchen gemischten Umgebungen arbeitet, braucht die Konvertierung zwischen diesen Formaten regelmässig.

2.2. JSON (JavaScript Object Notation)

JSON ist das am weitesten verbreitete Datenaustauschformat im Web. Es wurde von Douglas Crockford spezifiziert und ist seit den 2000er-Jahren der De-facto-Standard für REST-APIs.

Struktur und Syntax:

```
{
  "name": "Alice",
  "age": 30,
  "active": true,
  "address": {
    "city": "Zürich",
    "country": "CH"
  },
  "hobbies": ["coding", "hiking"]
}
```

Merkmale:
- Schlüssel-Wert-Paare mit geschweiften Klammern
- Datentypen: String, Number, Boolean, null, Object, Array
- Keine Kommentare möglich (Nachteil für Konfigurationsdateien)
- Strenge Syntax (Kommas, Anführungszeichen)

Typische Anwendungsfälle:
- REST-API Responses und Requests
- NoSQL-Datenbanken (MongoDB, CouchDB)
- Package-Management (package.json, composer.json)
- Datenaustausch zwischen Frontend und Backend

Beispiel (REST-API): `{"status": 200, "data": {"users": [{"id": 1, "name": "Alice"}]}}`

2.3. YAML (YAML Ain't Markup Language)

YAML ist ein menschenlesbares Datenformat, das häufig für Konfigurationsdateien verwendet wird. Es nutzt Einrückungen statt Klammern, was die Lesbarkeit verbessert, aber auch eine häufige Fehlerquelle darstellt.

Struktur und Syntax:

```yaml
name: Alice
age: 30
active: true
address:
  city: Zürich
  country: CH
hobbies:
  - coding
  - hiking
```

Merkmale:

- Einrückungs-basierte Struktur (Spaces, keine Tabs)
- Kommentare möglich (mit #)
- Unterstützt komplexe Datentypen (Anker, Referenzen)
- Kompatibler mit JSON (jedes gültige JSON ist auch gültiges YAML)

Typische Anwendungsfälle:

- CI/CD-Pipelines (GitHub Actions, GitLab CI)
- Container-Orchestrierung (Docker Compose, Kubernetes)
- Ansible Playbooks
- Konfigurationsdateien (Spring Boot, Jekyll)

2.4. TOML (Tom's Obvious Minimal Language)

TOML wurde von Tom Preston-Werner (Mitgründer von GitHub) entwickelt mit dem Ziel, ein menschenlesbares Konfigurationsformat zu schaffen, das eindeutig auf eine Hash-Tabelle abgebildet werden kann.

Struktur und Syntax:

```toml
[package]
name = "convrs"
version = "0.1.0"
edition = "2024"

[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
clap = { version = "4.5", features = ["derive"], optional = true }
```

Merkmale:

- Explizite Sektionen mit eckigen Klammern
- Kommentare möglich (mit #)
- Starke Typisierung (Datum, Zeit, Integer, Float, Boolean, String)
- **Limitation:** Keine Arrays als Root-Element erlaubt

Typische Anwendungsfälle:

- Rust Cargo.toml (Package-Management)
- Python pyproject.toml
- Hugo-Konfiguration
- Einfache Applikations-Konfigurationen

2.5. CSV (Comma-Separated Values)

CSV ist das einfachste der vier Formate und wird für tabellarische Daten verwendet. Jede Zeile entspricht einem Datensatz, Spalten werden durch Kommas getrennt.

Struktur und Syntax:

```
name,age,city,active
Alice,30,Zürich,true
Bob,25,Bern,false
Charlie,35,Basel,true
```

Merkmale:

- Flache, tabellarische Struktur (keine Verschachtelung)
- Einfach zu lesen und zu erzeugen
- Weit verbreitet in Excel, Datenbanken, Datenanalyse
- Keine Typisierung (alles ist ein String)
- Kein Standard für Escape-Regeln (Variationen möglich)

Typische Anwendungsfälle:

- Daten-Export aus Datenbanken
- Excel-Import/Export
- Datenanalyse (Pandas, R)
- Einfache Datenlisten (User-Exporte, Logfiles)

Wichtige Einschränkung: CSV eignet sich nur für flache, tabellarische Daten. Verschachtelte Strukturen (wie in JSON, YAML, TOML) müssen beim Konvertieren „geflattened“ werden.

2.6. Vergleichstabelle

Eigenschaft	JSON	YAML	TOML	CSV
Lesbarkeit	Mittel	Hoch	Hoch	Hoch
Kommentare	Nein	Ja	Ja	Nein
Verschachtelung	Ja	Ja	Ja	Nein
Datentypen	6	Viele	Streng	Keine
Parsing-Komplexität	Niedrig	Hoch	Niedrig	Sehr niedrig
Dateiendung	.json	.yaml / .yml	.toml	.csv
Haupt-Einsatz	APIs	Config/CI	Config	Tabellen

Zusammenfassung:

- **JSON:** Universell für Datenaustausch, besonders im Web
- **YAML:** Am besten lesbar für komplexe Konfigurationen, aber fehleranfällig durch Einrückung
- **TOML:** Explizit und eindeutig, ideal für einfache Konfigurationen
- **CSV:** Optimal für tabellarische Daten, keine Verschachtelung möglich

Wann welche Konvertierung?

