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

Änderungshistorie **I**

Abkürzungsverzeichnis **IV**

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziel und Fragestellung . . . 1
1.3 Beitrag dieses Papers . . . 2

2 Installation & Entwicklungsumgebung **2**
2.1 Voraussetzungen . . . 2
2.1.1 Layer-Modell der Entwicklungsumgebung . . . 2
2.2 Voraussetzung: Python . . . 4
2.2.1 Prüfen der Installation . . . 5
2.3 Voraussetzung: Git . . . 5
2.3.1 Prüfen der Installation . . . 6
2.4 Quickstart . . . 6
2.4.1 Terminal Checkup . . . 7
2.5 Erläuterung der Checkup-Bereiche . . . 8
2.6 Visual Studio Code (Installation via –`vscode`) . . . 8
2.7 Tauri (Prüfung, Abhängigkeiten und Initialisierung via –`tauri`) . . . 9
2.8 Tauri Dev-Start (Ausführen via –`start`/–`run`) . . . 10

3 Diskussion und Ausblick **13**
3.1 Limitationen des Ansatzes . . . 13
3.2 Zukünftige Arbeiten . . . 13

4 Fazit **13**

Literaturverzeichnis **14**

Verzeichnis der Anhänge **15**

A Beispielanhänge **15**

Abbildungen

1 Projektlogo FMD Flashcard (logo).Eigendarstellung, 2026 1
2 Layer-Modell FMD Flashcard .Eigendarstellung, 2026 3
3 Terminal Checkup.Eigendarstellung, 2026 7
4 DesktopTauri.Eigendarstellung, 2026 12

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).
- **Shell/Terminal:** Keine feste Shell-Vorgabe. Die Projektsteuerung erfolgt über **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.

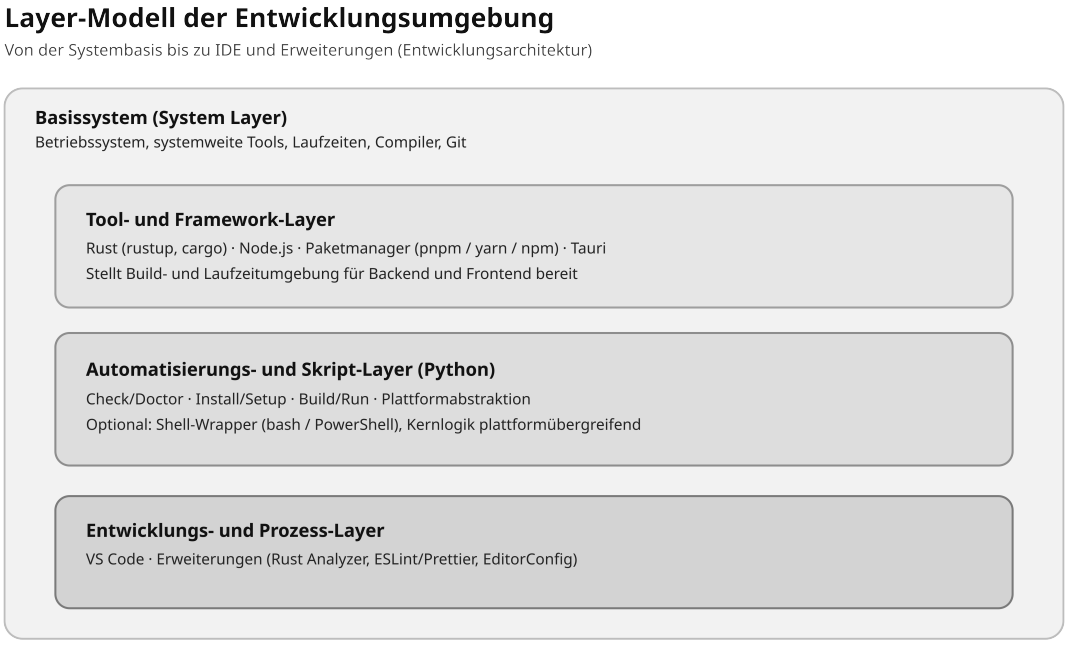

Abbildung 2: Layer-Modell FMD Flashcard .Eigendarstellung, 2026

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).GitHub, 2025[2]

