Projekt-Dokumentation

FMD Flashcard

Vault-basierte Lern-App

Version:	0.1.0
Status:	Entwurf
Datum:	Dezember 2025
Autor:	Marcel Tenhaft
Repository:	https://<dein-repo-link>
Build:	Commit: `<hash>` Branch: `main`
Kontakt:	`<mail>` / `<discord/github>`

Dieses Dokument beschreibt Anforderungen, Architektur, Implementierung und Betrieb des Projekts FMD Flashcard.

Änderungshistorie

Version	Datum	Autor	Änderung
0.1.0	Dezember 2025	Marcel Tenhaft	Initiale Projektstruktur, Präambel, Kapitel-Imports

Inhaltsverzeichnis

Abbildungen

Tabellen

Abkürzungsverzeichnis

Abk.	Deutsch	Englisch / Kommentar
A/B-Test	Vergleichstest	Split-Run-Test zweier Varianten
IT	Informationstechnologie	Information Technology

Abbildung 1: Projektlogo FMD Flashcard (logo).Eigendarstellung, 2026

1 Einleitung

Diese Arbeit dokumentiert die Konzeption und Umsetzung des Projekts FMD Flashcard, einer Vault-basierten Lern- und Flashcard-Anwendung. Der Schwerpunkt liegt auf der technischen Projektdokumentation (Architektur, Implementierung, Build-/Run-Prozess, Tests und Betrieb), sodass das System nachvollziehbar reproduziert, bewertet und weiterentwickelt werden kann.

Die Dokumentation dient als zentrale Referenz für Entscheidungen und Vorgehensweisen im Projektverlauf. Sie reduziert Einarbeitungszeit, erleichtert Reviews und schafft eine belastbare Grundlage für Wartung, Erweiterungen und spätere Refactorings.

Als Ergebnis entsteht eine strukturierte Projektbeschreibung mit klaren Anforderungen, einem konsistenten Architekturmodell, einem nachvollziehbaren Entwicklungsprozess sowie konkreten Anleitungen für Setup, Nutzung und Betrieb.

1.1 Motivation

Digitale Lerninhalte verteilen sich häufig über Notizen, PDFs, Karteikarten-Apps und verschiedene Geräte. Dadurch entstehen Medienbrüche, redundante Inhalte und ein hoher Pflegeaufwand. Insbesondere beim langfristigen Lernen ist es hilfreich, wenn Wissen strukturiert, versionierbar und wiederverwendbar vorliegt.

Das Projekt adressiert dieses Problem durch eine Vault-basierte Organisation der Inhalte (analog zu wissensbasierten Notizsystemen) und verbindet diese mit einer Flashcard-Logik. Ziel ist eine Lösung, die Inhalte konsistent verwaltet, den Lernfortschritt abbildet und gleichzeitig eine einfache Erweiterbarkeit für spätere Funktionen (z. B. Synchronisation, Import/Export, Statistiken) ermöglicht.

1.2 Ziel und Fragestellung

Ziel des Projekts ist die Entwicklung eines lauffähigen Prototyps einer Lernanwendung, die Lerninhalte in einer klar definierten Datenstruktur (Vault) verwaltet und daraus Flashcards für wiederholtes Lernen ableitet.

Die leitende Fragestellung lautet: „Wie kann eine Vault-basierte Lernanwendung so konzipiert und implementiert werden, dass Inhalte reproduzierbar verwaltet, effizient gelernt und technisch wartbar weiterentwickelt werden können?“

1.3 Beitrag dieses Papers

Dieses Dokument liefert die für das Projekt wesentlichen Artefakte und Entscheidungen in strukturierter Form:

- eine nachvollziehbare Beschreibung der Anforderungen und Zielkriterien,
- eine konzeptionelle Architektur (Datenmodell, Komponenten, Schnittstellen),
- eine strukturierte Darstellung der Entwicklungsphasen von den Grundlagen bis zum Prototyp,
- konkrete Hinweise zu Setup, Build/Run, Konfiguration und Projektstruktur,
- eine Zusammenfassung zentraler Entscheidungen, Risiken sowie offener Punkte.

2 Installation & Entwicklungsumgebung

Dieses Kapitel beschreibt die notwendigen Voraussetzungen sowie die empfohlene Toolchain, um FMD Flashcardlokal zu bauen und auszuführen. Der Schwerpunkt liegt auf einer reproduzierbaren Entwicklungsumgebung und einem klaren Setup-Prozess. Da zentrale Installations- und Diagnoseaufgaben über Skripte automatisiert werden, wird **Python** als erste Voraussetzung behandelt.