1. **JSON → YAML:** APIs liefern JSON; Kubernetes, Docker Compose und CI/CD (GitHub Actions, GitLab) erwarten YAML. Konvertierung macht API-Daten für DevOps-Workflows nutzbar.
2. **JSON → TOML:** Wenn Konfiguration aus einer API (z.B. package.json) in Rust-Projekte (Cargo.toml) übernommen oder mit TOML-basierten Tools geteilt werden soll.
3. **JSON → CSV:** API- oder Datenbank-Export für Excel, Pandas, R oder andere Tabellen-Tools. CSV ist das universelle Format für Tabellenkalkulation und Statistik.
4. **YAML → JSON:** Konfiguration liegt in YAML (z.B. Ansible, Spring Boot); ein Tool oder eine API erwartet JSON. Umgekehrt: JSON-Config für Menschen lesbar als YAML speichern.
5. **TOML ↔ JSON:** Cargo.toml (Rust) vs. package.json (JavaScript) – Konfigurationen zwischen Ökosystemen austauschen oder migrieren.
6. **CSV → JSON:** Tabellenexport (z.B. aus Excel oder DB) für APIs, Frontends oder weitere Verarbeitung. CSV → YAML/TOML für config-artige Darstellung von Listen.

---

3. Das Konvertierungs-Tool: convrs

3.1. Projektübersicht

convrs ist ein in Rust geschriebenes Tool zur bidirektionalen Konvertierung zwischen JSON, YAML, TOML und CSV. Dank WebAssembly kann convrs auch im Browser genutzt werden, wobei die Konvertierung vollständig clientseitig im WASM-Modul stattfindet. Zusätzlich steht eine CLI-Version für Automation und Scripting zur Verfügung. Beide Versionen teilen sich dieselbe Rust-Codebasis. Für beide Varianten wird eine lokale Rust-Installation benötigt.

Unterstützte Konvertierungen:

Von / Nach	JSON	YAML	TOML	CSV
JSON	Ja	Ja	Ja	Ja
YAML	Ja	Ja	Ja	Ja
TOML	Ja	Ja	Ja	Ja
CSV	Ja	Ja	Ja	Ja

Alle 16 Konvertierungskombinationen (4 × 4) werden unterstützt, inklusive Format-zu-gleiches-Format (z.B. JSON → JSON). Diese „Selbst-Konvertierungen“ dienen zwei Zwecken:

1. **Pretty-Printing:** Minifizierter oder unformatierter Input wird sauber formatiert mit konsistenter Einrückung ausgegeben. Beispiel: `{"name":"Alice","age":30}` wird zu lesbarem, eingerücktem JSON.
2. **Validierung als Nebeneffekt:** Das Parsing ist der erste Schritt jeder Konvertierung. Wenn der Input ungültig ist, schlägt das Parsing fehl, die Output-Datei wird nie geschrieben und eine Fehlermeldung erscheint im Terminal. Es gibt keine separate Validierung vor der Konvertierung.

Zusätzlich sorgt die vollständige Abdeckung aller 4 × 4 Kombinationen dafür, dass der Rust-Compiler per Exhaustiveness-Checking sicherstellt, dass kein Fall im Code vergessen wird.

Features:

- Im Browser nutzbar: Web-Version via WebAssembly (lokal mit Trunk gebaut und geserved)
- Bidirektionale Konvertierung zwischen allen vier Formaten (16 Kombinationen)
- Benutzerauswahl via Dropdown (Web) oder automatische Format-Erkennung anhand der Dateiendung (CLI)
- Robuste Fehlerbehandlung mit aussagekräftigen Meldungen
- Pretty-Printing und Validierung für alle Formate
- Plattformübergreifend (Windows, macOS, Linux, Browser)
- CLI-Version für Automation und Scripting

3.2. Installation

3.2.1. Voraussetzungen

- **Rust** (mindestens Version 1.85.0, da das Projekt die Rust Edition 2024 verwendet)
- **Cargo** (wird automatisch mit Rust installiert)

Installation von Rust über das offizielle Installationsskript:

```
# Linux/macOS
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# Windows: Installer herunterladen von https://rustup.rs/
```

3.2.2. Methode 1: Globale Installation

Diese Methode kompiliert das Projekt und kopiert das fertige Binary automatisch nach `~/.cargo/bin/`, welches normalerweise bereits im PATH liegt. Danach kann `convrs` von überall im Terminal aufgerufen werden.

```
git clone https://github.com/Arda450/convrs.git
cd convrs
cargo install --path crates/convrs-cli

# Testen:
convrs --version
```

3.2.3. Methode 2: Lokale Installation (für Weiterentwicklung)

Diese Methode kompiliert das Projekt lokal, ohne das Binary global zu installieren. Nützlich, wenn man am Code weiterentwickeln möchte.