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
```

[2] https://git-scm.com/

```
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: Clone the repo & switch to a standard project directory

```
# Standard project directory (works on Linux systems):
mkdir -p ~/Projects
cd ~/Projects

# Clone repository (replace URL)
git clone https://github.com/kleiveist/FMDFlashcard.git
cd FMDFlashcard

# Control-Skript
cd ~/Projects/FMDFlashcard
# optional: health check / doctor
python3 tools/control.py --doctor

# Install & start
cd ~/Projects/FMDFlashcard
python3 tools/control.py --install
```

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`).

Nach dem Ausführen von `python3 tools/control.py –doctor` wird eine Auflistung der noch fehlenden Pakete und Module angezeigt. Diese werden anschließend automatisch mit folgendem Befehl installiert:

```
python3 tools/control.py --install
```

Nach der Installation wird `python3 tools/control.py –doctor` erneut ausgeführt. Das Ergebnis ist in Abbildung 4 dargestellt.

Abbildung 3: Terminal Checkup.Eigendarstellung, 2026

2.4.1 Terminal Checkup

In den folgenden Unterabschnitten werden die einzelnen Bereiche der Terminal-Überprüfung erläutert. Ziel ist es, die angezeigten Komponenten kurz einzuordnen und zu begründen, warum diese Prüfungen durchgeführt werden. Dadurch wird sichergestellt, dass alle notwendigen Werkzeuge

und Systemabhängigkeiten für Installation, Build und Ausführung der Anwendung vorhanden und korrekt konfiguriert sind.

2.5 Erläuterung der Checkup-Bereiche

Die folgenden Punkte erklären die im Terminal-Checkup ausgegebenen Bereiche und deren Zweck.

- **Shell (in den meisten Systemen bereits installiert):** Zeigt die verwendete Shell sowie die PATH-Konfiguration. Damit wird geprüft, ob wichtige Programme über die Kommandozeile gefunden werden und die Entwicklungsumgebung korrekt eingerichtet ist.
- **Core Tools (in den meisten Systemen bereits installiert):** Enthält grundlegende Entwicklungswerkzeuge (z. B. `git`, Compiler, Build-Tools). Diese werden benötigt, um Quellcode zu beziehen, native Abhängigkeiten zu kompilieren und Build-Prozesse zuverlässig auszuführen.
- **Rust:** Prüft die Rust-Toolchain (`rustup`, `rustc`, `cargo`). Dies ist erforderlich, da das Tauri-Backend in Rust gebaut wird und ohne eine passende Toolchain keine Kompilierung und kein Packaging möglich ist.
- **Node:** Überprüft Node.js sowie Paketmanager wie `npm`/`pnpm`. Diese werden für das Frontend benötigt, um JavaScript-Abhängigkeiten zu installieren und das Web-Bundle für die Tauri-App zu erstellen.
- **Tauri System Libs:** Listet systemweite Bibliotheken auf, die unter Linux für WebView und GUI-Funktionalitäten notwendig sind (z. B. `gtk3`, `webkit2gtk`, `openssl`). Dadurch wird sichergestellt, dass die Anwendung sowohl gebaut als auch zur Laufzeit korrekt ausgeführt werden kann.

Hinweis zu Windows und macOS Neben der Linux/Unix-Variante existieren auch Installationsskripte für Windows und macOS, die über die automatische Betriebssystem-Erkennung in `control.py` referenziert werden (u. a. `installwin.py` und `installmac.py`). :contentReference[oaicite:0]index=0 Diese Skripte sind derzeit jedoch noch nicht in virtuellen Maschinen (VMs) verifiziert bzw. reproduzierbar erprobt worden und gelten deshalb als *ungetestet*.

