

Independent Coursework 02

Hochschule für Technik und Wirtschaft Berlin



Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Feicount: Eine React Web-App zur gemeinsamen Finanzverwaltung mit ASP.NET Core

Dokumentation

Faye Bullwinkel
im Studiengang Internationale Medieninformatik
Matrikelnummer: 581885

7. März 2024

Bewertung: Prof. Dr. Gefei Zhang

Inhaltsverzeichnis

1	Einleitung	1
2	Anforderungen	2
2.1	System	2
2.2	Softwarearchitektur	3
2.3	Schulung und Wartung	3
3	Methodik	5
3.1	Projektsetup	5
3.2	Backend	8
3.2.1	Controller-Klassen und Mapper	8
3.2.2	Service-Klassen	10
3.2.3	Repositories	11
3.2.4	Models und Types	12
3.3	Frontend und Design	14
3.3.1	Startseite	16
3.3.2	Feicount	18
3.3.3	Abrechnung	23
3.4	Usability Testing	25
3.4.1	Versuchsdurchführung	25
3.4.2	Ergebnisse und Diskussion	26
4	Fazit und Ausblick	28
A	Anhang	30

1 Einleitung

Die effiziente und einfache Verwaltung gemeinsamer Ausgaben und Finanzen innerhalb einer Gruppe kann in vielen sozialen Situationen enorme Vorteile bringen. Beispielsweise ist es in einem gemeinsamen Haushalt (WG) oft so, dass eine Person die Einkäufe für die anderen Mitbewohner:innen übernimmt. Auch bei gemeinsamen Reisen oder Restaurantbesuchen mit Freunden kann es Zeit und Nerven sparen, wenn eine Person für alle bezahlt. Je größer die Gruppe und je unterschiedlicher die Ausgaben pro Person sind, desto aufwendiger wird es jedoch, den eigenen Anteil an einer Gruppenausgabe zurückzuzahlen. Genau in solchen Situationen ist eine gemeinsame Anwendung zur Verwaltung von gemeinsamen Finanzen von großem Wert. Durch ihre Nutzung rückt das gemeinsame Gruppenerlebnis in den Vordergrund und die manchmal lästige Frage nach den Finanzen in den Hintergrund.

Im Rahmen dieser Arbeit wurde daher beschlossen, eine Anwendung zu implementieren, die Listen über die innerhalb einer Gruppe geteilten Ausgaben erstellt und die Verrechnung von Transaktionen so einfach, übersichtlich und effizient wie möglich gestaltet. Das Hauptziel dieser Arbeit besteht darin, durch die Implementierung einer entsprechenden webbasierten Benutzeroberfläche wertvolle Erfahrungen im Umgang mit der Programmbibliothek React zu sammeln.

Zur Umsetzung des Forschungsvorhabens wurden zunächst die wesentlichen Anforderungen an das zu entwickelnde System sowie die Systemarchitektur erarbeitet. Anschließend wurde in Anlehnung an ein bereits bestehendes System eine eigene Anwendung mit einer klaren Trennung zwischen Backend und Frontend entwickelt. Abschließend wurde eine Untersuchung der Gebrauchstauglichkeit des Systems mit vier Teilnehmer:innen durchgeführt, die die Anwendung mithilfe von offenem Feedback und einem Fragebogen bewerteten. Die Ergebnisse zeigen eine hohe Zufriedenheit mit der Usability des Systems.

Die vorliegende Dokumentation beschreibt die Konzeption, Entwicklung und Testung der Anwendung *Feicount*. Der zugehörige Quellcode kann hier¹ eingesehen werden.

¹https://gitlab.rz.htw-berlin.de/s0581885/ic_02

2 Anforderungen

Das Hauptziel dieser Arbeit ist es, Erfahrungen mit der Programmbibliothek React zu sammeln. Zu diesem Zweck soll eine webbasierte Benutzeroberfläche entwickelt werden, die das Teilen von Ausgaben und die Verwaltung gemeinsamer Finanzen in Gruppen ermöglicht (siehe „Tricount - Organisieren von Gruppenausgaben“, n. d.). Der Schwerpunkt liegt auf der Benutzerfreundlichkeit des Systems, die durch eine nahtlose und einfache Verwaltung von Gruppenausgaben, eine intuitive Bedienung des Systems und ein ansprechendes Design definiert wird. Im Folgenden werden die Anforderungen an das System, die Softwarearchitektur sowie die Schulung und Wartung der Applikation näher erläutert.

2.1 System

Ziel der Anwendung ist es, gemeinsame Ausgaben innerhalb einer Gruppe transparent zu verfolgen, zu dokumentieren und gerecht aufzuteilen. Sie soll eine effiziente und nutzungsfreundliche Lösung für die gemeinsame Abrechnung von Ausgaben in Gruppenkontexten bieten. Es sollen neue Feicounts erstellt werden können, die Ausgaben in einem Kontext (Reise, Haushalt, Freundeskreis, etc.) zusammenfassen. Pro Feicount können Ausgaben erstellt werden, die unter ausgewählten Teilnehmer:innen des Feicounts fair aufgeteilt werden. Feicounts und deren Ausgaben müssen unter bestimmten Bedingungen nachträglich editierbar sein. Pro Feicount gibt es zudem eine Anzeige der Salden der aktuellen Mitglieder. Diese soll übersichtlich und nachvollziehbar darstellen, welche Person welchen Saldo hat. Für die Abrechnung sollen aus den Salden möglichst wenige Transaktionen errechnet werden. Wurde eine Transaktion durchgeführt, soll diese möglichst einfach in der App abgebildet und in der Saldenübersicht angezeigt werden.

Während der Entwicklung soll sich an den Usability-Kriterien von Nielsen orientiert werden. Diese bieten zehn Grundprinzipien für die Entwicklung von Benutzeroberflächen (UI), deren Einhaltung die Entwicklung intuitiver und nutzungsfreundlicher Produkte fördert (Nielsen, 1994). So soll das System die Nutzenden schnell und eindeutig über den Systemzustand informieren (intuitive Navigation und klare Informationsdarstellung) und bekannte Konzepte verwenden (z.B. bekanntes Vokabular). Weiterhin sollen die Nutzenden die Kontrolle über den Systemfluss haben (Dialoge können jederzeit abgebrochen oder verlassen werden) und das System soll in sich konsistent sein (z.B. sind

Zurück-Buttons immer an der gleichen Stelle). Darüber hinaus sollen Fehler vermieden werden, indem kritische Aktionen, die z.B. den Systemzustand verändern, gesondert bestätigt werden müssen. Außerdem wird ein minimalistisches Design angestrebt, das keine irrelevanten Informationen enthält und die wichtigen Informationen in den Vordergrund stellt.

2.2 Softwarearchitektur

Für die Entwicklung des Frontends wird React verwendet. Um die Entwicklung, Wartung und Skalierbarkeit des Systems zu erleichtern, wird eine komponentenbasierte Architektur implementiert, d.h. die Benutzeroberfläche wird in unabhängige, wiederverwendbare Komponenten aufgeteilt. Außerdem verwendet React ein virtuelles DOM, das die Effizienz bei der Aktualisierung des realen DOM verbessert, indem nur die geänderten Teile der Benutzeroberfläche aktualisiert werden, was zu einer besseren Performance führt. Für die Nutzung auf unterschiedlichen Endgeräten (Desktop, Tablet, Smartphone) soll grundlegendes Responsive Design integriert werden.

Basierend auf Vorkenntnissen in C# und dem Interesse, praktische Erfahrungen mit einem weiteren Web Application Framework zu sammeln, wird ASP.NET Core für die Implementierung des Backends verwendet. Damit wird der serverseitige Code in C# geschrieben und gleichzeitig React für die Erstellung der Benutzeroberfläche verwendet. Zur Speicherung von Nutzer:inneninformationen, Gruppendaten und Ausgaben wird eine PostgreSQL-Datenbank verwendet. Außerdem sollen effiziente Datenabfragen und -aktualisierungen für eine reaktionsfähige Benutzeroberfläche implementiert werden. Hierfür sowie für die Durchführung von Migrationen und die Generierung von Datenbankschemata wird zusätzlich das Kommandozeilentool *dotnet ef* in der .NET-Anwendung eingesetzt. Innerhalb der Architektur muss eine klare Kommunikation zwischen der React Frontend Applikation und dem ASP.NET Core Backend bestehen.

2.3 Schulung und Wartung

Das System soll von Grund auf intuitiv gestaltet sein. Die Nutzer:innen sollen bestenfalls so durch den Systemfluss geleitet werden, dass sie es selbst gar nicht bemerken. Das minimalistische Design und die leicht verständlichen Funktionen der Applikation sollen dies unterstützen. Die Applikation soll wenig fehleranfällig sein, indem Interaktionen selbsterklärend gestaltet

sind und das System umfassendes Feedback gibt. Mögliche Interaktionsfehler sollen vom System selbständig erkannt und abgefangen werden. Damit soll sichergestellt werden, dass die Nutzer:innen die App ohne zusätzliche Hilfe nutzen können.