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

# Binary liegt unter target/release/convrs (bzw. convrs.exe auf Windows)
# Aufruf mit vollem Pfad:
./target/release/convrs convert -i input.json -o output.yaml

# Oder während der Entwicklung direkt mit cargo run:
cargo run -p convrs-cli -- convert -i input.json -o output.yaml
```

3.2.4. PATH-Konfiguration

Falls `convrs` nicht gefunden wird: Linux/macOS `export PATH="$HOME/.cargo/bin:$PATH"` zu Shell-Config hinzufügen; Windows: `~/.cargo/bin` prüfen.

3.3. Verwendung (CLI)

> **Tipp:** Wer convrs lieber visuell im Browser nutzen möchte, findet die Anleitung dafür im nächsten Abschnitt (Web-Version).

3.3.1. Grundlegende Syntax

```
convrs convert --input <input-datei> --output <output-datei>

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

Das Tool erkennt das Format automatisch anhand der Dateiendung. Unterstützte Endungen:

- `.json`
- `.yaml`
- `.yml`
- `.toml`
- `.csv`

3.3.2. 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

# YAML zu TOML
convrs convert -i docker-compose.yaml -o config.toml

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

3.3.3. Terminal-Output

Bei erfolgreicher Konvertierung:

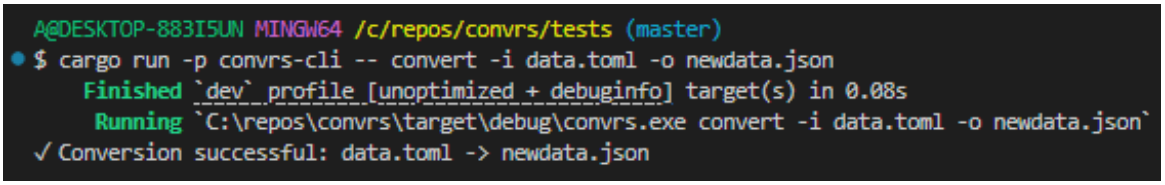

Abbildung 1: Terminal-Ausgabe: TOML zu JSON mit cargo run.

Bei einem Fehler:

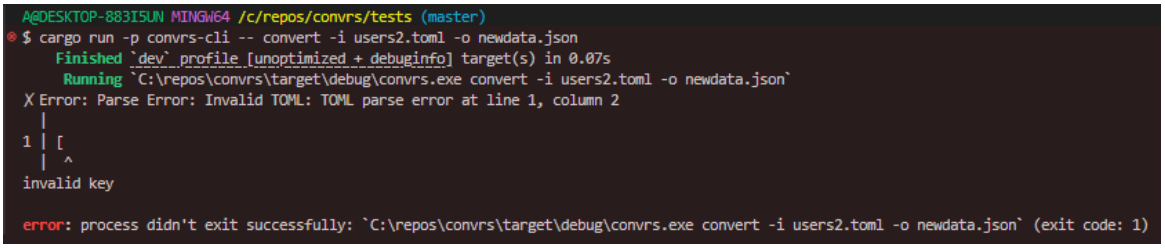

Abbildung 2: Terminal-Ausgabe bei Fehler: Ungültiges TOML, z.B. Root-Array ohne Tabellennamen.

3.4. Verwendung der Web-Version (WebAssembly)

Nach der einmaligen Einrichtung (Rust + Trunk) genügt `trunk serve`, um die Web-Version im Browser zu starten. Die Konvertierung läuft dann vollständig clientseitig im Browser.

Die Web-Version ermöglicht die Nutzung von convrs im Browser. Der Rust-Code wird lokal zu WebAssembly kompiliert und von Trunk als lokaler Development-Server bereitgestellt. Die Konvertierung findet dann vollständig im Browser statt, ohne serverseitige Verarbeitung. Gerade für Studierende in web-basierten Studiengängen bietet diese Variante einen schnellen Zugang zur Formatkonvertierung.

3.4.1. Technologie-Stack

- **Trunk:** Build-Tool für Rust WebAssembly (vergleichbar mit npm, Vite/Webpack für JavaScript)
- **wasm-bindgen:** Brücke zwischen Rust und JavaScript
- **web-sys:** Rust-Bindings für Browser-APIs (DOM, Events)

3.4.2. Server starten

```
# Trunk installieren
cargo install trunk

# Development-Server starten
trunk serve

# Oder mit automatischem Browser-Öffnen
trunk serve --open
```

Der Server läuft standardmässig auf `http://127.0.0.1:8080`.

3.4.3. Web-Interface

Das Web-Interface besteht aus drei Panels im CLI-artigen Matrix-Design:

1. **Input-Panel:** Textarea für die Eingabedaten mit Format-Dropdown (JSON, YAML, TOML, CSV)
2. **Output-Panel:** Textarea für das konvertierte Ergebnis mit Format-Dropdown
3. **Steuerung:** „RUN“-Button für die Konvertierung, „COPY“-Button für die Zwischenablage

Fehler- und Erfolgsmeldungen werden im Footer angezeigt.

Abbildung 3: convrs Web-Interface: Eingabe (JSON), Ausgabe (TOML) und RUN-Button zur Konvertierung.