2.1 Voraussetzungen

Für die Entwicklung werden folgende Rahmenbedingungen empfohlen:

- **Betriebssystem:** Linux (primär getestet unter Arch Linux), Windows/macOS optional.
- **Shell/Terminal:** Keine feste Shell-Vorgabe. Die Projektsteuerung erfolgt über plattformübergreifende **Python-Skripte**; eine interaktive Shell wird lediglich zum Aufruf der Befehle benötigt.
- **Zugriffsrechte:** Installation von Paketen/Toolchains (je nach System via Paketmanager).
- **Versionsverwaltung:** Git.

2.1.1 Layer-Modell der Entwicklungsumgebung

Zur besseren Einordnung der eingesetzten Werkzeuge lässt sich die Entwicklungsumgebung in funktionale Layer gliedern. Diese Layer beschreiben nicht die Programmarchitektur, sondern die Umgebung, in der Entwicklung, Build und Ausführung stattfinden.

- **Basissystem (System Layer)**
  Betriebssystem und grundlegende Systemdienste (Linux, macOS, Windows) bilden die technische Basis. In diesem Layer befinden sich außerdem systemweite Werkzeuge wie Git, Compiler, Paketmanager und Laufzeitumgebungen.
- **Automatisierungs- und Skript-Layer**
  Installations-, Diagnose- und Steuerungslogik ist plattformübergreifend in **Python** implementiert. Dieser Layer kapselt wiederkehrende Aufgaben wie Setup, Dependency-Prüfung, Build und Start des Projekts und abstrahiert Plattformunterschiede.
- **Tool- und Framework-Layer**
  Entwicklungswerkzeuge und Frameworks wie Rust, Node.js, Tauri sowie der verwendete Paketmanager stellen die eigentliche Build- und Laufzeitumgebung für Backend und Frontend bereit.
- **Entwicklungs- und Prozess-Layer**
  IDEs (z. B. VS Code), Editor-Erweiterungen, Linter, Formatter und Dev-Server unterstützen den täglichen Entwicklungsprozess. Dieser Layer beeinflusst die Developer Experience, jedoch nicht das resultierende Programmverhalten.

Dieses Layer-Modell dient der strukturellen Einordnung der Entwicklungsumgebung und schafft eine klare Trennung zwischen Systembasis, Automatisierung, Toolchain und entwicklungsbegleitenden Prozessen.

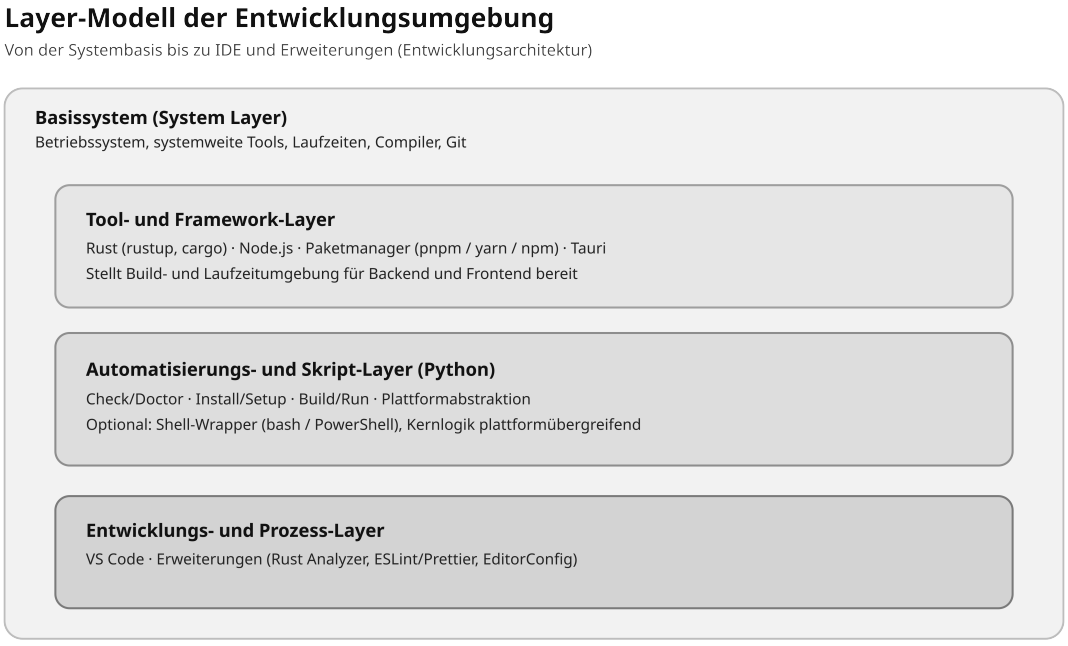

Abbildung 2: Layer-Modell FMD Flashcard .Eigendarstellung, 2026

2.2 Voraussetzung: Python

Python ist eine weit verbreitete, plattformübergreifende Programmiersprache, die häufig für Automatisierung, Systemadministration und Tooling eingesetzt wird. In diesem Projekt wird Python primär als **administrative Unterstützung** genutzt: Installations- und Setup-Schritte werden über Skripte standardisiert, und das Checkup-/Diagnose-Skript verwendet Python, um Systemzustand, Abhängigkeiten und Toolchain konsistent zu prüfen.Python Software Foundation, 2025[1]