Eine mögliche Weiterentwicklung des Systems soll durch eine gut dokumentierte Softwarearchitektur und eine klare Projektstruktur unterstützt werden. Es soll eine klare Trennung zwischen dem React-Frontend und dem .NET-Backend geben. Um Fehler und Ausfälle effektiv zu vermeiden, wird das System parallel zur Entwicklung getestet und das Versionskontrollsystem Git zur Nachverfolgung von Änderungen eingesetzt. Die kontinuierliche Weiterentwicklung und Pflege des Systems liegt im persönlichen Interesse, da die Anwendung privat genutzt werden soll. Die selbst implementierte Variante soll die Nutzung der kommerziellen Tricount-App ersetzen. Offene Umfragen im Freundes- und Bekanntenkreis haben ergeben, dass eingeblendete Werbung oder mögliche Paywalls anderer Anbieter als störend empfunden werden und daher die Nutzung einer selbst implementierten Alternative bevorzugt wird.

3 Methodik

Während der Entwicklung wurde sich stark an der bereits bestehenden Anwendung Tricount („Tricount - Organisieren von Gruppenausgaben“, n. d.) orientiert. In der ersten Projektphase lag der Fokus ausschließlich auf dem Backend. Nachdem die wichtigsten REST-Routen des Systems implementiert waren, wurde mit der Umsetzung des Frontends in React begonnen.

Es wurden laufend Klicktests durchgeführt und offenes Feedback von Freunden eingeholt, um so die Wünsche der potenziellen Nutzer:innen kontinuierlich in den Entwicklungsprozess mit einfließen zu lassen. Nach Fertigstellung des Systems wurde ein standardisierter Test des Systems durchgeführt, um die Usability des Systems quantitativ bewerten zu können. Nachfolgend wird die Umsetzung der React Web-App für die gemeinsame Finanzverwaltung mit ASP.NET Core im Detail beschrieben.

3.1 Projektsetup

Zunächst wurde mithilfe des .NET-CLI-Befehls *dotnet new react* ein neues React-Projekt in Kombination mit ASP.NET Core namens *tricount* (Stand früher Entwicklung) erstellt. Zusätzlich wurde das Target Framework auf *net8.0* gesetzt und EF-Tools, SQL Server und PostgreSQL für einen Dotnet-Entity-Framework-Core eingebunden. Die Open Source Datenbanktechnologie Postgres wurde aufgrund ihrer Erweiterbarkeit und Datenintegrität ausgewählt. Ihre hohe Skalierbarkeit ermöglicht die Unterstützung komplexer, standardbasierter SQL-Abfragen. Die schnelle und einfache Containerisierung der Datenbank mit Docker wurde gewählt, um die Daten von der Datenbankanwendung zu trennen. Im Falle eines Ausfalls kann einfach ein anderer Container gestartet werden, während die gesicherten Daten vor Beschädigung geschützt sind (Charboneau, 2022).

Um eine klare Trennung zwischen Backend und Frontend zu gewährleisten sowie die Datenbank- und Routing-Funktionalitäten effizient zu gestalten, wurde die Projektdatei *IC_02.sln* erstellt und der genaue Pfad zu den dotnet-Dateien durch den Eintrag *backend/tricount.csproj* festgelegt. In der Datei *Program.cs* wurde eine automatische Erstellung von Datenbanktabellen implementiert. In diesem Fall erfolgt die Migration der Datenbank über den C# Code, der die Serviceinstanz *ApplicationDbContext* in einem definierten Scope aufruft und anschließend die Funktion *Migrate* aufruft. Dadurch werden die Datenbanktabellen automatisch erzeugt. Die Erreichbarkeit der Backend-

Routen wurde durch die Integration der Datei *SetupProxy.js* sichergestellt. Diese Datei ermöglicht es, alle Routen in einem vordefinierten Kontext anzusprechen (in diesem Fall */api* und */swagger*). Die Controller-Routen wurden dann nach dem definierten Schema *[Route(/api/[controller])]* strukturiert, um ein klares und konsistentes Routing im Backend zu gewährleisten.

Um eine bessere Codequalität und Wartbarkeit des Systems zu gewährleisten, wurde beschlossen, TypeScript im Projekt zu verwenden. TypeScript bietet den Vorteil der statischen Typisierung, was die Codequalität verbessert, die Fehlerrate reduziert und die Entwicklung durch Autovervollständigung und bessere IDE-Unterstützung erleichtert. Wie in der React-Dokumentation beschrieben, wurde TypeScript zu dem bereits erstellten Projekt hinzugefügt (siehe „Using TypeScript – react“, 2024) und anschließend die bereits vorhandenen Dateien Schritt für Schritt in TypeScript konvertiert.

Abschließend wurde *Swagger* mithilfe des *Swashbuckle*-Pakets in das Backend integriert und das Projekt mit Git versioniert. Mithilfe von Swagger können alle Backend-Routen unabhängig von der Implementierung getestet werden, indem mithilfe eines webbasierten Dashboards direkt auf alle verfügbaren API-Routen zugegriffen wird (siehe Abbildung 1). Die Schnittstelle ermöglicht nicht nur das Betrachten der API-Endpunkte, sondern auch das Testen von Anfragen und das Überprüfen von Antworten in Echtzeit (siehe Abbildung 2).

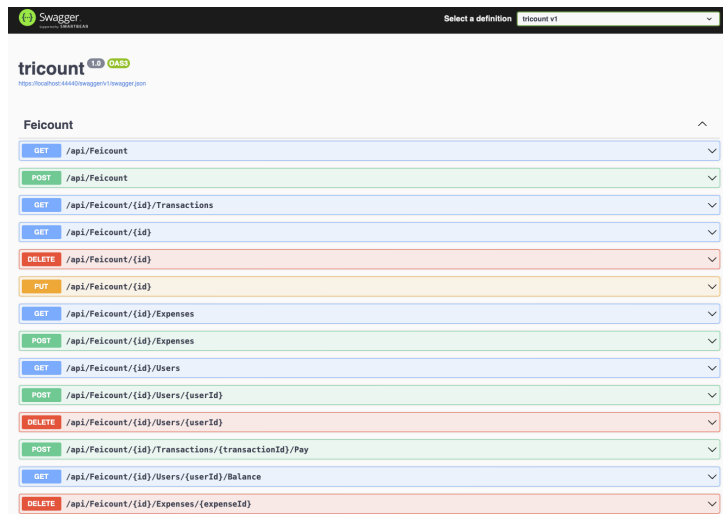


Abbildung 1: Eine Übersicht der mit Swagger testbaren Routen, Stand früher Entwicklung.

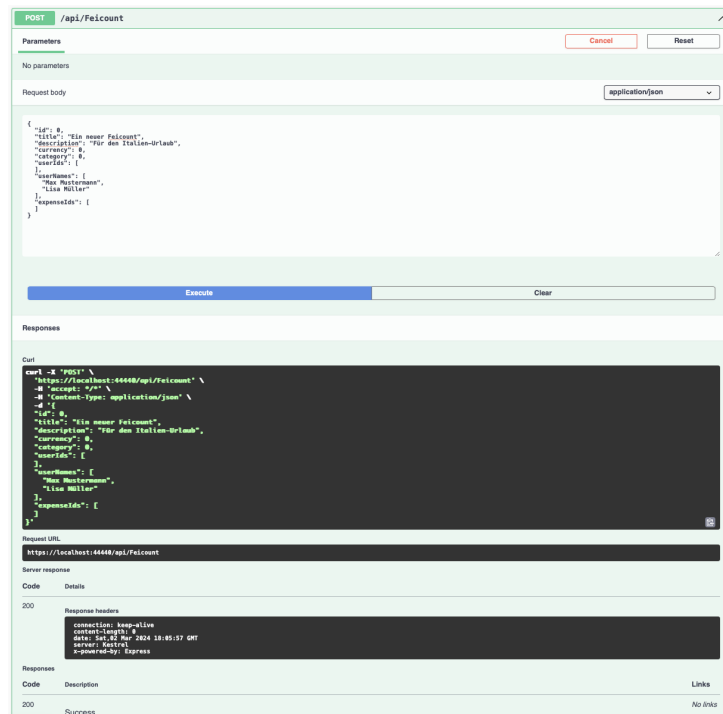


Abbildung 2: Eine erfolgreiche POST-Anfrage zur Erstellung eines neuen Feicounts.

3.2 Backend

Zunächst wurde das Backend implementiert. Dabei wurden Mechanismen zur Berechnung von Ausgaben, Aufteilung von Kosten und anderen finanzbezogenen Operationen implementiert. Gleichzeitig erfolgte die Validierung von Nutzer:inneneingaben, um die Datenkonsistenz sicherzustellen. Abbildung 3 zeigt die wichtigsten Bestandteile des Backends, deren Implementierung und Funktionen in den folgenden Unterkapiteln detailliert beschrieben werden.