3.4.4. Web vs. CLI im Vergleich

Aspekt	Web-Version	CLI-Version
Voraussetzungen	Rust + Trunk + WASM-Target	Rust + `cargo install --path crates/convrs-cli`
Starten	`trunk serve` → Browser öffnen	`convrs convert -i ... -o ...`
Technologie	WebAssembly (WASM)	Natives Binary
Input	Textarea / direkte Texteingabe	Datei-Pfade
Output	Textarea / Copy-to-Clipboard	Datei auf Festplatte
Use-Case	Demos, Lernen, Portfolio, visuell	Scripts, Automation, CI/CD
Zielgruppe	Web-Studierende, visuelle Nutzer	Entwickler mit Terminal-Erfahrung

---

4. Rust-Grundlagen für CLI-Tools

4.1. Warum Rust?

Rust wurde für dieses Projekt gewählt, weil es mehrere Vorteile vereint, die für ein plattformübergreifendes Tool (Web + CLI) besonders relevant sind:

- **Erstklassiger WebAssembly-Support:** Rust ist eine der am besten unterstützten Sprachen für WebAssembly. Der gesamte Code kann zu WASM kompiliert werden und läuft damit ohne Plugins direkt im Browser, mit nahezu nativer Performance.
- **Memory-Safety ohne Garbage Collector:** Keine Null-Pointer-Exceptions, keine Data Races, keine Use-after-Free-Bugs. Alle Speicherfehler werden zur Compile-Zeit erkannt.
- **Performance auf C/C++ Niveau:** Kompiliert zu nativem Maschinencode (CLI) oder hochoptimiertem WASM (Web), ohne Runtime-Overhead.
- **Cargo Package Manager:** Integriertes Build-System mit Dependency-Management, Testing und Dokumentation.
- **Starkes Typ-System:** Enums, Pattern Matching und Traits ermöglichen ausdrucksstarke, sichere Abstraktionen.
- **Fehlerbehandlung mit `Result<T, E>`:** Explizites Error-Handling ohne Exceptions.

4.2. Cargo: Der Rust Package Manager

Cargo ist das zentrale Werkzeug für Rust-Entwicklung. Es verwaltet Dependencies, kompiliert Code und führt Tests aus.

4.2.1. Cargo.toml: Dependencies verwalten

Die `Cargo.toml` ist das Herzstück der Projektkonfiguration:

```
[package]
name = "convrs"
version = "0.1.0"
edition = "2024"

[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
serde_yaml = "0.9"
toml = "0.8"
csv = "1.3"
clap = { version = "4.5", features = ["derive"], optional = true }
```

4.2.2. Wichtige Build-Befehle

Befehl	Beschreibung
`cargo build -p convrs-cli`	Kompiliert die CLI-Crate (Debug-Modus)
`cargo build -p convrs-cli --release`	Optimierter Build für Produktion
`cargo run -p convrs-cli -- ...`	Kompiliert und führt die CLI aus
`cargo test`	Führt alle Tests des Workspaces aus
`cargo fmt`	Formatiert den Code
`cargo clippy`	Statische Analyse (Linting)

4.3. Rust-Konzepte im Projekt

Die convrs-Architektur nutzt zentrale Rust-Konzepte: **Borrowing** (z.B. `input: &str` in den Konvertierungsfunktionen), **Pattern Matching** für die 16 Format-Kombinationen im FileFormat-Enum, **Enums mit Daten** für FormatError (jede Variante trägt eine Fehlermeldung) und **Result** mit dem `?`-Operator für Fehlerbehandlung. Für vertiefte Rust-Grundlagen verweist das E-Learning auf die weiterführenden Ressourcen (Rust Book).

4.4. Externe Crates (Libraries)

Das convrs-Projekt nutzt folgende externe Crates:

Crate	Beschreibung
serde	Universelles Framework für Serialisierung/Deserialisierung. `#[derive(Serialize, Deserialize)]` generiert automatisch den nötigen Code.
serde_json	JSON-Parser und -Serializer. `serde_json::Value` dient als universelle Intermediate Representation.
serde_yaml	YAML-Parser und -Serializer. Unterstützt YAML 1.2.
toml	TOML-Parser und -Serializer. `toml::Value` für TOML-spezifische Datentypen.
csv	CSV-Reader und -Writer. Header-basiertes Lesen und Schreiben.
clap	CLI-Argument-Parsing mit Derive-Makros. Generiert automatisch Hilfe-Texte.
wasm-bindgen	Brücke zwischen Rust und JavaScript für WebAssembly.
web-sys	Rust-Bindings für Browser-APIs (DOM-Manipulation, Events).

---

5. Architektur und Implementierung

5.1. Projektstruktur

Das convrs-Projekt ist ein Cargo-Workspace mit drei Crates: **convrs-core** (shared Library), **convrs-cli** (CLI-Binary) und **convrs-web** (WASM-Binary). CLI und Web nutzen beide convrs-core; die Trennung erfolgt über separate Crates statt Features. Die folgende Struktur zeigt die vollständige Verzeichnis- und Modulorganisation:

```
convrs/
├── crates/
│   ├── convrs-cli/          # CLI Binary (Clap, File-I/O)
│   │   ├── src/
│   │   │   ├── main.rs      # CLI-Parser (Clap), ruft lib auf
│   │   │   └── lib.rs       # convert_file, Unit-Tests
│   │   ├── tests/
│   │   │   └── cli.rs       # Integrationstests (Binary)
│   │   └── Cargo.toml
│   ├── convrs-core/         # Shared Library (ohne I/O)
│   │   ├── src/
│   │   │   ├── lib.rs
│   │   │   ├── format.rs    # FileFormat Enum, convert()
│   │   │   ├── error.rs    # FormatError
│   │   │   └── formats/
│   │   │       ├── mod.rs
│   │   │       ├── json.rs, yaml.rs, toml.rs, csv.rs
│   │   │       └── utils.rs
│   │   ├── tests/
│   │   │   └── conversion.rs # Integrationstests (API)
│   │   └── Cargo.toml
│   └── convrs-web/          # WebAssembly Binary
│       ├── src/main.rs      # WASM, DOM, perform_conversion
│       └── Cargo.toml
├── index.html               # Web-Version Entry (Trunk)
├── Trunk.toml               # Trunk-Konfiguration
├── Cargo.toml               # Workspace-Root
├── Cargo.lock               # Exakte Versionen (auto-generiert)
├── target/                  # Build-Output
└── README.md                # Projekt-Dokumentation
```

