HAW HAMBURG

BACHELOR THESIS
Soheil Nazari

Robusteres State Management in Frontend Webapplikationen mit DFA Übergängen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Soheil Nazari

Robusteres State Management in Frontend Webapplikationen mit DFA Übergängen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Wirtschaftsinformatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr.-Ing. Lars Hamann

Eingereicht am: 13. Januar 2025

Soheil Nazari

Thema der Arbeit

Robusteres State Management in Frontend Webapplikationen mit DFA Übergängen

Stichworte

State Management, Webapplikationen, Frontend

Kurzzusammenfassung

Arthur Dents Reise in eine neue Zukunft ...

Soheil Nazari

Title of Thesis

Making State Management in Frontend Web Applications Robuster with DFA Transitions

Keywords

State Management, Web Applications, Frontend

Abstract

Arthur Dents travel to a new future ...

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Die Rolle des State-Managements in Frontend Webapplikationen

Moderne Webseiten folgen dem Single Page Appliaction (SPA) Ansatz. Dem nach bleibt die gleiche Instanz der Webapplikation solange der Nutzer auf der Webseite ist, bestehen. In der Regel sind mehrere Teile einer Applikation, beispielsweise bei der Komponenten-Architektur, von gleichen Daten abhängig. Außerdem werden die Daten basierend auf Interaktionen des Benutzers modifiziert. Änderungen in den Daten müssen den betroffenen Komponenten mitgeteilt werden. In einigen Fällen ist die Synchronisierung der Daten im Frontend mit den Daten des Servers erforderlich. Um HTTP Aufrufe zu sparen, können verschiedene Mechanismen, wie beispielsweise Caching oder Debouncing verwendet werden. Diese Faktoren erhöhen, die ohnehin schon hohe Komplexität und Fehleranfälligkeit zusätzlich.

Um diese Komplexität effizient zu verwalten, werden State-Management Lösungen wie Redux, NgRx, Zustand oder Pinia verwendet. Mit Hilfe dieser Open Source JavaScript Bibliotheken, können Daten beim Bedarf von einer API abgerufen, transformiert und gespeichert werden. Die meisten State-Management Bibliotheken sind eng mit einem UI-Framework gekoppelt. Aus diesem Grund sind sie ein fundamentaler Baustein jeder größeren Frontend Webapplikation.

1.2 Ziel der Arbeit

Mit der Komplexität erhöht sich auch die Fehleranfälligkeit. Fehler im Zustand, also Daten der Applikation, haben einen direkten Einfluss auf das Angezeigte. Wenn die Applikation sich in einem „falschen“ Zustand befindet und es keine Laufzeitfehler gab, kön-

nen die Verantwortlichen (in der Regel, die Entwickler) unter Umständen, nicht darüber informiert sein. Dies führt zu langlebigen Bugs.

Ziel dieser Arbeit ist es, einen Ansatz zu erarbeiten, bei dem die Möglichkeit eines Befindens in einem „falschen“ oder „illegalem“ Zustand eliminiert wird. Dazu wird jeder zusammenhänge Teil des Zustands als ein endlicher Automat abgebildet. Dahingehend wird jede Änderung in diesem Zustand wie ein Übergang bei einem endlichen Automaten behandelt. Es wird vorgeschlagen die beliebten State-Management Lösungen um „strikte“ Übergänge, wie bei einem DFA, zu erweitern. Auf diesem Wege wird eine Reduzierung von Bugs in größeren Applikationen bestrebt. Dabei wird insbesondere auf die Lesbarkeit und Wartbarkeit des Quellcodes und die Developer Experience geachtet.

Folgende Forschungsfragen werden behandelt:

1. Können Bugs, die Aufgrund eines falschen Zustandes entstehen, mit Hilfe von „strikten“ Übergängen reduziert werden?
2. Steigt oder sinkt die Developer-Experience?
3. Steigt oder sinkt die Lesbarkeit und Wartbarkeit des Codes?