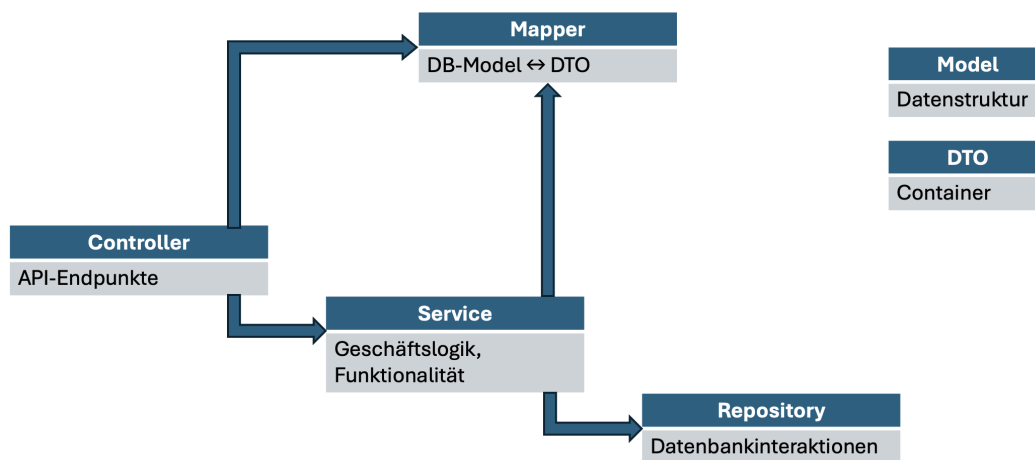


Abbildung 3: Vereinfachte Darstellung der wichtigsten Komponenten des Backends und ihrer Relationen.

3.2.1 Controller-Klassen und Mapper

Die Kommunikation zwischen Frontend und Backend erfolgt über API-Endpunkte, die in den Controller-Klassen implementiert sind. Im Rahmen dieses Projekts wurden zwei Controller implementiert, der *FeicountController*, der für alle Anfragen bezüglich eines Feicounts zuständig ist, und der *UserController*, der für die Interaktionen mit Nutzer:innenobjekten verantwortlich ist.

Der *FeicountController* stellt den Kern der Anwendung dar und enthält 14 verschiedene API-Endpunkte. Über diese können alle existierenden Feicounts oder einzelne Feicounts anhand ihrer ID abgefragt werden. Es können auch

die einem spezifischen Feicount zugehörigen Nutzer:innen oder Ausgaben abgerufen oder welche hinzugefügt werden. Ebenso können neue Feicounts oder neue Ausgaben in einem Feicount angelegt und bestehende Benutzer:innen zu einem Feicount hinzugefügt werden. Nachdem ein Feicount angelegt wurde, kann dieser mithilfe der *PUT /api/Feicount/{id}* Route anhand seiner ID mit den im Request Body übergebenen Daten aktualisiert werden. Es ist auch möglich, den Saldo eines Benutzers oder einer Benutzerin für einen bestimmten Feicount abzufragen. Es ist zu beachten, dass unter bestimmten Bedingungen Nutzer:innen, Ausgaben oder ganze Feicounts gelöscht werden können, weswegen der Controller entsprechende Routen beinhaltet. Der Controller enthält außerdem zwei Routen, um mit den Transaktionen pro Feicount zu interagieren. Die erste liefert eine Liste der Transaktionen, die aus den Salden des Feicounts berechnet werden. Die zweite markiert eine Transaktion als abgeschlossen, indem eine Ausgabe vom Schuldner an den Gläubiger mit dem Betrag der Transaktion erstellt wird.

Im *FeicountController* sind vier Routen unterschiedlicher HTTP-Methoden definiert. Der Basispfad für alle Routen ist */api/User*. Zunächst ist es möglich, alle in der Datenbank angelegten, sowie einzelne Nutzer:innen über deren ID ausgeben zu lassen. Außerdem kann ein neuer Nutzer oder eine neue Nutzerin per POST-Request angelegt oder unter bestimmten Bedingungen anhand der ID wieder gelöscht werden.

Um eine konsistente und effiziente Datenverarbeitung innerhalb der Anwendung zu gewährleisten, wurden neben den Controller-Klassen auch zwei Mapper-Klassen (*FeicountMapper* und *UserMapper*) implementiert. Diese bieten eine Reihe von Methoden, um Modelle in Datenübertragungsobjekte (DTOs) zu übertragen und umgekehrt. Die klare Trennung der Konvertierungslogik in den Mappern trägt zur verbesserten Wartbarkeit und Lesbarkeit des Codes bei, da die Verantwortlichkeiten für die Datenkonvertierung klar definiert sind.

Die Klasse *FeicountMapper* enthält Methoden zur Konvertierung zwischen Feicount- oder Ausgaben-Datenbankobjekten und den entsprechenden DTOs. Diese werden für die Datenübertragung und Transformation im *FeicountController* benötigt. In der Klasse *UserMapper* sind Methoden definiert, um ein User-Modell aus der Geschäftslogik in ein User-DTO zu konvertieren und umgekehrt. Das User-DTO wird neben den Anfragen im *UserController* im *FeicountController* benötigt, um als Antwort auf eine HTTP-GET-Anfrage an *api/Feicount/id/Users* alle Nutzer:innen eines bestimmten Feicounts zu senden.

3.2.2 Service-Klassen

Service-Klassen kapseln Geschäftslogik und Funktionalität, die nicht direkt mit der Verarbeitung von HTTP-Anfragen und -Antworten in den Controllern zu tun hat. Auf diese Weise konzentrieren sich die Controller auf die Verarbeitung von HTTP-Anfragen und -Antworten, während die Service-Klassen die eigentliche Funktionalität bereitstellen. Dadurch wird der Code in den Controllern übersichtlicher und leichter verständlich.

Die Service-Klasse *FeicountService* stellt eine Reihe von Methoden zur Verfügung, um Operationen im Zusammenhang mit Feicounts und zugehörigen Entitäten durchzuführen. Die Klasse implementiert das Interface *IFeicountService* und arbeitet mit verschiedenen Repositories und dem entsprechenden Mapper (*IFeicountMapper*) zusammen. Die Methode *CreateFeicount* ermöglicht die Erstellung eines neuen Feicounts unter Verwendung von DTO-Daten, wobei die entsprechenden Nutzer:innen und Ausgaben aus den Repositories abgerufen werden. Andere Methoden bieten Funktionalitäten wie das Abrufen von Feicount-Details (wie Ausgaben, Nutzer:innen und Transaktionen), das Löschen von Feicounts und Ausgaben, das Hinzufügen/Löschen von Nutzer:innen zu/von Feicounts, das Abrufen des Saldos eines Users, das Bezahlen von Transaktionen sowie das Aktualisieren von Feicount-Details basierend auf DTO-Informationen. Zunächst wurde im Rahmen einer *DistributeRemainingUserExpenses*-Funktion die Logik implementiert, dass beim Löschen eines Nutzers oder einer Nutzerin die verbleibenden Nutzer:innen-Ausgaben gleichmäßig auf die verbleibenden Nutzer:innen im Feicount aufgeteilt werden. Dieser Ansatz wurde im Laufe des Projekts verworfen und festgelegt, dass Nutzer:innen nur dann gelöscht werden dürfen, wenn sie keine Ausgaben in einem Feicount haben.

Die Service-Klasse *UserService* bietet analog Methoden zur Verwaltung von Nutzer:innen und deren Beziehungen zu Feicounts und Ausgaben. Die Klasse implementiert das Interface *IUserService* und arbeitet mit verschiedenen Repositories und dem Mapper (*IUserMapper*) zusammen. Die *CreateUser*-Methode ermöglicht die Erstellung eines neuen Users unter Verwendung von DTO-Daten, wobei die zugehörigen Feicounts und Ausgaben aus den jeweiligen Repositories abgerufen werden. Es wird überprüft, ob die referenzierten Feicounts und Ausgaben existieren, andernfalls werden entsprechende Ausnahmen (Exceptions) ausgelöst. Der User wird dann mithilfe des Mappers erstellt und in der Datenbank gespeichert. Die restlichen Methoden umfassen das Abrufen aller Nutzer:innen (*FindAll*), das Abrufen eines bestimmten

Nutzers oder einer bestimmten Nutzerin anhand seiner/ihrer ID (*FindById*), sowie das Löschen von Nutzer:innen (*Delete*).

Die Service-Klassen ermöglichen grundlegende CRUD-Operationen im Kontext von Feicounts und Usern und bieten eine Schnittstelle für die Interaktion zwischen Controllern und Datenbank-Repositories.

3.2.3 Repositories

Die Repositories befinden sich im Verzeichnis *Data* und dienen als Schnittstelle zwischen der Anwendungslogik und der persistenten Datenspeicherung (repräsentiert durch den *ApplicationDBContext*). Sie dienen dazu, den Datenzugriff zu abstrahieren und die Interaktion mit der Datenbank zu kapseln. Diese Abstraktion ermöglicht eine verbesserte Wartbarkeit und Testbarkeit des Codes, da die konkrete Implementierung des Datenzugriffs leicht ausgetauscht werden kann, ohne die darüber liegende Geschäftslogik zu beeinträchtigen. Im Rahmen dieses Projekts wurden drei Repositories implementiert, das *FeicountRepository*, das *UserRepository* und das *ExpenseRepository*.