2.6 Visual Studio Code (Installation via –vscode)

Als Code-Editor wird Visual Studio Code (VS Code) empfohlen, da er plattformübergreifend auf allen gängigen Betriebssystemen verfügbar ist und eine große Auswahl an Erweiterungen bietet. Für dieses Projekt sind insbesondere die Python-Unterstützung (z. B. Linting, Debugging) sowie KI-gestützte Erweiterungen (z. B. Codex-Integration) als Produktivitätswerkzeuge relevant. Weitere Erweiterungen für die Frontend-Entwicklung (CSS/JavaScript), z. B. Formatierung, Linting und Framework-spezifische Hilfen, werden in späteren Kapiteln ergänzt.

Aufruf Die Installation kann automatisiert über das Projekt-Tooling angestoßen werden:

```
python3 tools/control.py --vscode
```

Der Schalter –vscode ist in control.py hinterlegt und lädt das passende Installationsmodul installuixvs (Linux) und führt dessen run_install( ) aus. :contentReference[oaicite:0]index=0

Funktionsweise des Installationsskripts Das Installationsskript ist als „Unified Installer“ für Linux/Unix umgesetzt und unterstützt Arch-basierte Systeme sowie Debian/Ubuntu. Es prüft zunächst, ob VS Code bereits vorhanden ist (Binary code); in diesem Fall wird keine Installation durchgeführ

- **Arch/Derivate:** Standardmäßig wird – sofern ein AUR-Helper (paru oder yay) vorhanden ist – der Microsoft-Build (visual-studio-code-bin) aus dem AUR installiert. Falls kein AUR-Helper gefunden wird, erfolgt ein Fallback auf code (Code - OSS) via pacman. Optional kann über VSCODE_VARIANT=oss erzwungen werden, dass immer die OSS-Variante installiert wird. Zusätzlich verhindert das Skript bewusst eine AUR-Installation als root, um typische Arch-Konventionen einzuhalten. :contentReference[oaicite:2]index=2
- **Debian/Ubuntu:** Das Skript stellt sicher, dass curl verfügbar ist, ermittelt die Systemarchitektur (z. B. x64/arm64/armhf), lädt das aktuelle stabile .deb direkt von der offiziellen Update-Quelle von VS Code herunter und installiert es anschließend per apt. Dabei wird im nicht-interaktiven Modus gearbeitet, um eine saubere Automatisierung zu ermöglichen.

Nutzen für neue VMs Gerade auf frischen oder neu provisionierten VMs ist dieser Ansatz hilfreich, weil die Installation reproduzierbar und ohne manuelle Zwischenschritte erfolgt (inklusive OS-Erkennung, Paketmanager-Pfad und ggf. sudo-Nutzung). Dadurch steht die Entwicklungsumgebung schnell für nachfolgende Schritte (Python-, CSS- und JS-Workflow) bereit.

2.7 Tauri (Prüfung, Abhängigkeiten und Initialisierung via –tauri)

Der Befehl python3 tools/control.py –tauri dient dazu, eine Linux-Umgebung für die Entwicklung einer Tauri-Anwendung vorzubereiten. Dabei werden die erforderlichen Systembibliotheken geprüft und (falls nötig) installiert. Zusätzlich wird ein lauffähiges Tauri-Projektgerüst (React + TypeScript, pnpm) erstellt und die JavaScript-Abhängigkeiten werden installiert.

Aufruf

```
python3 tools/control.py --tauri
```

Optional kann mit –dry-run zunächst nur angezeigt werden, welche Installationsbefehle ausgeführt würden.

Was passiert beim Ausführen? Beim Start lädt `control.py` unter Linux das Modul `installuixtauri` und ruft dessen `run_install(...)` auf (unter Windows/macOS wird der Vorgang abgebrochen, da die Routine Linux-only ist).

