NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

File Explorer

Ein Projekt für den Informaticup

Autoren:
Brandt Marco (Matrikel-Nr. 13779)
Lauritz Wiebusch (Matrikel-Nr. 13771)
Sören Panten(Matrikel-Nr. 13766)
Schatz Daniel (Matrikel-Nr. 13796)

Projekt: Informaticup
Abgabedatum: keine Ahnung

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung von Software für den eigenen Gebrauch – ausgestattet mit exakt jenen Funktionen, die man sich über Jahre hinweg im Alltag gewünscht hat – stellt nicht nur eine kreative, sondern auch eine methodisch interessante Herangehensweise dar. Insbesondere im Bereich der Dateiverwaltung, der tagtäglich durch den Einsatz verschiedenster File Explorer geprägt ist, zeigt sich ein eklatantes Missverhältnis zwischen praktischer Nutzung und funktionaler Erfüllung. Bestehende Systeme sind oftmals primär darauf ausgerichtet, Dateien darzustellen und ein minimales Interface bereitzustellen; tiefergehende Anforderungen moderner Nutzerinnen und Nutzer bleiben hingegen häufig unbeachtet.

Die Möglichkeit, einen File Explorer vollständig neu zu denken, eröffnet daher nicht nur den Freiraum zur Integration lang ersehnter Funktionen, sondern stellt zugleich ein erkenntnisreiches Experiment im Bereich der Mensch-Computer-Interaktion dar. Welche Strukturen, Navigationsprinzipien und Automatisierungsansätze begünstigen tatsächliche Effizienz? Wie lässt sich ein Interface gestalten, das nicht nur funktioniert, sondern die Nutzerin bzw. den Nutzer aktiv unterstützt, informiert und entlastet? Eine solche Neuentwicklung ist nicht bloß als technischer Fortschritt zu verstehen, sondern als reflektierter Beitrag zur Weiterentwicklung alltäglicher Softwarewerkzeuge.

Auf das zugrunde liegende Projekt wurden wir durch ein YouTube-Video aufmerksam [1], in dem der Autor seine Idee eines besonders schnellen File Explorers erläutert.

Wir haben dieses Projekt gezielt ausgewählt, da wir mit dem Ersteller persönlich Kontakt aufgenommen haben und dieser sich über unser Interesse erfreut zeigte. Bereits bei der ersten Auseinandersetzung mit dem Konzept kamen unserem Team konkrete Erweiterungsideen, da auch wir bestimmte Funktionen in einem Datei-Explorer vermissen. Zudem erläutert der Autor in einem weiteren Video detaillierte Schwierigkeiten im Zusammenhang mit der Suchfunktion [2], die wir als spannende Herausforderung empfanden.

Das Projekt basiert auf dem Framework Tauri [5], das es ermöglicht, Anwendungen in zwei Module zu unterteilen: einerseits das User Interface, welches mit den gängigsten Frontend-Frameworks entwickelt werden kann [3], andererseits die Verarbeitungsschicht, die in Rust [6] implementiert wird. Bei der Übernahme des Projekts war das Frontend noch nicht vollständig umgesetzt, da der ursprüngliche Fokus auf der Funktionalität lag und weniger auf einer ansprechenden Gestaltung. In der Rücksprache mit dem Projektinitiator bestätigte dieser, dass ein ästhetisch überzeugenderes Frontend auch in seinem Interesse liege. Aus diesem Grund entschieden wir uns für die Verwendung von React zur Gestaltung der Benutzeroberfläche.

Durch den Einsatz von Rust für die Verarbeitungskomponenten profitieren wir von den bekannten Vorteilen dieser Programmiersprache: hoher Laufzeitgeschwindigkeit und effiziente Ressourcennutzung – vorausgesetzt, die Programmierung erfolgt entsprechend sorgfältig.

2 Daten vom Backend zum Frontend

Die Datenverarbeitung findet hauptsächlich im Backend statt, das im Framework Tauri [5] durch in Rust geschriebenen Code realisiert wird. Das Frontend kann Methoden aufrufen, und über die asynchrone Runtime können Daten an das Frontend übermittelt werden. Dies ist relativ unkompliziert über JSON oder auch mit primitiven Datentypen möglich. Wenn jedoch benutzerdefinierte Objekte übergeben werden sollen, müssen die Felder im Frontend dieselben Namen tragen und identisch strukturiert sein. Für dieses Projekt haben wir uns entschieden, sämtliche Daten ausschließlich über JSON bereitzustellen. Diese einheitliche Lösung verhindert Probleme im Zusammenhang mit unterschiedlichen Datentypen und fehleranfälligen Konvertierungen.

Die Methoden, die vom Frontend aus aufgerufen werden können, werden mit dem Attribut `#[tauri::command]` markiert und dem Application-Builder hinzugefügt. Jeder Endpoint muss explizit registriert werden. Danach können die Methoden über ihren Namen im Frontend aufgerufen werden. Wichtig ist dabei, dass alle Methoden standardmäßig *asynchron* sind.

```
#[tauri::command]
pub async fn open_file(path: &str) -> Result<String, String> {
let path_obj = Path::new(path);
// Check if path exists
if !path_obj.exists() {
return Err(format!(//TODO));
}
// Check if path is a file
if !path_obj.is_file() {
return Err(format!(//TODO));
}
// Read the file
fs::read_to_string(path).map_err(|err| format!("Failed to read file: {}", err))
}
```

Eine weitere Möglichkeit, um Daten an das Frontend zu übermitteln, bietet der sogenannte *State* [4]. Der Vorteil dieser Methode liegt in der zentralisierten Datenspeicherung für das Frontend, welche über die gesamte Laufzeit der Applikation hinweg bestehen bleibt. Der State kann jederzeit vom Frontend abgefragt werden. Ein Beispiel für die Verwendung des State ist im Anhang zu finden 4.1.