Das *FeicountRepository* implementiert das *IFeicountRepository*-Interface und beinhaltet Methoden, um auf verschiedene Datenbankoperationen für die Entität *Feicount* zuzugreifen. Die Methoden ermöglichen das Finden aller Feicounts (*FindAll*), das Finden eines Feicounts anhand seiner ID (*FindById*), das Erstellen neuer Feicounts (*Create*), das Löschen eines Feicounts (*Delete*), das Hinzufügen von Ausgaben (*AddExpenseToFeicount*) und Nutzer:innen (*AddUserToFeicount*) zu einem Feicount, das Abrufen einer Liste aller Nutzer:innen eines Feicounts (*GetUsers*), das Löschen eines Users aus einem Feicount (*DeleteFeicountUser*), das Abrufen des Feicounts für eine bestimmte Ausgabe (*GetFeicountForExpense*) und das Aktualisieren eines Feicounts (*UpdateFeicount*). Die Implementierung nutzt Entity Framework Core, eine ORM-Lösung, um den Datenzugriff zu abstrahieren. Die *FindAll*- und *FindById*-Methoden verwenden das Lazy-Loading-Konzept, um gleichzeitig die zugehörigen Nutzer:innen und Ausgaben abzurufen und Datenbankabfragen zu optimieren.

Analog dazu implementiert das *UserRepository* Methoden, die den Zugriff auf alle Nutzer:innen im System oder auf einzelne Nutzer:innen per ID ermöglichen, mit deren Hilfe neue Nutzer:innen auch erstellt oder bestehende gelöscht werden können. Zusätzlich bietet die Methode *FindByNameOrCreate* die Möglichkeit, Nutzer:innen anhand ihrer Usernames in der Datenbank zu suchen. Wenn ein:e Nutzer:in gefunden wird, wird diese:r zurückgegeben.

Andernfalls wird ein neues User-Objekt mit dem angegebenen Namen erstellt, zur Datenbank hinzugefügt und die Änderungen in der Datenbank gespeichert. Schließlich wird das erzeugte oder gefundene User-Objekt zurückgegeben.

Das *ExpenseRepository* implementiert die Methoden *FindAll*, *FindById*, *Delete* und *Create* entsprechend für die Ausgaben eines Feicounts. Zusätzlich können mithilfe der Methode *FindForFeicount* alle Ausgaben für einen bestimmten Feicount abgerufen werden. Die Methode verwendet Entity Framework Core, um eine Liste von Ausgaben zu extrahieren, die dem angegebenen Feicount anhand seiner ID zugeordnet sind. Dabei werden auch die zugehörigen Feicounts, Ausgabenersteller:innen (*Spender*) und Empfänger:innen (*Recipients*) mit eingeschlossen.

Listing 1: Pseudocode zur Berechnung des Saldos eines Users

```
function CalculateUserBalance(user):
    totalSpent = sum(expense.Amount for expense in Expenses if
        expense.Spender.Id == user.Id)

    totalReceived = sum(expense.Amount / len(expense.Recipients) for
        expense in Expenses if user in expense.Recipients)
    balanceAmount = totalSpent - totalReceived

    return createUserBalance(userId: user.Id, amount: balanceAmount)
```

3.2.4 Models und Types

Die Models und Types validieren Nutzer:inneneingaben und stellen somit die Datenkonsistenz innerhalb des Systems sicher. Die Models repräsentieren den Kern der Datenstruktur und Geschäftslogik der Anwendung, indem sie die strukturelle Grundlage der Anwendung bilden. Sie enthalten sowohl Datenattribute als auch Methoden zur Verarbeitung dieser Daten. Ein Beispiel ist die Klasse *Expense*, die eine Ausgabe repräsentiert und Attribute wie *Titel*, *Betrag*, *Währung* und *Datum* enthält. Die Klasse *Feicount* repräsentiert eine Finanzgemeinschaft und enthält Methoden wie *CalculateUserBalance* und *CalculateTransactions* zur Berechnung von Nutzer:innenbilanzen (siehe Listing 1) und Transaktionen (siehe Listing 2).

Um die Gesamtanzahl der Transaktionen zu minimieren, wird zunächst der Saldo aller Nutzer:innen berechnet. Hierbei werden die Gesamtausgaben

von den erhaltenen Beträgen pro Person abgezogen. Anschließend werden die Transaktionen erstellt. Positive Salden werden mit einer Gläubiger-ID und negative Salden mit einer Schuldner-ID gekennzeichnet, die der User-ID entspricht. Die Transaktionen werden nach ihrer Höhe sortiert und so kombiniert, dass Schulden möglichst effizient beglichen werden (jwoolard, 2009).

Die Vorgehensweise erinnert an das Partitionsproblem (Wikipedia, 2024) und das Bin Packing Problem (Wikipedia, 2024). Wie beim Partitionsproblem wird versucht, eine Menge von Zahlen in zwei Teile aufzuteilen, deren Summen möglichst gleich sind. Der Algorithmus weist auch Ähnlichkeiten zum Bin Packing Problem auf, da er versucht, die Transaktionen in möglichst wenigen Paketen zu organisieren. Jedes Paket stellt dabei eine finanzielle Transaktion dar. In beiden Fällen geht es darum, Ressourcen (Geldbeträge) effizient zu verteilen oder zu organisieren.

Listing 2: Pseudocode zur Berechnung von Transaktionen in einem Feicount

```
function CalculateTransactions():
    transactions = createEmptyList()

    for each user in Users:
        totalAmount = calculateUserBalance(user).Amount

        switch totalAmount:
            case > 0:
                // Creditor
                transactions.add(createTransaction(0, user.Id,
                    totalAmount))
                break
            case < 0:
                // Debtor
                transactions.add(createTransaction(user.Id, 0,
                    totalAmount))
                break

    creditors = filterTransactions(transactions,
        debtorIdEqualsTo(0)).orderByDescending(amount).toList()
    debtors = filterTransactions(transactions,
        creditorIdEqualsTo(0)).orderBy(amount).toList()
```

```

for each creditor in creditors:
    for each debtor in debtors:
        if creditor.Amount <= 0 or debtor.Amount >= 0:
            continue

        transferAmount = min(creditor.Amount, -debtor.Amount)

        transactions.add(createTransaction(debtor.DebtorId,
            creditor.CreditorId, transferAmount))
        creditor.Amount -= transferAmount
        debtor.Amount -= transferAmount

return filterTransactions(transactions,
    nonZeroCreditorAndDebtorIds()).toList()

```

Weitere Klassen der Datenstruktur sind *User*, *UserBalance* und *Transaction*. Enums wie *Category* und *Currency* werden verwendet, um vordefinierte Kategorien und Währungen zu repräsentieren.

Die im Verzeichnis *Types* definierten DTOs dienen als Container für den effizienten Austausch von Daten, ohne dabei die gesamte Komplexität der Models abzubilden. Sie enthalten keine Geschäftslogik, sondern nur Datenattribute, die benötigt werden, um Informationen zwischen verschiedenen Teilen eines Systems (Backend und Frontend) zu übertragen. Die *ExpenseDto*-Klasse fungiert als Datencontainer für Ausgabeninformationen, wobei sie im Vergleich zum *Expense*-Model lediglich die IDs des *Spenders* und der *Recipients* und nicht die entsprechenden Models enthält. Im *UserDto* sind analog die IDs der Feicounts und der Ausgaben eines Users definiert. Das *FeicountDto* enthält die IDs der Nutzer:innen und Ausgaben in einem Feicount, sowie eine Liste der Namen von Nutzer:innen. Damit können im Frontend direkt die Namen von Teilnehmer:innen in einem Feicount anstatt deren IDs eingegeben werden.

3.3 Frontend und Design

Nach Abschluss der Backend-Entwicklung wurde mit der Umsetzung des Frontends in React begonnen.

Hierzu wurde sich zunächst grundlegend in die Funktionsweisen von React anhand eines Tutorials („Tutorial: Tic-Tac-Toe – React“, n. d.) eingearbeitet. Im Rahmen des Tutorials wurde ein einfaches TicTacToe-Spiel entwickelt,

welches von zwei Spieler:innen am gleichen Endgerät gespielt werden kann. Besonders hervorzuheben ist die integrierte *TimeTravel* Funktion, eine der besonderen Stärken von React, mit deren Hilfe Spielzüge dokumentiert und später wieder aufgerufen werden können. Abbildung 4 zeigt das Ergebnis einer Partie.

Winner: X

X	O	
X	X	X
O		O

1. Go to game start
2. Go to move #1
3. Go to move #2
4. Go to move #3
5. Go to move #4
6. Go to move #5
7. Go to move #6
8. Go to move #7

Abbildung 4: Ergebnis einer Partie, in der Spieler X nach sieben Zügen gewonnen hat.

Im Rahmen des Frontends der Feicount-Anwendung wurden eine Reihe von Komponenten implementiert, die hierarchisch voneinander abhängen. In React kommunizieren Parent- und Child-Komponenten hierarchisch über Props und Callback-Funktionen miteinander. Parent-Komponenten übergeben Daten über Props an ihre Kinder, während Callbacks als Funktionen an die Kinder übergeben werden. Die Kinder können dann diese Funktionen aufrufen, um Daten oder Ereignisse an die Eltern zurückzugeben. Die Beziehungen zwischen den Komponenten werden in Abbildung 5 abstrakt dargestellt. Im folgenden Kapitel werden die Komponenten und ihr Zusammenwirken detailliert beschrieben.