1.3 Aufbau

In dieser Arbeit werden die bestehende SM-Ansätze um Übergänge wie bei einem DFA erweitert. Um dies zu erreichen, ist es notwendig die Funktionsweise bestehender Ansätze zu kennen. Diese werden im Kapitel 3 aufgeführt. Anschließend werden die DFA-Übergänge angepasst auf Anwendungfall einschließlich der JavaScript API zur Definition im Kapitel 4 dargestellt. Die Erkenntnisse aus Kapitel 3 und 4 werden kombiniert, um zwei konkrete Implementierungen für redux und pinia zu zeigen. Abschließend werden diese im Kapitel 5 verglichen.

2 Methodologie

TODO

2.1 Code Ausschnitte

TypeScript wird benutzt um, Aufbau von Objekten oder Funktionen zu beschreiben. Längere Strukturen werden mit Hilfe von Code-Bespielen veranschaulicht. Hierfür wird ebenfalls TypeScript verwendet. An viele Stellen wird auf Type-Annotationen verzichtet, damit die Beispiele leicht lesbar bleiben.

3 State-Management Ansätze

Bei den populären SM Lösungen folgt Redux und NgRx dem Flux-Pattern[3][1], wobei Zustand und Pinia einen anderen, Framework-nahen Ansatz verfolgen. Im Folgenden wird die Funktionsweise und die Eigenschaften von Redux und Pinia näher beschrieben. Da diese grundlegend unterschiedliche Ansätze verfolgen und andere SM-Lösungen sich einem der beiden ähneln.

3.1 Redux

Redux definiert sich durch folgenden vier Eigenschaften:

1. Unveränderlichkeit (Immutability): Änderung am State sind ausschließlich über die APIs von Redux unter Beachtung der Unveränderlichkeit möglich.
2. Zentralisierung des Zustandes: Der gesamte Applikationszustand lebt in einem zentralen JavaScript Objekt.
3. Nachvollzierbarkeit (Traceability): Während der gesamten Lebensdauer der Applikation, sind Änderungen am Zustand auf deren Ursprung verfolgbar.
4. Event basiert: Es wird das Beobachter-Muster (Observer Pattern) verwendet.

Das Verhalten des Stores wird durch *actions* und *reducer* definiert.

3.1.1 Actions

Eine Aktion (Action) beschreibt eine Änderung oder Interaktion in und mit der Applikation. Beispielsweise könnte eine *counter-clicked* Action versendet (dispatch) werden, wenn der Nutzer auf den *Zähler erhöhen* Button drückt. Oder, wenn der Nutzer sich

erfolgreich angemeldet hat, kann eine entsprechende Action versendet werden. Intern ist eine Action ein POJO.[4]

Es wird folgende Struktur für Actions empfohlen:

```
type Action<T> = {
  type: string,
  payload?: T
}
```

Das Feld *type* beschreibt die Action und das optionale Feld *payload* enthält weiterführende Daten.

3.1.2 Reducer

Ein Reducer ist für die Initialisierung und Aktualisierung des Zustandes zuständig. Ein Reducer wird als eine Pure-Function mit zwei Parametern definiert. Der erste Parameter ist das Zustandsobjekt und der zweite die versendete Action. Der Rückgabewert dieser Funktion ist das neue Zustandsobjekt. Da es sich hier um eine Pure-Funtion handelt, dürfen es hier keine Seiteneffekte stattfinden. Wie anfangserwähnt, ist der Zustand Unveränderlich, daher dürfen hier keine direkten Veränderungen des Zustandes stattfinden. Es wird lediglich ein neues Objekt zurückgegeben. Fall es keine Veränderungen stattfinden sollen, kann das ursprüngliche Objekt aus dem ersten Parameter unverändert zurückgegeben werden.[4]

Es wird folgende Struktur für Reducer empfohlen:

```
type Reducer<S, A> = (state: S, action: A) => S
```