Jedes Format hat ein eigenes Modul (`json.rs`, `yaml.rs`, `toml.rs`, `csv.rs`), was die Wartbarkeit und Erweiterbarkeit verbessert. Die gemeinsame Logik liegt in `utils.rs`. Unit-Tests stehen in den jeweiligen Modulen (`#[cfg(test)]`); Integrationstests liegen in den `tests/`-Verzeichnissen (convrs-cli: Binary-Tests über `Command`; convrs-core: API-Tests über `FileFormat::convert`).

5.2. Das FileFormat Enum: Kern der Architektur

Das `FileFormat` Enum ist das zentrale Element der Architektur. Es repräsentiert die vier unterstützten Formate und stellt eine `convert()`-Methode bereit:

```
#[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::Json) => json_to_json_string(input),
            (FileFormat::Json, FileFormat::Yaml) => json_to_yaml_string(input),
            (FileFormat::Json, FileFormat::Toml) => json_to_toml_string(input),
            (FileFormat::Json, FileFormat::Csv)  => json_to_csv_string(input),
            // ... 12 weitere Kombinationen
        }
    }
}
```

Design-Entscheidung: Enum statt String-Matching

Enum-Ansatz (gewählt)	String-Ansatz (Alternative)
Type-Safety zur Compile-Zeit	Fehler erst zur Runtime
Exhaustiveness-Checking	Vergessene Fälle möglich
Autocompletion in IDE	Tippfehler möglich
Refactoring-sicher	Fragile String-Vergleiche

5.3. Konvertierungsarchitektur: Zweischichtige Struktur

Die Architektur trennt klar zwischen Core-Logik (String-zu-String, ohne I/O) und plattformspezifischem I/O. WebAssembly hat keinen Dateisystem-Zugriff; daher muss die Konvertierungslogik I/O-frei sein. Dieselbe Core-Logik wird von CLI und Web genutzt.

Übersicht:

- **convrs-core:** Enthält 16 String-zu-String-Funktionen und die einheitliche API `FileFormat::convert()`. Kein Dateisystem, kein DOM.
- **convrs-cli:** `convert_file()` liest Dateien, ruft `FileFormat::convert()` auf, schreibt die Ausgabe in eine Datei. Fügt **Datei-I/O** hinzu.
- **convrs-web:** `perform_conversion()` nimmt Text aus der Textarea, ruft `FileFormat::convert()` auf, setzt das Ergebnis in die Ausgabe-Textarea. Nutzt die String-API direkt; Ein- und Ausgabe kommen aus dem DOM, nicht aus Dateien. Es gibt keine separate Datei-I/O-Schicht.

5.3.1. Schicht 1: String-zu-String (convrs-core)

16 Funktionen in `formats/json.rs`, `yaml.rs`, `toml.rs`, `csv.rs`: reine Konvertierungslogik, Signatur `input: &str -> Result<String, FormatError>`. Sie werden über `FileFormat::convert()` aufgerufen:

```
// FileFormat::convert() in format.rs dispatched zu:
json_to_json_string(), json_to_yaml_string(), json_to_toml_string(), json_to_csv_string()
yaml_to_json_string(), yaml_to_yaml_string(), yaml_to_toml_string(), yaml_to_csv_string()
toml_to_json_string(), toml_to_yaml_string(), toml_to_toml_string(), toml_to_csv_string()
csv_to_json_string(), csv_to_yaml_string(), csv_to_toml_string(), csv_to_csv_string()
```

Beispiel: json_to_yaml_string (eine der 16 Funktionen)

```
pub fn json_to_yaml_string(input: &str) -> Result<String, FormatError> {
    let json_value: serde_json::Value = serde_json::from_str(input)
        .map_err(|e| FormatError::ParseError(...))?;
    serde_yaml::to_string(&json_value)
        .map_err(|e| FormatError::SerializationError(...))
}
```

5.3.2. Schicht 2a: File-I/O (nur CLI)

Die CLI fügt eine Datei-I/O-Schicht hinzu. `convert_file()` in `convrs-cli/src/lib.rs`:

```
pub fn convert_file(input_path: &str, output_path: &str) -> Result<(), FormatError> {
    let input_format  = FileFormat::from_str(Path::new(input_path).extension()?)?;
    let output_format = FileFormat::from_str(Path::new(output_path).extension()?)?;
    let content = fs::read_to_string(input_path)?;      // Datei lesen
    let result  = input_format.convert(&content, output_format)?;  // Core!
    fs::write(output_path, result)?;                   // Datei schreiben
    Ok(())
}
```