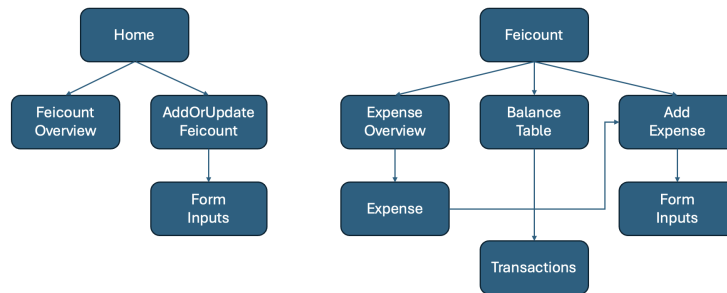


Abbildung 5: Vereinfachte Übersicht über die wichtigsten Frontend-Komponenten und ihre Beziehungen untereinander.

3.3.1 Startseite

Die Startseite der Anwendung zeigt eine Übersicht aller bisher angelegten Feicounts (siehe Abbildung 6). Ein Feicount-Element in der Übersicht enthält den Titel des Feicounts und, falls vorhanden, eine Beschreibung. Wurden noch keine Feicounts angelegt, ist die Übersichtsseite bis auf einen prominent platzierten Button zum Hinzufügen eines neuen Feicounts leer. Am oberen Bildschirmrand befindet sich ein Menü, das während der gesamten Anwendung sichtbar ist und jederzeit zur Startseite zurückführt.

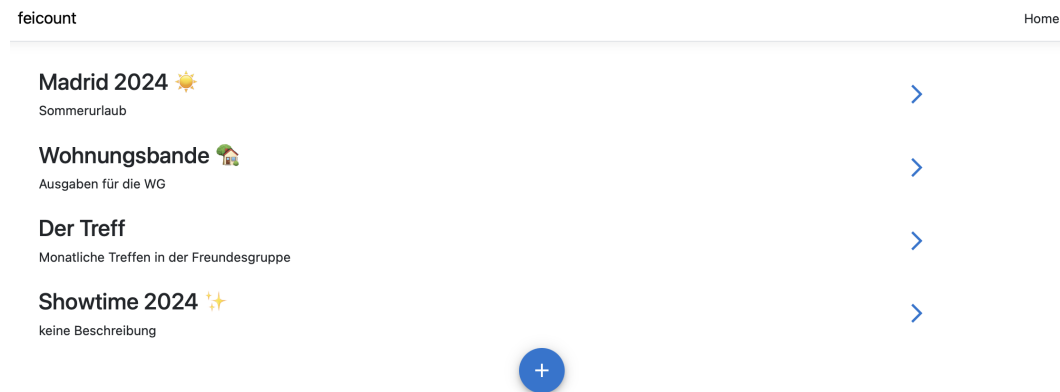


Abbildung 6: Startseite der Anwendung mit einer Übersicht aller bereits angelegten Feicounts.

Wenn Nutzer:innen auf den Hinzufügen-Button klicken, werden sie auf eine zweite Seite weitergeleitet. Dort können sie mithilfe eines Formulars die Feicount-Daten festlegen (siehe Abbildung 7). Der Titel und die optionale Beschreibung werden als Freitext eingegeben. Die Währung des Feicounts sowie die Kategorie können aus einem Dropdown-Menü ausgewählt werden. Die Namen der Teilnehmer:innen und Teilnehmer werden individuell über ein Textfeld und einen separaten Hinzufügen-Button in eine Liste eingetragen. Es ist zu beachten, dass bei der Erstellung eines Feicounts die Nutzer:innen noch bearbeitet oder gelöscht werden können, falls man sich beispielsweise vertippt hat. Ist der Feicount erst einmal erstellt, können die Nutzer:innen nur dann noch gelöscht werden, wenn sie keine Ausgaben haben.

feicount
Home

Neuer Feicount

Titel *
Ostsee-Urlaub 🏖️

Beschreibung
Reise ans Meer

Währung *
EUR

Kategorie *
Reise

Teilnehmer:innen
Faye

SPEICHERN

Anton
ÄNDERN
LÖSCHEN

Raphi
ÄNDERN
LÖSCHEN

Toni
ÄNDERN
LÖSCHEN

Kathi
ÄNDERN
LÖSCHEN

Vivi
ÄNDERN
LÖSCHEN

Fay
ÄNDERN
LÖSCHEN

ABBRECHEN

SPEICHERN

Abbildung 7: Formular zum Anlegen eines neuen Feicounts mit nachträglicher Änderung eines UserNames.

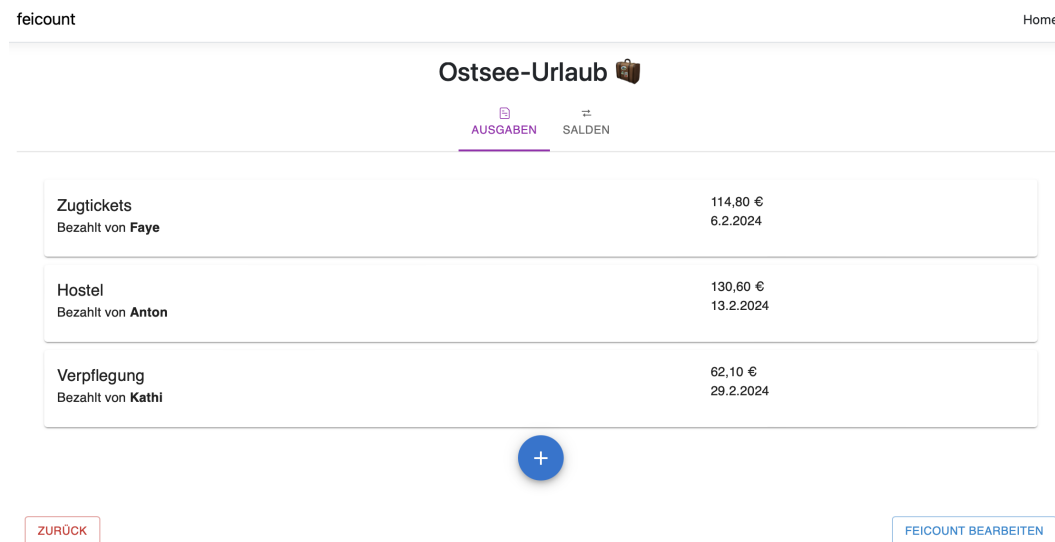
Zum besseren Verständnis wurden alle Buttons mit einer Textbeschriftung versehen, die die Funktion des jeweiligen Buttons erklärt. Mit einem Klick

auf den Bestätigen-Button wird der neu erstellte Feicount gespeichert und die Nutzer:innen gelangen wieder auf die Übersichtsseite aller Feicounts.

3.3.2 Feicount

Auf der Feicount-Übersichtsseite gelangt man über einen Pfeil am rechten Rand zur Detailseite des jeweiligen Feicounts. Diese ist durch ein *TabPanel* am oberen Rand zweigeteilt. Das *TabPanel* wurde wie viele andere Designelemente mit Hilfe der React-Komponentenbibliothek *Material UI* (MUI, n. d.) erstellt. Material-UI bietet vorgefertigte, stilisierte React-Komponenten, die eine einfache Integration von Material Design-Elementen in React-Anwendungen ermöglichen. Diese Komponenten umfassen Elemente wie Buttons, Formulare, Dialoge, Navigationsleisten und vieles mehr.

Analog zum Design der Startseite wird beim Öffnen der Seite der Tab mit einer Übersicht aller bisher angelegten Ausgaben innerhalb eines Feicounts angezeigt (siehe Abbildung 8).



The screenshot shows a web interface for a 'Feicount' titled 'Ostsee-Urlaub'. At the top, there are tabs for 'AUSGABEN' (selected) and 'SALDEN'. Below the tabs is a list of three expenses, each in a white box with a light purple border. The first expense is 'Zugtickets' for 114,80 € paid by 'Faye' on 6.2.2024. The second is 'Hostel' for 130,60 € paid by 'Anton' on 13.2.2024. The third is 'Verpflegung' for 62,10 € paid by 'Kathi' on 29.2.2024. Below the list is a blue circular button with a white plus sign. At the bottom, there are two buttons: 'ZURÜCK' (red outline) and 'FEICOUNT BEARBEITEN' (blue outline).

Ostsee-Urlaub 🗑️	
AUSGABEN	SALDEN
Zugtickets Bezahlt von Faye	114,80 € 6.2.2024
Hostel Bezahlt von Anton	130,60 € 13.2.2024
Verpflegung Bezahlt von Kathi	62,10 € 29.2.2024

ZURÜCK FEICOUNT BEARBEITEN

Abbildung 8: Übersichtsseite aller Ausgaben in einem Feicount.

In der Übersicht werden der jeweilige Titel der Ausgabe, die Person, die die Ausgabe getätigt hat, der Betrag und das Datum der Ausgabe angezeigt. Zusätzlich befindet sich in der Mitte, direkt unter der letzten Ausgabe,

ein Button zum Hinzufügen einer neuen Ausgabe. Über diesen gelangen die Nutzer:innen zu einem weiteren Formular zum Anlegen einer Ausgabe. Das Design dieser Komponente orientiert sich an der Komponente zum Anlegen eines neuen Feicounts, wurde aber um einige ausgaben-spezifische Funktionen erweitert (siehe Abbildung 9).