Beispiel reducer:

```
function reducer(state = { user: null }, action) {
  switch (action.type) {
    case 'user-logged-in':
      return {
        ...state,
        user: {
          userId: action.payload.userId
        },
```

```
      }
    case 'user-logged-out':
      return {
        ...state,
        user: null,
      }
    default:
      return state
  }
}
```

Es wird die *Spread Syntax: ...* aus ECMAScript 6 genutzt, um as ursprüngliche Zustandsobjekt zu klonen.[2]

3.1.3 Definition und Interaktion mit dem Store

Der Store wird mit Hilfe der *createStore* API erstellt. Als Parameter wird die Reducer-Function übergeben. Der Rückgabewert ist das Store-Objekt. Dieses bietet Zugang zu unteranderem *dispatch* und *getState* Methoden. Mit diesen kann jeweils Actions versendet und aus dem Store gelesen werden.

```
import { createStore } from 'redux'

const store = createStore(reducer)
store.dispatch(action)
const user = store.getState().user
```

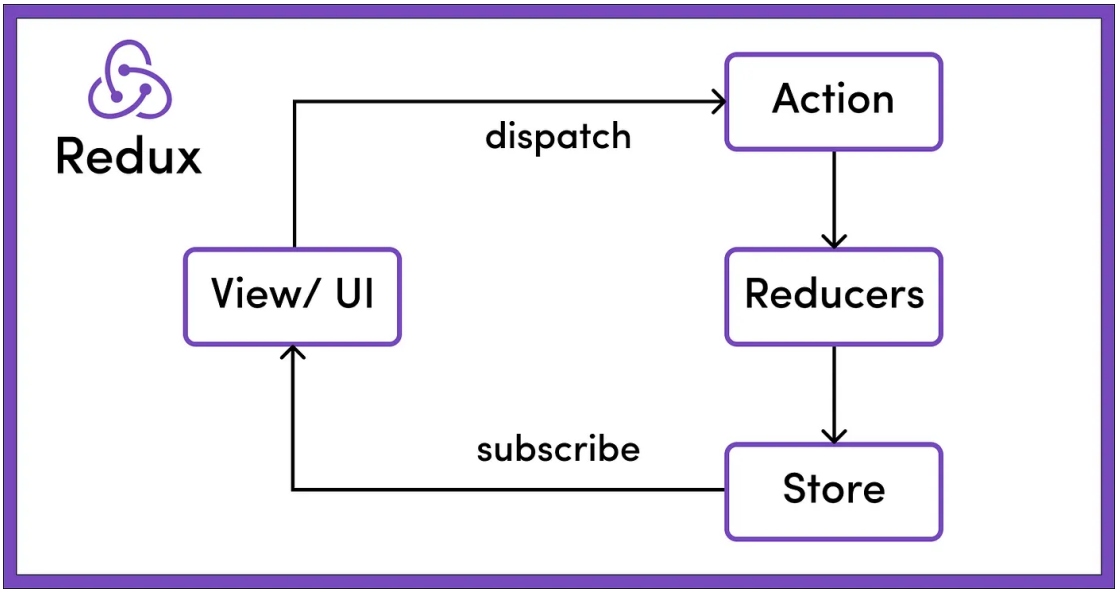

Abbildung 3.1: Redux Datenfluss

3.2 Pinia

Pinia ist sehr eng gekoppelt mit dem Vue Framework und nutzt dessen Mechanismen der Reaktivität zu Datenhaltung. Das führt dazu, dass Pinia selbst minimal bleibt und die Daten ohne weiteres reaktiv sind. Im Gegensatz zu Redux und NgRx setzt diese Store-Lösung nicht das Flux-Pattern um. Dank dieser Praxis, ist weniger Code nötig um einen Store zu definieren. Außerdem folgt Pinia nicht den Single-Store-Ansatz, bei dem alle Daten in einem zentralen Objekt leben. Sondern sind für Teile der Daten eigenständige Store-Instanzen zuständig. Pinia bietet zwei verschiedene APIs zu Definition von Stores an. In dieser Arbeit wird die *Options API* verwendet. Die Konzepte lassen sich auch auf die *Composition API* übertragen.[9] Die zwei essentiellen Konzepte sind *State* und *Action*.