Im File Explorer wird unter anderem folgende Datenstruktur über einen solchen State bereitgestellt:

```
pub struct MetaData {
version: String,
abs_file_path_buf: PathBuf,
abs_file_path_for_settings_json: PathBuf,
pub abs_folder_path_buf_for_templates: PathBuf,
pub template_paths: Vec<PathBuf>,
all_volumes_with_information: Vec<VolumeInformation>,
current_running_os: String,
current_cpu_architecture: String,
user_home_dir: String
}
```

Das oben gezeigte `struct` beinhaltet grundlegende Informationen, die entweder über die Applikation selbst gespeichert oder aus dem System, auf dem das Programm läuft, ausgelesen werden. Es wird beim Start der Anwendung initialisiert und mithilfe von Tauri dem Frontend zur Verfügung gestellt.

Da das `struct` unter anderem den Typ `PathBuf` verwendet, ergeben sich bei der Umsetzung in `TypeScript` gewisse Einschränkungen. Diese Typen lassen sich dort nicht ohne Weiteres abbilden. Ein möglicher Ansatz zur Abbildung besteht im Einsatz einer äquivalenten Struktur, wobei jedoch zu beachten ist, dass diese nicht automatisch generiert werden kann.

3 Performanceoptimierung

Performance-Optimierung ist ein zentraler Bestandteil der Softwareentwicklung, insbesondere bei Anwendungen, die regelmäßig mit großen Datenmengen oder komplexen Verarbeitungsprozessen arbeiten. Ziel ist es, Ressourcen wie CPU-Zeit, Arbeitsspeicher und I/O effizient zu nutzen, um die Reaktionsgeschwindigkeit und Gesamtleistung der Anwendung zu verbessern. Eine performante Anwendung trägt nicht nur zu einem besseren Nutzererlebnis bei, sondern reduziert auch Energieverbrauch und Hardwareanforderungen.

Ein nützliches Werkzeug zur Performance-Analyse sind sogenannte *Flamegraphs*. Diese visualisieren die Aufrufstruktur eines Programms und zeigen an, wie viel Zeit in welchen Funktionen verbracht wird. Die Breite eines Balkens in einem Flamegraph steht dabei für die Häufigkeit bzw. Dauer der Ausführung einer Funktion innerhalb eines Profiling-Zeitraums – je breiter der Balken, desto höher der Ressourcenverbrauch an dieser Stelle.

Flamegraphs helfen, sogenannte Hotspots im Code zu identifizieren – also Funktionen oder Abschnitte, die besonders viel Rechenzeit beanspruchen. Durch gezielte Analyse dieser Bereiche können Engpässe erkannt und durch algorithmische Verbesserungen, Caching oder Parallelisierung gezielt behoben werden. Besonders hilfreich ist dabei die hierarchische Darstellung: Sie zeigt, welche Funktionen von welchen anderen aufgerufen wurden, was eine Ursachenanalyse erheblich erleichtert.

4 Anhang

4.1 State Beispiel

Rust Code:

```
use tauri::State;
struct MyState(String);
#[tauri::command]
fn get_value(state: State<'_, MyState>) -> String {
    state.0.clone()
}
fn main() {
    tauri::Builder::default()
        .manage(MyState("Hello from backend".to_string())) // Initial value
        .invoke_handler(tauri::generate_handler![get_value])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Frontend in Typescript

```
'use client';
import { useEffect, useState } from 'react';
import { invoke } from '@tauri-apps/api/tauri';
export default function GetStateExample() {
  const [value, setValue] = useState<string>('');
  useEffect(() => {
    async function fetchValue() {
      const result = await invoke<string>('get_value');
      setValue(result);
    }
    fetchValue();
  }, []);
  return (
    <div>
      <h1>Backend State</h1>
      <p>{value}</p>
    </div>
  );
}
```

Literaturverzeichnis

[1] conaticus01. *I Made a FAST File Explorer (in Rust)*. https://www.youtube.com/watch?v=Z60f2g-COJY. YouTube video. Juni 2023.

[2] conaticus02. *How I RUINED My Rust Project*. https://youtu.be/4wdAZQROc4A?si=ucYtPULZfrVvErBy. YouTube video. Juli 2023.

[3] Tauri Project. *Frontend Configuration*. Accessed: 2025-05-10. 2025. URL: https://v2.tauri.app/start/frontend/.

[4] Tauri Project. *State Management*. Accessed: 2025-05-10. 2025. URL: https://v2.tauri.app/develop/state-management/.

[5] Tauri Project. *Tauri - Build Smaller, Faster, and More Secure Desktop Applications with a Web Frontend*. Accessed: 2025-05-10. 2025. URL: https://v2.tauri.app/.

[6] Rust Team. *The Rust Lang*. Accessed: 2025-05-10. 2025. URL: https://www.rust-lang.org/.

Abbildungsverzeichnis

Eidesstaatliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und dabei keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher weder gesamt noch in Teilen einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

.....	.....	.....	.....
Marco Brandt	Daniel Schatz	Sören Panten	Lauritz Wiebusch

Ort, den 13. Mai 2025