The screenshot shows a web application interface for creating a new output. At the top, there's a header with 'feicount' on the left and 'Home' on the right. The main title of the form is 'Neue Ausgabe'. The form contains several input fields: 'Titel' (Title) with the value 'Zugtickets', 'Betrag *' (Amount) with '114,80', 'Währung *' (Currency) with a dropdown set to 'EUR', 'Datum *' (Date) with '06.03.2024', and 'Bezahlt von *' (Paid by) with a dropdown set to 'Kathi'. Below these fields is a section titled 'Für wen' (For whom) which contains a list of participants, each with a checked checkbox and a corresponding amount of '19,13 EUR'. The participants are Kathi, Toni, Raphi, Anton, Vivi, and Faye. At the bottom of the form, there are two buttons: 'ABBRECHEN' (Cancel) and 'SPEICHERN' (Save).

Neue Ausgabe		
Titel Zugtickets		
Betrag *	114,80	Währung * EUR
Datum * 06.03.2024		
Bezahlt von * Kathi		
Für wen		
<input checked="" type="checkbox"/> Kathi		19,13 EUR
<input checked="" type="checkbox"/> Toni		19,13 EUR
<input checked="" type="checkbox"/> Raphi		19,13 EUR
<input checked="" type="checkbox"/> Anton		19,13 EUR
<input checked="" type="checkbox"/> Vivi		19,13 EUR
<input checked="" type="checkbox"/> Faye		19,13 EUR
ABBRECHEN		SPEICHERN

Abbildung 9: Formular zum Erstellen einer neuen Ausgabe.

Der optionale Titel der Ausgabe wird durch einen Freitext festgelegt. Der Betrag der Ausgabe wird auf formale Korrektheit geprüft und bei erfolgreicher Eingabe mit den Funktionen aus Listing 3 geparkt und formatiert. Das Datum der Ausgabe kann über einen Kalender gesetzt werden. Die Person, die die Ausgabe getätigt hat, kann über ein Dropdown-Feld aus allen Teilnehmer:innen des Feicounts ausgewählt werden. Standardmäßig sind alle Teilnehmer:innen als Empfänger:innen einer Ausgabe ausgewählt und diese wird gleichmäßig zwischen allen aufgeteilt. Über die Checkboxes neben den jeweiligen Namen können einzelne Teilnehmer:innen von einer Ausgabe entfernt werden, falls diese nicht für sie getätigt wurde. In diesem Fall wird der

Betrag gleichmäßig zwischen den restlichen Teilnehmer:innen aufgeteilt.

Nutzer:innen gelangen über die beiden Buttons am unteren Rand der Seite zurück zur Übersicht aller Ausgaben in einem Feicount. Mit Klick auf Speichern wird eine neue Ausgabe mit den eingegebenen Daten angelegt und in der Übersicht angezeigt, mit Abbrechen werden die Änderungen verworfen. Zum aktuellen Zeitpunkt können Ausgaben mit Nachhinein nicht mehr editiert werden.

Listing 3: Pseudocode zum Parsen und Formatieren eines gültigen Ausgabebetrages.

```
function parseMonetaryValue(valueString: string):
    re = /^(-)?(\d*)(?:[.,](\d{0,2}))?\d*$/

    // Remove whitespaces and match the string with the regular
    // expression
    match = valueString.replace(/\s/g, "").match(re)

    // Check for a valid match and presence of digits
    if not match or length(match[2]) + (length(match[3]) if match[3]
        else 0) == 0:
        return null

    // Determine the sign, euros, and cents
    sign = match[1] ? -1 : 1
    euros = parseInt(match[2]) or 0
    cents = parseInt(match[3]) or 0

    // Calculate the total value in cents
    return sign * (euros * 100 + cents)

function formatPrice(value: number): string:
    // Extract euros and cents, then format the monetary value
    euros = value / 100 >> 0
    formattedEuros = euros.toString().padStart(1, "0")
    formattedCents = Math.abs(value % 100).toString().padStart(2,
        "0")

    // Combine euros and cents with a comma separator
    return formattedEuros + "," + formattedCents
```

Am unteren Rand der Übersichtsseite der Ausgaben (Abbildung 8) in einem Feicount befindet sich eine Buttonleiste mit einem Zurück-Button, der die Nutzer:innen zurück zur Übersicht aller Feicounts führt, sowie einem Button zum Bearbeiten des aktuell angezeigten Feicounts. Über den Bearbeiten-Button gelangen die Nutzer:innen zu einer separaten Bearbeitungsseite des Feicounts. Hier können der Titel, die Beschreibung, die Währung und die Kategorie des Feicounts nachträglich angepasst werden (siehe Abbildung 10).

The screenshot shows a web form for editing a 'Feicount'. The form is titled 'Ostsee-Urlaub' with a suitcase icon. It contains four input fields: 'Titel *' (filled with 'Nordsee-Urlaub' and a suitcase icon), 'Beschreibung' (filled with 'Reise an die Nordsee'), 'Währung *' (filled with 'USD'), and 'Kategorie *' (filled with 'Freundesgruppe'). At the bottom of the form are two buttons: 'ABBRECHEN' (red outline) and 'SPEICHERN' (purple outline). The page header shows 'feicount' on the left and 'Home' on the right.

Abbildung 10: Bearbeitungsansicht eines bereits angelegten Feicounts.

Zu beachten ist, dass zur Aktualisierung eines bereits angelegten Feicounts die gleiche Komponente verwendet wird wie für die Erstellung eines Feicounts. Beim Laden der Seite wird geprüft, ob beim Aufruf der optionale Parameter *id* übergeben wurde. Wenn ja, wird der Feicount mit der entsprechenden ID mittels der Methode *FetchFeicount* abgefragt (siehe Listing 4). Die Methode *useEffect*, in der die Funktion *FetchFeicount* eingebettet ist, wird in React verwendet, um Seiteneffekte in funktionalen Komponenten zu verwalten. Sie ermöglicht die Integration von Code zum Laden, Abrufen oder Manipulieren von Daten nach dem Rendern der Komponente, d.h. in diesem Fall das nachträgliche Laden eines Feicounts.

Listing 4: Pseudocode zur API-Abfrage eines existierenden Feicounts

```
useEffect(() => {
  function fetchFeicount():
    feicountResponse = async fetch(`/api/Feicount/${id}`)
```

```

    if not feicountResponse.ok:
        throw new Error('Failed to fetch feicount with id ${id}')

    feicountData = await feicountResponse.json()
    setFeicount(feicountData)

    if id exists:
        call fetchFeicount function

}, [id])

```

Anschließend wird das Formular mit den Daten des Feicounts vorausgefüllt und beim Klick auf den Speichern-Button wird der Feicount aktualisiert und nicht wie bisher ein neuer angelegt. Wird ein Feicount aktualisiert, gelangen die Nutzer:innen mit einem Klick auf den Abbrechen-Button zur Ausgabenübersicht des aktuellen Feicounts anstatt zur Übersicht aller Feicounts.

feicount		Home
<div>Nordsee-Urlaub 🏠</div> <div> <div>AUSGABEN</div> <div>SALDEN</div> </div>		
	-111,73 €	Kathi
	-85,98 €	Toni
	-86,73 €	Raphi
Anton	23,27 €	
Vivi	298,72 €	
	-37,48 €	Faye
ZURÜCK		ZUR ABRECHNUNG

Abbildung 11: Tabellarische Übersicht der Salden der Teilnehmer:innen des Feicounts.

Der zweite Tab der Feicount-Seite führt die Nutzer:innen zur Übersicht der aktuellen Salden aller Teilnehmer:innen eines Feicounts (siehe Abbildung 11). Diese werden tabellarisch angezeigt, wobei jeweils in einer Spalte der Name

und in der anderen Spalte der Saldo der Teilnehmer:in angezeigt wird. Wenn der Saldo positiv ist, wird er in der rechten Spalte vor einem grünen Balken angezeigt, wenn er negativ ist, wird er in der linken Spalte vor einem roten Balken angezeigt. Die Länge der Balken entspricht dem prozentualen Anteil des Saldos, wobei der größte Betrag aller Ausgaben 100% entspricht.

3.3.3 Abrechnung

feicount		Home
Kathi schuldet Vivi	111,73 €	BEZAHLEN
Raphi schuldet Vivi	86,73 €	BEZAHLEN
Toni schuldet Vivi	85,98 €	BEZAHLEN
Faye schuldet Vivi	14,28 €	BEZAHLEN
Kathi schuldet Anton	23,27 €	BEZAHLEN
ZURÜCK		

Abbildung 12: Übersicht aller Transaktionen eines Feicounts.

Durch Klicken auf *Zur Abrechnung* auf der Saldenübersichtsseite gelangen die Nutzer:innen zu einer Übersicht aller Transaktionen, die zur Begleichung der Feicount-Salden benötigt werden (siehe Abbildung 12). Die Transaktionen werden in Material UI *Card*-Komponenten angezeigt, um sie optisch voneinander zu unterscheiden.