3.2.1 State

State ist eine Funktion, die ein Objekt, das den Zustand beschreibt zurückgibt.

3.2.2 Action

Eine Action ist eine Methode, die den State verändert und in einem *actions* Objekt definiert wird.

3.2.3 Definition eines Stores

Zu Definition eines Stores wird die *defineStore* API genutzt. Als Parameter wird ein eindeutiger Name und eine Beschreibung des Stores in Form eines Objekts übergeben. In dem zweiten Paramter werden die Felder *state* und *actions* definiert.

```
const useUserStore = defineStore('user-store', {
  state: () => {
    user: null
  },
  actions: {
    updateUser(newUser) {
      this.user = newUser
    }
  }
})
```

Auf die Felder in dem State-Objekt wird in einer Action mit *this* zugegriffen. Das State-Objekt wird seitens Pinia intern jeder Action gebunden.

3.2.4 Interaktion mit dem Store

Der Store kann in einer belibiegen Vue-Komponente importiert werden. Die Felder des Objekts, das von der state Funktion zurückgegeben wird, werden automatisch zu Feldern des Store Objekts. Genauso werden auch die Methoden des Actions-Objekts auch zu Member des Store Objekts.

```
const userStore = useUserStore()

// userStore.user
// userStore.updateUser
```

Die State im oberen Beipiel ist reaktiv und kann als *userStore.user* im Template der Komponente referenziert werden. Die Desktrukturierung (destructuring) des Store-Objekts, im oberen Beispiel *userStore*, führt zur Verlust der Reaktivität. Aus diesem Grund wird die Punktnotation empfohlen.[9]

4 Der neue Ansatz

4.1 Steigende Robustheit durch TypeScript

TypeScript verfügt, im Gegensatz zu JavaScript, über statische Typisierung. Dank der statischen Typisierung sind statische Typeanalysen und Operationen wie *Go to Definition* und *Go to Implementations* der Entwicklungsumgebungen (IDE) möglich. Diese Eigenschaften reduzieren Fehler im Zusammenhang mit falschen Typen erheblich. Wie in 4.1 abgebildet, wird TypeScript von immer mehr Entwicklern genutzt, während die JavaScript Nutzung abnimmt. 4.1 beinhaltet die tatsächliche Nutzung von TS nicht. Der TypeScript Compiler (tsc) ist in den meisten modernen IDEs, wie Visual Studio Code und den JetBrains IDEs wie IntelliJ IDEA und WebStorm integriert. Dies führt dazu, dass man auch beim JavaScript Code einige Vorteile von TypeScript bekommt.[8]