5.3.3. Schicht 2b: DOM-Ein-/Ausgabe (nur Web)

Die Web-Version hat **keine** Datei-I/O. `perform_conversion()` in `convrs-web/src/main.rs` nutzt die String-API direkt:

```
fn perform_conversion(input_text: &str, input_format: &str, output_format: &str)
    -> Result<String, String> {
    let input_fmt  = FileFormat::from_str(input_format)?;
    let output_fmt = FileFormat::from_str(output_format)?;
    input_fmt.convert(input_text, output_fmt)  // Direkt Core, kein fs::
        .map_err(|e| e.to_string())
}
// Aufruf im Event-Handler: Text kommt aus textarea.value(), Ergebnis geht in
output_textarea.set_value()
```

Warum diese Trennung?

- **Web-kompatibel:** Die Core-Funktionen sind I/O-frei; WebAssembly hat kein `fs::`.
- **Wiederverwendung:** `FileFormat::convert()` wird von CLI (mit Dateien) und Web (mit DOM) genutzt.
- **Separation of Concerns:** Core-Logik ist unabhängig von Dateien oder DOM.
- **Ein Bugfix, zwei Plattformen:** Korrekturen in convrs-core wirken in CLI und Web.

5.4. Intermediate Representation (IR)

Die String-Funktionen nutzen intern Serdes Value-Typen als Zwischenrepräsentation. Es gibt keine eigene, zentrale IR-Datenstruktur, sondern `serde_json::Value` dient als pragmatische Brücke zwischen den Formaten. Jede der 16 Konvertierungsfunktionen ist explizit implementiert, nutzt aber intern diesen gemeinsamen Weg:

```
// JSON als Quelle: direkt über serde_json::Value (1 Stufe)
JSON String → serde_json::Value → YAML/TOML/CSV String

// YAML als Quelle: über format-eigene Value + JSON-Value als Brücke (2 Stufen)
YAML String → serde_yaml::Value → serde_json::Value → JSON/TOML/CSV String

// TOML als Quelle: gleicher Weg über JSON-Value als Brücke
TOML String → toml::Value → serde_json::Value → JSON/YAML/CSV String

// Spezialfall JSON → TOML: manuelle Konvertierung nötig
JSON String → serde_json::Value → json_to_toml_value() → toml::Value → TOML String
```

Die manuelle Funktion `json_to_toml_value()` in `utils.rs` ist nötig, da Serde keine direkte Konvertierung zwischen `serde_json::Value` und `toml::Value` unterstützt. Sie konvertiert rekursiv alle JSON-Typen (Null, Bool, Number, String, Array, Object) in ihre TOML-Äquivalente.

Vorteile dieser pragmatischen IR-Nutzung:

- **Weniger Redundanz:** Jede Funktion nutzt intern denselben Weg über `serde_json::Value`
- **Testbarkeit:** String-Funktionen ohne Dateisystem testbar
- **Konsistenz:** `serde_json::Value` ist gut dokumentiert, flexibel und unterstützt alle gängigen Datentypen
- **Pragmatisch:** Nutzt bestehende Serde-Typen, anstatt eine eigene IR zu definieren

5.5. Fehlerbehandlung

Das Projekt verwendet einen zentralen `FormatError`-Enum für alle Fehlerarten:

```
#[derive(Debug)]
pub enum FormatError {
    IoError(String),             // Datei lesen/schreiben fehlgeschlagen
    ParseError(String),          // Ungültige Syntax im Input
    SerializationError(String),  // Konvertierung fehlgeschlagen
    InvalidFormat(String),       // Unbekanntes Format
    UnknownError(String),        // Fallback für unerwartete Fehler
}

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 Error: {}", msg),
            FormatError::ParseError(msg) => write!(f, "Parse Error: {}", msg),
            // ...
        }
    }
}

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

5.6. CLI-Integration mit Clap

Clap ermöglicht deklaratives CLI-Design mit Derive-Makros (`#[derive(Parser)]`, `Commands::Convert { input, output }`). Die Format-Erkennung erfolgt über die Dateiendung: `Path::new(input_path).extension()` → `FileFormat::from_str(input_ext)`.

5.7. Web-Version: Rust zu WebAssembly

Die Web-Version nutzt denselben Core-Code wie die CLI, kompiliert ihn jedoch zu WebAssembly.

Ablauf:

1. Rust-Code wird mit Trunk zu WASM kompiliert
2. `wasm-bindgen` erstellt JavaScript-Bindings
3. `web-sys` ermöglicht DOM-Zugriff aus Rust
4. Der Browser führt den WASM-Code aus

Zentraler Code (vereinfacht):

```
#[wasm_bindgen(start)]
pub fn start() -> Result<(), JsValue> {
    // HTML-Elemente holen
    let input_textarea = document
        .get_element_by_id("input")
        .dyn_into::<HtmlTextAreaElement>()?;

    // Event-Handler für Convert-Button
    let convert_closure = Closure::wrap(Box::new(move || {
        let input_text = input_textarea.value();
        let input_fmt = FileFormat::from_str(&input_format_val)?;
        let output_fmt = FileFormat::from_str(&output_format_val)?;

        // Dieselbe convert()-Funktion wie in der CLI!
        let result = input_fmt.convert(&input_text, output_fmt);

        match result {
            Ok(output) => output_textarea.set_value(&output),
            Err(e) => status.set_inner_html(&format!("error: {}", e)),
        }
    }) as Box<dyn FnMut()>);

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

Entscheidender Vorteil: Die Zeile `input_fmt.convert(&input_text, output_fmt)` ist identisch in CLI und Web. Die gesamte Konvertierungslogik wird geteilt, nur die I/O-Schicht unterscheidet sich. Dieses Architektur-Pattern („Shared Core“) ist ein bewährter Ansatz in der modernen Cross-Platform-Entwicklung.