- **Sicherheits-/Ausführungsmodus:** Das Skript muss als normaler Benutzer laufen (nicht als `root`); für Systempakete wird gezielt `sudo` verwendet.
- **Distro-Erkennung:** Erkennung von Debian/Ubuntu vs. Arch (über `/etc/os-release` und Fallbacks).
- **Systemabhängigkeiten (Linux):** Es wird geprüft, ob zentrale Build-Tools und `pkg-config`-Einträge (u. a. `webkit2gtk-4.1`, `openssl`, `librsvg-2.0`) vorhanden sind. Wenn nicht, werden passende Pakete automatisch installiert (via `apt` oder `pacman`).
- **pnpm bereitstellen:** Falls `pnpm` fehlt, wird bevorzugt `corepack` verwendet (`corepack enable` und `corepack prepare pnpm@latest –activate`); alternativ erfolgt ein Fallback über `npm i -g pnpm` (ggf. mit `sudo`).
- **Rust sicherstellen:** Falls `rustc`/`cargo` fehlen, wird über `rustup` die stabile Toolchain installiert und als Standard gesetzt.
- **Projektgerüst erstellen:** Anschließend wird nicht-interaktiv ein neues Tauri-Projekt mittels `pnpm create tauri-app` mit dem Template `react-ts` erzeugt (Standardziel: `apps/fmd-desktop`). Danach folgt `pnpm install`.

Wann wird `–tauri` eingesetzt und wofür? `–tauri` wird eingesetzt, wenn eine neue Entwicklungsumgebung (z. B. frische VM) für Tauri vorbereitet werden soll oder wenn typische Tauri-Buildfehler auf fehlende Systembibliotheken bzw. Toolchains hindeuten. Der Abschnitt automatisiert insbesondere die Linux-spezifischen Voraussetzungen (WebView/GTK/WebKit, Build-Tools) und reduziert manuelle Installationsschritte, bevor anschließend die eigentliche Anwendung entwickelt bzw. gebaut wird.

2.8 Tauri Dev-Start (Ausführen via `–start`/`–run`)

Der Befehl `python3 tools/control.py –start` (Alias: `–run`) startet die Desktop-App im Entwicklungsmodus. Dabei wird in das Tauri-Projektverzeichnis gewechselt und `pnpm tauri dev` ausgeführt. Falls JavaScript-Abhängigkeiten noch nicht installiert sind, wird vorher automatisch `pnpm install` ausgeführt. :contentReference[oaicite:0]index=0 :contentReference[oaicite:1]index=1

Aufruf

```
python3 tools/control.py --start
# oder identisch:
python3 tools/control.py --run
```

Optional kann mit `–dry-run` nur angezeigt werden, welche Befehle ausgeführt würden (ohne Ausführung). :contentReference[oaicite:2]index=2 :contentReference[oaicite:3]index=3

Was passiert beim Ausführen? Beim `--start/--run` lädt `control.py` das Modul `run` (entspricht `tools/inst/run.py`) und ruft dessen `run_install(dry_run=...)` auf. :contentReference[oaicite:4]index=4 :contentReference[oaicite:5]index=5

- **Kontext/Projektpfade:** Ermittelt werden *Repo-Root* und das Zielverzeichnis `apps/fmd-desktop`. Falls nur ein Legacy-Pfad existiert (`tools/apps/fmd-desktop`), wird darauf hingewiesen und dieser Pfad verwendet. :contentReference[oaicite:6]index=6
- **Vorbedingungen prüfen:**
  - **Zielverzeichnis vorhanden?** Wenn `apps/fmd-desktop` nicht existiert, wird mit Hinweis abgebrochen, dass zuerst `--tauri` auszuführen ist. :contentReference[oaicite:7]index=7
  - **pnpm vorhanden?** Wenn `pnpm` nicht im `PATH` gefunden wird, bricht das Skript ab und verweist auf `--tauri`. :contentReference[oaicite:8]index=8
  - **Rust-Toolchain nutzbar?** Es wird geprüft, ob `rustc --version` und `cargo --version` erfolgreich laufen; sonst Abbruch mit Hinweis auf `--tauri`. :contentReference[oaicite:9]index=9
- **JavaScript-Abhängigkeiten:** Existiert `node_modules` im Zielverzeichnis noch nicht, wird `pnpm install` ausgeführt. Ist `node_modules` vorhanden, wird die Installation übersprungen. :contentReference[oaicite:10]index=10
- **Start des Dev-Modus:** Anschließend wird `pnpm tauri dev` im Zielverzeichnis ausgeführt. Dieser Prozess läuft dauerhaft weiter, bis er beendet wird. :contentReference[oaicite:11]index=11
- **Beenden (Ctrl+C mit Bestätigung):** Bei interaktivem Terminal fängt das Skript `Ctrl+C` ab und fragt nach `Beenden? (j/n)`. Bei `j` wird ein `SIGINT` an den gestarteten Prozess (Prozessgruppe) gesendet. :contentReference[oaicite:12]index=12
- **Headless-Fallback (Linux):** Wenn kein `DISPLAY` bzw. `WAYLAND_DISPLAY` gesetzt ist und der Dev-Start fehlschlägt, versucht das Skript automatisch über `xvfb-run` erneut zu starten. Falls `xvfb-run` fehlt, wird es (wenn möglich) über `apt-get` oder `pacman` nachinstalliert. :contentReference[oaicite:13]index=13