Eine Transaktion zeigt auf der linken Seite, welche Person, wem etwas schuldet. Auf der rechten Seite befindet sich in der Mitte der zu überwei-

sende Betrag und direkt darunter ein Button zum Begleichen der jeweiligen Transaktion. Um versehentlichen Eingaben vorzubeugen, müssen Nutzer:innen zum Begleichen einer Transaktion separat über ein Dialogfenster bestätigen, dass der Transaktionsbetrag überwiesen wurde (siehe Abbildung 13). Nachdem die Rückzahlung einer Transaktion bestätigt wurde, wird automatisch eine Ausgabe in Höhe des zu begleichenden Betrags vom *Deptor* an den *Creditor* dem Feicount zur Nachverfolgung hinzugefügt. Die Salden der von der Transaktionen betroffenen Teilnehmer:innen werden automatisch angepasst.

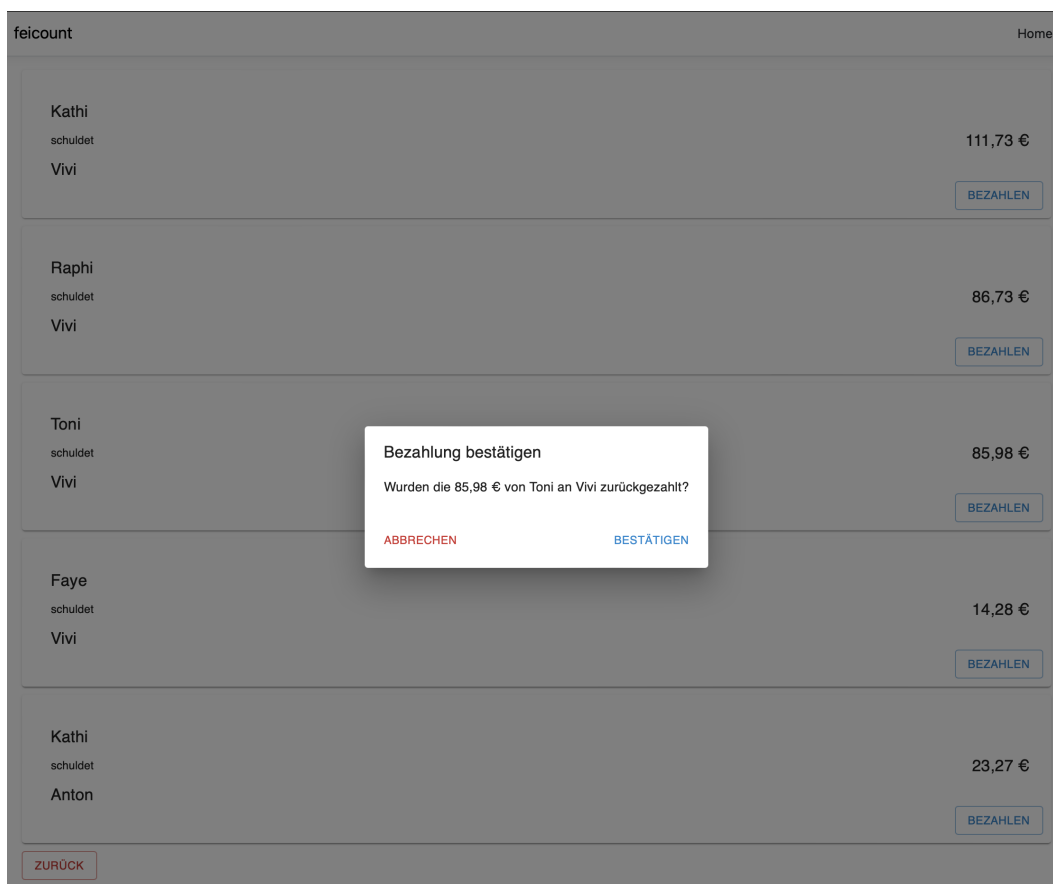


Abbildung 13: Dialogfenster zur Bestätigung, dass eine Transaktion bezahlt wurde.

3.4 Usability Testing

Nach der Konzeption und der Implementierung der Anwendung sollte exemplarisch getestet werden, ob die Anwendung die im Kapitel 2 definierten Anforderungen erfüllt. Als wichtigste Bedingung und notwendige Voraussetzung für den Erfolg des Forschungsvorhabens wurde zu Beginn die praktische Umsetzung einer Webanwendung mit React festgelegt, die von Nutzer:innen selbstständig genutzt werden kann.

Zusätzlich wurde als Ziel definiert, dass die Anwendung eine hohe Usability aufweisen soll. Neben einer Reihe von offenen Feedbackrunden und Klicktests mit Freunden und Bekannten, die an der Entwicklung des Projekts interessiert waren, wurde abschließend eine quantitative Evaluation des Systems durchgeführt. Auf die Evaluation und ihre Ergebnisse wird im Folgenden näher eingegangen.

3.4.1 Versuchsdurchführung

Im Zeitraum zwischen dem 28.02. und 06.03.2024 fand eine Befragung von Nutzer:innen statt, die das System zuvor getestet hatten. Insgesamt wurden vier Personen (zwei Frauen und zwei Männer) im Alter zwischen 25 und 28 Jahren befragt. Davon waren drei Studierende (Elektrotechnik und Medieninformatik) und eine Softwareberaterin.

Nach einer kurzen Einführung zum Zweck der Anwendung und der Erhebung der demografischen Daten wurden die Teilnehmenden gebeten, eigenständig einen Feicount und zugehörige Ausgaben anzulegen. Nebenbei sollten sie frei über ihre Beobachtungen und Erfahrungen im Umgang mit der Applikation sprechen (Vgl. *Thinking Aloud Method*, Nielsen, 1994). Alle Teilnehmenden konnten die Anwendung problemlos bedienen und die gestellten Aufgaben eigenständig lösen. Der Systemfluss wurde als leicht verständlich beschrieben und das Systemfeedback als hilfreich empfunden. Als positivster Aspekt wurde von allen Teilnehmenden genannt, dass die Anwendung im Vergleich zur echten Tricount-App sehr aufgeräumt wirkt und es wurde als besonders positiv empfunden, dass die Anwendung keine Werbung enthält.

Während der Tests wurden drei Bugs identifiziert. Zum einen fiel auf, dass die Breite der farbigen Balken nicht immer der Höhe des Saldos entsprach, was erfolgreich behoben werden konnte. Zum anderen traten Fehler bei der nachträglichen Bearbeitung von Teilnehmer:innen in einem Feicount auf, die aus Zeitgründen durch einen Hotfix umgangen werden konnten, bei

dem die nachträgliche Bearbeitung der Feicount-Nutzer:innen ausgeblendet wird. Das letzte Problem betrifft die Berechnungslogik der Salden in einem Feicount. Bei der Berechnung der Salden wird der Gesamtbetrag einer Ausgabe durch die Anzahl aller Empfänger:innen im Feicount geteilt. Da dabei für alle Teilnehmer:innen abgerundet wird, entstehen Rundungsfehler im Gesamtsaldo. Da diese Rundungsfehler im Cent-Bereich liegen und nichts mit der Implementierung des Systems und dem Erlernen von React oder .NET zu tun haben, wurde entschieden, diesen Fehler bei der Entwicklung eines ersten Systems zu vernachlässigen. Er wird jedoch in der weiteren Entwicklung als erstes behandelt.

Zur Überprüfung der Gebrauchstauglichkeit der Anwendung wurde beschlossen, die Testpersonen gemäß den ISO-Kriterien mithilfe des ISOMetric-Fragebogens (Willumeit, 1993) zu befragen. Dieser bietet eine umfassende, quantitative Perspektive auf die Gebrauchstauglichkeit des entwickelten Systems. Der Fragebogen operationalisiert die sieben Designkriterien (*task*, *self-descriptiveness*, *controllability*, *conformity*, *error tolerance*, *individualisation* und *learning*) durch die Bewertung von 75 Aussagen auf einer 5-Punkte-Skala von *trifft nicht zu* bis *trifft sehr zu*. Dieser Fragebogen wurde für die Evaluation ausgewählt, da die ISO-Kriterien international anerkannte und weit verbreitete Standards zur Bewertung von Benutzeroberflächen und Softwareanwendungen darstellen.

3.4.2 Ergebnisse und Diskussion

Die Testpersonen bewerteten verschiedene Aussagen zur Anwendung anhand der ISO-Kriterien auf einer 5-Punkte-Skala. Die beste Bewertung erhielt die Kategorie *learning* mit einem Mittelwert von 4,68. Die Kategorien *task* und *self-descriptiveness* erhielten ebenfalls hohe Bewertungen (jeweils 4,42). Die Kategorie *conformity* wurde mit einem Mittelwert von 4,41 und die Kategorie *controllability* mit einem Mittelwert von 4,3 bewertet. Die Kategorie *error tolerance* wurde mit einem Mittelwert von 3,82 etwas schlechter bewertet. Die Kategorie *individualisation* wurde mit einem Mittelwert von 3,04 am schlechtesten bewertet.