Warum Python zuerst?

- Installationsskripte und Checks können damit auf **Windows, Linux und macOS** einheitlich ausgeführt werden.
- Python eignet sich für robuste Systemabfragen (z. B. Pfade, Versionen, verfügbare Tools) und reduziert manuelle Fehler.
- Das Projekt nutzt Python nicht als Laufzeitabhängigkeit der Anwendung selbst, sondern als **Tooling-Schicht** rund um Setup und Wartung.

Hinweis zur Vorinstallation: Auf vielen Linux-Distributionen ist `python3` in typischen Desktop-Installationen bereits vorhanden (z. B. Ubuntu, Fedora Workstation, openSUSE Leap). Das ist jedoch nicht garantiert: Bei Minimal-Images oder sehr schlanken Installationen kann Python fehlen (z. B. bei einer reinen Arch-`base`-Installation). Auf macOS wird Python nicht zuverlässig mitgeliefert und sollte daher explizit installiert werden. Für eine reproduzierbare Umgebung wird in jedem Fall empfohlen, die verwendete Python-Version zu prüfen und zu dokumentieren.

Beispiel Installationsbefehle:

Python installieren (Beispiele)

```
#  Arch Linux (Details vollstaendiges Setup: siehe Anhang A)
sudo pacman -S python python-pip

# Fedora / RPM-basiert (DNF)
sudo dnf install -y python3 python3-pip

# Ubuntu/Debian
sudo apt update
sudo apt install python3 python3-pip

# macOS (Homebrew)
brew install python

# Windows (Winget)
winget install -e --id Python.Python.3
```

[1] https://www.python.org/

2.2.1 Prüfen der Installation

Nach der Installation sollte die Python-Version überprüft werden. Je nach System ist Python entweder über `python` oder `python3` erreichbar.

Python-Version prüfen

```
python3 --version
# alternativ (falls passend):
python --version
```

2.3 Voraussetzung: Git

Git ist ein verteiltes Versionsverwaltungssystem, das Änderungen am Quellcode nachvollziehbar speichert und Zusammenarbeit über Branches, Commits und Tags ermöglicht. In diesem Projekt wird Git benötigt, um das Repository zu klonen, Updates einzuspielen und Stände reproduzierbar zu referenzieren (z. B. über Tags/Commits).

Warum Git?

- **Reproduzierbarkeit:** definierte Stände über Tags/Commits
- **Nachvollziehbarkeit:** Historie von Änderungen und Entscheidungen
- **Zusammenarbeit:** Branching/Merging für parallele Entwicklung

Hinweis zur Vorinstallation: Ob Git bereits vorinstalliert ist, hängt vom Betriebssystem und der jeweiligen Installation (Desktop vs. Minimal-Image) ab. Auf macOS ist Git häufig über die Xcode Command Line Tools verfügbar bzw. wird beim ersten Aufruf nachinstalliert. Auf einigen Linux-Images kann Git bereits vorhanden sein (z. B. wird es bei Fedora Silverblue häufig mitgeliefert), bei Minimalinstallationen ist es jedoch nicht garantiert.

Praxisbeispiel: In der Testumgebung (Arch Linux / CachyOS) waren Git und Python bereits vorinstalliert; dies kann je nach Distribution und Installationsprofil abweichen.

Beispiel Installationsbefehle:

Git installieren (Beispiele)

```
# Arch Linux
sudo pacman -S git

# Fedora / RPM-basiert
sudo dnf install git-all

# Ubuntu/Debian
sudo apt update
sudo apt install git
```

```
# macOS (Apple/Xcode CLT oder Homebrew)
xcode-select --install
# alternativ:
brew install git

# Windows (Winget)
winget install --id Git.Git -e --source winget
```

2.3.1 Prüfen der Installation

Nach der Installation sollte Git verfügbar sein:

Git-Version prüfen

```
git --version
```

2.4 Quickstart

Die folgenden Schritte zeigen den schnellsten Weg, um FMD Flashcardlokal zu starten. Das Setup wird über das Control-/Checkup-Tooling standardisiert (z. B. `doctor`, `install`, `run`).