Warum sieht man beim ersten Start häufig Downloads und Compiling? `pnpm tauri dev` startet typischerweise zwei Teilprozesse: (1) den Frontend-Dev-Server (z. B. Vite) und (2) den Rust/Tauri-Dev-Build via `cargo run`. Beim ersten Lauf (oder nach Cache-Löschung) lädt `cargo` benötigte Rust-Abhängigkeiten und kompiliert sie in das Projekt-`target/`-Verzeichnis; spätere Starts sind meist deutlich schneller, weil bereits gebaute Artefakte wiederverwendet werden.

Wann wird `--start/--run` eingesetzt und wofür? `--start/--run` ist der Standard-Workflow für die tägliche Entwicklung: Nach erfolgreichem Setup (`--tauri`) wird der Dev-Modus gestartet, um UI und Backend gemeinsam auszuführen und Änderungen iterativ zu testen.

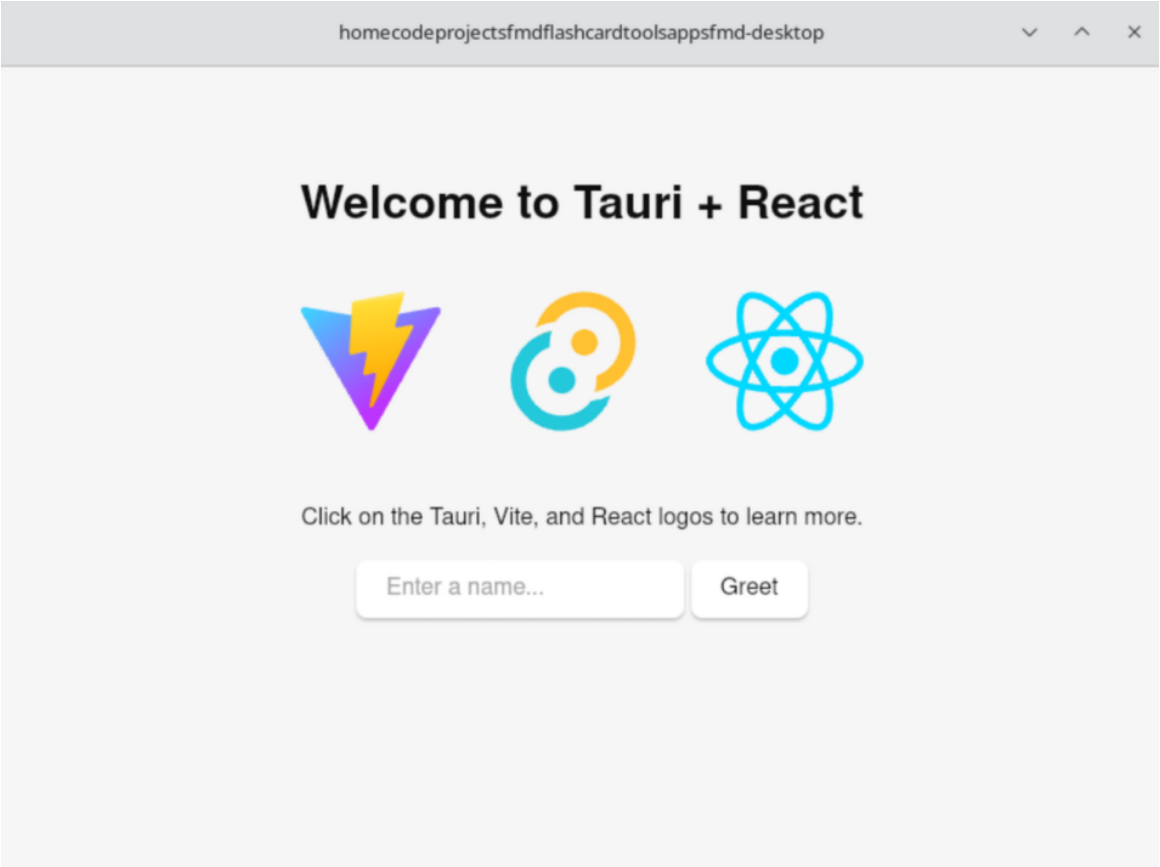

Abbildung 4: DesktopTauri.Eigendarstellung, 2026

Ergebnis nach dem Start Nach erfolgreicher Initialisierung und dem Aufruf von `python3 tools/control.py –start` erscheint ein Desktop-Fenster der Tauri-Anwendung. Im Fenster ist die Standard-Startseite des Templates sichtbar: die Überschrift „Welcome to Tauri + React“, darunter die Logos von Vite, Tauri und React sowie ein kurzer Hinweistext. Im unteren Bereich befinden sich ein Eingabefeld („Enter a name…“) und ein Button („Greet“). Diese Ansicht bestätigt, dass (a) der Frontend-Dev-Server läuft und (b) die Tauri-Runtime die WebView korrekt startet und die UI lädt.

Hinweis: Dateisystem (Festplattenformat) und `node_modules` unter Linux