Die vorliegende Untersuchung zeigt, dass der entwickelte Prototyp die ISO-Kriterien in hohem Maße erfüllt. Das Ziel, eine benutzerfreundliche und leicht verständliche Plattform für das Finanzmanagement zu schaffen, scheint weitgehend erreicht worden zu sein. Die etwas niedrigere Bewertung in der Kategorie *error tolerance* weist darauf hin, dass bei der Fehlerbehandlung

und -vermeidung noch Verbesserungspotential besteht. Bei der Weiterentwicklung des Systems sollte auf eine konsequentere Überprüfung der Systemeingaben geachtet werden, um mögliche Fehlerquellen weiter zu minimieren. Auch im Bereich der Individualisierbarkeit des Systems könnten noch Verbesserungen vorgenommen werden, wie beispielsweise die Integration von individuellen Konten, bei denen persönliche Präferenzen (wie ein Farbschema oder weitere Anzeigeoptionen) hinterlegt werden können. Gleichzeitig würden damit nur noch die Feicounts des jeweiligen Accounts angezeigt werden.

4 Fazit und Ausblick

Im Rahmen dieses Semesterprojektes wurde eine eigenständige Alternative zu kommerziellen Anwendungen für das Finanzmanagement konzipiert und entwickelt. Dabei konnten wertvolle praktische Erfahrungen in der Entwicklung mit React und .NET gesammelt werden. Die klare Trennung von Frontend und Backend sowie die Verwendung bewährter Technologien trugen zur Effizienz und Skalierbarkeit des Systems bei. Darüber hinaus wurden persönliche Fähigkeiten im Bereich des eigenständigen Projektmanagements und der kreativen Lösungsfindung gestärkt.

Usability-Tests des Systems ergaben positive Ergebnisse hinsichtlich der Benutzerfreundlichkeit und der intuitiven Bedienbarkeit der Anwendung. Die Untersuchung anhand der ISO-Kriterien ermöglichte eine fundierte Bewertung, wobei insbesondere die Kategorien *Lernen*, *Aufgabe* und *Selbstbeschreibungsfähigkeit* positiv hervorstachen. Bei einer Weiterentwicklung des Systems könnte die Anwendung vor allem in den Bereichen *Fehlertoleranz* und *Individualisierung* weiter optimiert werden.

Das hier vorgestellte System stellt den Kern und die wichtigsten Komponenten einer funktionierenden Anwendung für die Verwaltung gemeinsamer Ausgaben in einer Gruppe dar. In der weiteren Entwicklung könnten individuelle Konten für die Nutzerinnen und Nutzer erstellt werden, die sicherstellen, dass eine Person nur Zugriff auf ihre eigenen Feicounts hat. Damit einher ginge eine effiziente Umsetzung zum Teilen bzw. Beitreten bestehender Feicounts, bei der die User-Instanzen der Nutzer:innen des Systems und der Teilnehmer:innen in einem Feicount einander klar zugeordnet sind.

Literatur

- Charboneau, T. (2022, Oktober). How to Use the Postgres Docker Official Image | Docker. <https://www.docker.com/blog/how-to-use-the-postgres-docker-official-image/>
- jwoolard. (2009, Mai). What algorithm to use to determine minimum number of actions required to get the system to SZeroState? <https://stackoverflow.com/questions/877728/what-algorithm-to-use-to-determine-minimum-number-of-actions-required-to-get-the/>
- MUI. (n. d.). MUI: The React component library you always wanted. <https://mui.com/>
- Nielsen, J. (1994). *Usability Engineering*. Elsevier Science. <https://books.google.de/books?id=95As2OF67f0C>
- Tricount - Organisieren von Gruppenausgaben. (n. d.). <https://www.tricount.com/de/>
- Tutorial: Tic-Tac-Toe – React. (n. d.). <https://react.dev/learn/tutorial-tic-tac-toe>
- Using TypeScript – react. (2024). <https://react.dev/learn/typescript>
- Wikipedia. (2024, Januar). Partition problem. https://en.wikipedia.org/wiki/Partition_problem
- Willumeit, H. (1993). IsoMetricsS, Fragebogen zur Evaluation von graphischen Benutzungsschnittstellen (Kurz-Version) [Publisher: Universität Osnabrück, Fachbereich Psychologie].

A Anhang

Name:	Feicount	<-- aktualisieren Sie hier den Namen des evaluierten Systems							
	Mittelwert gesamt (automatisch b	Mittelwerte je User (automatisch berechnet):							
A: task	4,416666667	4,266666667	4,066666667	4,733333333		4,6			
S: self-descriptiv	4,416666667	4,333333333	4,25	4,916666667		4,166666667			
T: controllability	4,295454545	3,727272727	4,181818182	4,818181818		4,454545455			
E: conformity	4,40625	3,625	4,75	4,875		4,375			
F: error toleranc	3,819047619	3,133333333	4	4		4,142857143			
I: individualisati	3,041666667	2,5	1,666666667	4,166666667		3,833333333			
L: learning	4,6875	4,75	4,625	4,875		4,5			
Tragen Sie hier bitte die Werte aus den Fragebögen ein; die grauen Felder dürfen nicht verändert werden									
keine Antwort wird mit Null (0) k									
Die Daten der User 1-3 (Spalten C, F und G) sind Beispieldaten; bitte löschen Sie diese, bevor Sie Ihre Daten auswerten									
Item:	Pollung\User	User 1	1 corrected	User 2	2 corrected	User 3	3 corrected	User 4	4 corrected
A1	-1	2	4	2	4	1	5	1	5
A3	1	3	3	5	5	5	5	5	5
A4	1	5	5	4	4	5	5	5	5
A6	1	5	5	5	5	5	5	5	5
A7	1	5	5	3	3	5	5	5	5
A8	-1	1	5	4	2	1	5	2	4
A9	1	4	4	5	5	3	3	5	5
A10	1	5	5	5	5	5	5	4	4
A11	1	5	5	4	4	5	5	5	5
A12	1	3	3	4	4	5	5	5	5
A14	1	1	1	3	3	3	3	3	3
A15	1	5	5	3	3	5	5	4	4
A16	1	4	4	5	5	5	5	4	4
A17	1	5	5	4	4	5	5	5	5
A18	1	5	5	5	5	5	5	5	5
S2	1	5	5	2	2	5	5	4	4
S3	1	5	5	5	5	5	5	4	4
S5	1	4	4	5	5	5	5	5	5
S6	1	3	3	4	4	5	5	3	3
S7	1	2	2	2	2	5	5	3	3
S8	1	5	5	5	5	5	5	4	4
S9	1	4	4	4	4	5	5	4	4
S10	1	5	5	5	5	5	5	5	5
S11	1	5	5	5	5	5	5	4	4
S12	1	5	5	5	5	5	5	5	5
S13	1	4	4	4	4	4	4	5	5
S14	1	5	5	5	5	5	5	4	4
T2	1	4	4	4	4	5	5	4	4
T3	1	4	4	5	5	5	5	5	5
T4	1	2	2	5	5	5	5	5	5
T5	1	4	4	5	5	5	5	5	5
T6	1	5	5	4	4	5	5	4	4
T7	1	4	4	5	5	5	5	5	5
T8	1	5	5	3	3	5	5	5	5
T10	1	4	4	4	4	5	5	5	5
T12	-1	2	4	4	2	1	5	3	3
T13	1	1	1	5	5	3	3	3	3
T15	1	4	4	4	4	5	5	5	5
E8	-1	1	5	2	4	1	5	1	5
E1	1	4	4	5	5	4	4	3	3
E2	1	3	3	5	5	5	5	5	5
E3	1	4	4	5	5	5	5	4	4
E4	1	3	3	4	4	5	5	5	5
E5	1	3	3	5	5	5	5	5	5
E6	1	4	4	5	5	5	5	5	5

E7	1	3	3	5	5	5	5	3	3
F1	-1	5	1	2	4	3	3	1	5
F2	1	3	3	3	3	4	4	4	4
F3	1	4	4	0	0	5	5	4	4
F4	1	3	3	0	0	3	3	3	3
F5	1	3	3	0	0	5	5	5	5
F6	1	3	3	4	4	4	4	4	4
F7	-1	1	5	0	0	2	4	0	0
F8	1	4	4	0	0	4	4	5	5
F9	1	4	4	5	5	5	5	5	5
F10	1	3	3	0	0	3	3	3	3
F12	1	3	3	0	0	4	4	5	5
F13	1	3	3	0	0	5	5	5	5
F14	-1	3	3	0	0	2	4	4	2
F15	1	2	2	0	0	4	4	3	3
F16	1	3	3	0	0	3	3	5	5
I1	1	2	2	3	3	4	4	5	5
I4	1	2	2	3	3	4	4	5	5
I6	1	4	4	1	1	4	4	3	3
I7	1	1	1	1	1	5	5	3	3
I8	1	3	3	1	1	5	5	3	3
I11	1	3	3	1	1	3	3	4	4
L1	-1	1	5	1	5	1	5	1	5
L2	1	5	5	5	5	5	5	5	5
L3	1	4	4	2	2	5	5	5	5
L4	1	5	5	5	5	5	5	1	1
L5	1	5	5	5	5	5	5	5	5
L6	1	4	4	5	5	5	5	5	5
L7	-1	1	5	1	5	1	5	1	5
L8	1	5	5	5	5	4	4	5	5

Abbildung 14: Ergebnisse der Usability Untersuchung anhand des ISOMetric-Fragebogens.