---

6. Herausforderungen und Lösungen

6.1. TOML-Limitationen: Keine Root-Arrays

Problem: TOML erlaubt keine Arrays als Root-Element. Ein JSON-Array wie `[{...}, {...}]` kann nicht direkt in TOML dargestellt werden.

```
# Nicht erlaubt in TOML:
[[users]]        # Fehler: Root-Level Array
name = "Alice"

# Lösung: Wrapper-Objekt
[[data]]
name = "Alice"
age = 30

[[data]]
name = "Bob"
age = 25
```

Implementierung: JSON-Arrays werden automatisch in ein `data`-Wrapper-Objekt gepackt. Zusätzlich wurde `json_to_toml_value()` implementiert, da Serde keine direkte Konvertierung `serde_json::Value` → `toml::Value` bietet.

TOML-Datetime: TOML hat einen nativen Datetime-Typ, JSON nicht. Der `toml`-Crate verpackt Datetimes in ein internes Wrapper-Objekt (`$__toml_private_datetime`) bei der JSON-Konvertierung. Dies ist eine bekannte Limitation, kein Fehler.

6.2. CSV-Konvertierung: Flache vs. hierarchische Strukturen

Problem: CSV ist tabellarisch und flach, während JSON, YAML und TOML hierarchische Verschachtelungen unterstützen.

Lösung: Flattening mit Unterstrichen

```
// Input (JSON):
{ "user": { "name": "Alice", "address": { "city": "Zürich" } } }

// Output (CSV) nach Flattening:
user_name,user_address_city
Alice,Zürich
```

Die `flatten_json()`-Funktion geht rekursiv durch alle verschachtelten Objekte und verbindet die Schlüssel mit Unterstrichen. Unterstriche wurden statt Punkten gewählt, da Punkte in CSV-Headern problematisch sein können.

Wichtige Einschränkung: CSV eignet sich am besten für Arrays von gleichartigen Objekten (z.B. eine User-Liste). Einzelne stark verschachtelte Objekte (z.B. eine package.json) erzeugen beim Flattening zu viele Spalten und inkonsistente Strukturen.

---

7. Hands-On Übungen

7.1. Übung 1: Installation und erste Konvertierung

Ziel: Die Web-Version von convrs starten und eine erste JSON-zu-YAML-Konvertierung im Browser durchführen.

Schritte:

1. Installiere Rust von rustup.rs und Trunk (`cargo install trunk`)
2. Klone das Repository und starte den Server:

```
git clone https://github.com/Arda450/convrs.git
cd convrs
trunk serve --open
```

3. Der Browser öffnet sich auf `http://127.0.0.1:8080`. Im **Input-Panel** gib folgendes JSON ein (Input-Format: JSON, Output-Format: YAML):

```
{
  "name": "Test",
  "version": "1.0",
  "features": ["convert", "pretty-print"],
  "config": {
    "debug": false,
    "max_retries": 3
  }
}
```

4. Klicke auf den **RUN**-Button. Im Output-Panel erscheint das YAML-Ergebnis.
5. Prüfe die Ausgabe. Erwartet wird z.B.:

```
name: Test
version: '1.0'
features:
- convert
- pretty-print
config:
  debug: false
  max_retries: 3
```

6. **Optional:** Kopiere die YAML-Ausgabe mit [copy], füge sie ins Input-Panel ein, wähle Input-Format YAML und Output-Format TOML, klicke RUN – so siehst du die Konvertierung YAML → TOML.

7.2. Übung 2: Globale Installation und Format-Kette

Ziel: Das Tool global installieren und eine Kette von Konvertierungen durchführen.

Schritte:

1. Installiere convrs global (im Projektverzeichnis convrs, z.B. nach Übung 1 oder nach `git clone` und `cd convrs`):

```
cargo install --path crates/convrs-cli
```

2. Teste die Installation:

```
convrs --version
```

3. Erstelle eine CSV-Datei `users.csv`:

```
name,age,city,active
Alice,30,Zürich,true
Bob,25,Bern,false
Charlie,35,Basel,true
```

4. Konvertiere durch alle Formate:

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

# JSON → YAML
convrs convert -i users.json -o users.yaml

# YAML → TOML
convrs convert -i users.yaml -o users.toml

# TOML → CSV (Round-Trip)
convrs convert -i users.toml -o users_roundtrip.csv
```

5. Vergleiche `users.csv` mit `users_roundtrip.csv`.

7.3. Übung 3: Fehlerbehandlung testen

Ziel: Verstehen, wie convrs mit ungültigen Eingaben umgeht.

Schritte:

1. Erstelle eine Datei `broken.json` mit ungültigem JSON:

```
{
  "name": "Test",
  "age": 25,
  "active": true,
  "value":
}
```

2. Versuche die Konvertierung:

```
convrs convert -i broken.json -o output.yaml
```

3. Beobachte die Fehlermeldung. Was sagt sie aus?
4. Versuche eine Datei mit falscher Endung: Benenne eine YAML-Datei in `.json` um und konvertiere sie.

---

8. Weiterführende Ressourcen

8.1. Offizielle Dokumentationen

- **Rust Book:** https://doc.rust-lang.org/book/ – Kapitel 9 (Error Handling), 12 (minigrep)
- **Serde:** https://serde.rs/ – Value-Typen, Derive-Macros
- **Clap:** https://docs.rs/clap/ – CLI-Argument-Parsing
- **Rust WebAssembly Book:** https://rustwasm.github.io/docs/book/ – wasm-bindgen, web-sys

8.2. Format-Spezifikationen

- **JSON:** https://www.json.org/json-en.html – **YAML:** https://yaml.org/spec/1.2.2/ – **TOML:** https://toml.io/en/v1.0.0

8.3. Empfohlene Lernressourcen

- **Rust CLI Book:** https://rust-cli.github.io/book/ – **Rust by Example:** https://doc.rust-lang.org/rust-by-example/ – **W3C WebAssembly:** https://www.w3.org/TR/wasm-core-2/

---

9. Quellenverzeichnis

(Zitierweise: Cite Them Right Harvard, Anglia Ruskin University)

Clap-rs (2024) *clap: Command Line Argument Parser for Rust*. Verfügbar unter: https://docs.rs/clap/latest/clap/ (Zugriff am 27. Oktober 2025).

Crockford, D. (2023) *JSON*. Verfügbar unter: https://www.json.org/json-en.html (Zugriff am 10. Oktober 2025).

Karadavut, A. (2025) *Arbeitsblatt Lernziel des Advanced Specialised Projects (ASP): Datenkonvertierung mit CLI*. SAE Institute Zürich.

Klabnik, S. and Nichols, C. (2025) *The Rust Programming Language*. Verfügbar unter: https://doc.rust-lang.org/book/ (Zugriff am 3. Oktober 2025).

Preston-Werner, T. (2013) *TOML: Tom's Obvious Minimal Language*. Verfügbar unter: https://toml.io/en/ (Zugriff am 15. Oktober 2025).

Rust and WebAssembly (2024) *The Rust and WebAssembly Book*. Verfügbar unter: https://rustwasm.github.io/docs/book/ (Zugriff am 15. Oktober 2025).

The Rust CLI Working Group (2025) *Command Line Applications in Rust*. Verfügbar unter: https://rust-cli.github.io/book/ (Zugriff am 21. November 2025).

Tryzelaar, E. and Tolnay, D. (2014) *serde: Rust Serialization Framework*. Verfügbar unter: https://docs.rs/serde/latest/serde/ (Zugriff am 17. November 2025).

W3C (2025) *WebAssembly Core Specification*. Verfügbar unter: https://www.w3.org/TR/wasm-core-2/ (Zugriff am 29. Dezember 2025).

YAML.org (2011) *The Official YAML Web Site*. Verfügbar unter: https://yaml.org/ (Zugriff am 5. Oktober 2025).

9.1. Quellenkommentare (Auswahl)

Die folgenden vier Quellen waren für das convrs-Projekt und dieses E-Learning besonders hilfreich. Zweck und Nutzen für die Zielgruppe werden kurz erläutert.

Klabnik, S. and Nichols, C. (2025) – The Rust Programming Language

Das offizielle Rust Book erklärt Ownership, Borrowing, Traits und Error Handling systematisch. Für Dritte ist es die zentrale Referenz, um die convrs-Architektur nachzuvollziehen und eigene CLI-Tools in Rust zu entwickeln. Kapitel 12 (minigrep) diente als Vorlage für File I/O und strukturiertes Error Handling im Projekt.

Tryzelaar, E. and Tolnay, D. (2014) – serde

Serde ist das zentrale Framework für Serialisierung und Deserialisierung in convrs. Die Dokumentation erläutert die universellen Value-Typen (serde_json::Value), die als Intermediate Representation zwischen den Formaten dienen. Für die Zielgruppe erklärt sie, wie Parsing und Serialisierung ohne eigene Datenstrukturen funktionieren.

The Rust CLI Working Group (2025) – Command Line Applications in Rust

Dieses Buch vermittelt Best Practices für CLI-Tools in Rust. Es unterstützt die Zielgruppe bei der Einordnung von convrs in gängige CLI-Patterns (Subcommands, Argument-Parsing, Fehlerausgabe) und bei der Entwicklung eigener Tools.

Rust WebAssembly Book (via rustwasm.github.io)

Die Anleitung für Rust und WebAssembly erklärt wasm-bindgen, web-sys und den Build-Prozess mit Trunk. Für Web-Studierende ist sie der Schlüssel zum Verständnis, wie convrs im Browser läuft und warum die String-zu-String-Architektur WebAssembly-kompatibel ist.