Quickstart (Beispiel)

```
# git clone git clone https://github.com/octocat/Hello-World.git
git clone <REPO-URL>
cd <PROJEKT-ORDNER>

# optional: Health-Check / Doctor
 cd /<PFAD-SYSTEM-PROJEKT-ORDNER>/FMDFlashcard/tools
./control.py --doctor



# Dependencies installieren / Build vorbereiten
./control.sh install

# Projekt starten (Dev)
./control.sh run
```

Hinweis: Das Beispiel oben zeigt die Befehle für ein Linux-System; unter macOS sind `git clone` und `cd` identisch. Unter Windows funktionieren die Befehle in „Git Bash“ wie unter Linux; in PowerShell/CMD ebenfalls, nur das Auflisten des Verzeichnisses erfolgt typischerweise mit `dir` (PowerShell unterstützt auch `ls`).

2.5 Toolchain und Frameworks

Tabelle 1 fasst die eingesetzten Werkzeuge zusammen. Versionen sind als Mindestempfehlung zu verstehen und können projektabhängig angepasst werden (z. B. via `.tool-versions`, `rust-toolchain.toml` oder `package.json`).

Tool/Framework	Version	Zweck im Projekt
Git	>= 2.x	Repository klonen, Branching, Versionsverwaltung
VS Code	aktuell	IDE/Editor; empfohlen für konsistente Formatierung und Debugging
Rust (rustup, cargo)	>= 1.7x	Backend/Build (abhängig vom Projektanteil in Rust)
Node.js	>= 18 LTS	Frontend/Tooling (Build, Dev-Server, Bundling)
Paketmanager (pnpm/yarn/npm)	projektspezifisch	Abhängigkeiten installieren, Scripts ausführen
Control-Skript (`control.sh`)	repo-intern	Standardisierte Befehle: Check, Install, Build, Run

Tabelle 1: Toolchain-Übersicht

2.5.1 Empfohlene VS-Code-Erweiterungen

Für eine konsistente Developer Experience werden folgende Erweiterungen empfohlen (optional):

- **Rust Analyzer** (Rust-IDE-Features)
- **EditorConfig** (einheitliche Formatierung)
- **ESLint** / **Prettier** (bei JavaScript/TypeScript-Frontend)

2.6 Setup-Schritte

Dieser Abschnitt beschreibt die grundlegenden Setup-Schritte unabhängig vom Betriebssystem. OS-spezifische Installationsbefehle sind im Anhang dokumentiert.

2.6.1 Repository beziehen

Repository klonen

```
git clone <REPO-URL>
cd <PROJEKT-ORDNER>
```

2.6.2 Abhängigkeiten installieren

Wenn das Projekt ein Control-Skript bereitstellt, sollte dieses bevorzugt genutzt werden, da es wiederholbare Abläufe kapselt.

Installation via Control-Skript

```
./control.sh doctor
./control.sh install
```

Alternativ können (je nach Projektstruktur) die Abhängigkeiten direkt über den jeweiligen Paketmanager bzw. Cargo installiert werden:

Installation ohne Control-Skript (Beispiel)

```
# Frontend
pnpm install

# Rust-Anteile (falls erforderlich)
cargo fetch
```

2.6.3 Projekt starten (Entwicklung)

Start (Dev)

```
./control.sh run
```

2.7 Arch Linux: OS-spezifische Installation (Anhang)

Die vollständige Installationsanleitung für Arch Linux inklusive systemabhängiger Pakete und dem vollständigen Setup-Skript ist im Anhang dokumentiert:

- **Anhang A:** Installationsskript und Paketliste für Arch Linux

Für weitere Betriebssysteme (z. B. Ubuntu/Debian, Fedora, Windows, macOS) kann die Anleitung analog ergänzt werden. Dabei ist insbesondere auf systemabhängige Bibliotheken und Build-Tools zu achten (Compiler, Linker, ggf. UI-Framework-Abhängigkeiten).

3 Diskussion und Ausblick

3.1 Limitationen des Ansatzes

3.2 Zukünftige Arbeiten

4 Fazit

Literaturverzeichnis

Eigendarstellung. (2026). Abbildung/Entwurf (eigene Erstellung) [Die Abbildung wurde vom Autor für diese Arbeit erstellt.].

Python Software Foundation. (2025). *Python.org: The official home of the python programming language* [Offizielle Projektwebseite]. Verfügbar 17. Dezember 2025 unter https://www.python.org/

Verzeichnis der Anhänge