Wichtig: Die JavaScript-Abhängigkeiten (`node_modules`) werden bei `pnpm` intern über ein Store-/Link-System aufgebaut, das unter Linux typischerweise **Symlinks** (und teils Hardlinks) verwendet. Wenn das Projekt auf einem Datenträger liegt, dessen Dateisystem diese POSIX-Features nicht unterstützt, schlägt `pnpm install` fehl (z. B. mit `ERR_PNPM_EPERM` und Meldungen zu `symlink` / `operation not permitted`). In der Folge kann auch `pnpm tauri dev` scheitern (z. B. `tauri: Kommando nicht gefunden`), weil die lokalen Binaries in `node_modules/.bin` nicht korrekt angelegt wurden.

Typische Problemfälle unter Linux:

- **exFAT/FAT (häufig bei externen SSDs/USB-Sticks):** unterstützt keine POSIX-Symlinks → `pnpm install` bricht ab.

- **NTFS (je nach Mount/Optionen):** Symlinks können eingeschränkt sein → ebenfalls mögliche `EPERM`-Fehler.
- **Netzwerk-/FUSE-Mounts (z. B. Samba, GVFS, spezielle Verschlüsselungs-/Cloud-Mounts):** Symlinks/Dateirechte sind je nach Setup limitiert.

Empfehlung: Für Tauri/Node-Entwicklung das Repo (oder zumindest `apps/fmd-desktop/node_modules`) auf einem nativen Linux-Dateisystem ablegen, z. B. **ext4** oder **btrfs**. Dadurch funktionieren Symlinks zuverlässig und `pnpm install` kann korrekt ausführen.

Wenn es schief geht (Quick-Fix):

```
rm -rf apps/fmd-desktop/node_modules
pnpm install
```

Falls das Repo auf einem nicht geeigneten Dateisystem liegen muss, kann alternativ `node_modules` auf ein ext4/btrfs-Verzeichnis ausgelagert und per Bind-Mount eingebunden werden.

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.].

GitHub. (2025). *Github: Let's build from here* [Offizielle Projektwebseite]. Verfügbar 26. Dezember 2025 unter https://github.com/about

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

A Beispielanhänge