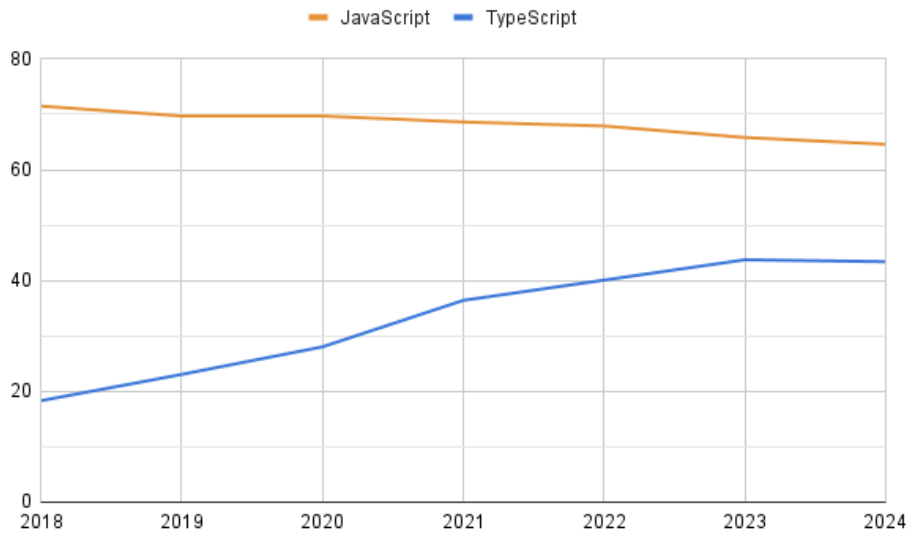

Abbildung 4.1: Prozentuale Nutzung von JavaScript und TypeScript unter professionellen Entwicklern von 2018 bis 2024

4.2 Unzureichende Robustheit in State-Management

Der TypeScript-Faktor macht Webapplikationen, somit auch State-Management auf Typ-Ebene robuster und weniger fehleranfällig. Allerdings ist es für die Applikation immer noch möglich in einem falschen Zustand zu sein. Gegeben sei ein Redux Store, der für das Speichern einer Liste von *items* zuständig ist. Definiert wird der Store wie folgt:

```
type FetchAction = {
  type: 'fetch'
}

type FetchSuccessfulAction = {
  type: 'fetch-successful',
  payload: Array<any>
}

type FetchFailedAction = {
  type: 'fetch-failed',
  payload: Error
}

type Action =
  | FetchAction
  | FetchSuccessfulAction
  | FetchFailedAction

function reducer(
  state = { items: 'not-fetched' },
  action: Action
) {
  switch (action.type) {
    case 'fetch':
      return {
        ...state,
        items: 'fetching'
      }
```

```
      case 'fetch-successful':
        return {
          ...state,
          items: action.payload
        }
      case 'fetch-failed':
        return {
          ...state,
          items: action.payload
        }
      default:
        return state
    }
  }

  const store = createStore(reducer)
```

Es ist erlaubt, die *FetchSuccessfulAction* Aktion zu versenden, ohne voher die *Fetch* Aktion versendet zu haben. Das heißt: „*items* wurden erfolgreich abrufen“, ohne die Anfrage zuvor gemacht zuhaben. Seitens Redux ist das Versenden einer Aktion immer, unbeachtet des aktuellen Zustandes, erlaubt. Dieser Faktor spricht gegen die Nachvollziehbarkeit und gilt für alle populäre SM Lösungen.

4.3 Robustere Zustandsübergänge

Es wird vorgeschlagen den Applikationszustand wie ein Zustand eines DFAs zu behandeln. Im Falle von Redux werden die Aktionen als Übergänge und der Reducer als die Übergangsfunktion eines DFAs gesehen. Restliche Eigenschaften des Quintupels eines DFAs werden hierbei ignoriert.

Um die Übergangsfunktion zu definieren, wird pro Zustand eine *Übergangsliste* aller Aktionen benötigt, die bei diesem Zustand erlaubt sind. Ein Problem hierbei ist allerdings, dass die Identifizierbarkeit der einzelnen Zustände nicht garantiert ist. Abweichend von DFAs, sind die Zustände in Webapplikationen nicht immer serialisierbar. Nicht serialisierbare Objekte sind nicht immer identifizierbar. Im oberen Beispiel, sind die Zustände

'not-fetched' und *'fetching'* vom Typ String und somit serialisierbar, allerdings sind die restlichen Zustände nicht serialisierbar (Zustand vom Typ Error und Array<any>). Um dieses Problem zu umgehen, wird eine *Identitätsfunktion* emfohlen, um zwischen verschiedenen Zuständen zu unterscheiden. Sie akzeptiert als Paramter den aktuellen Zustand und gibt ein Boolean zurück.

```
type IdentityFn<S> = (state: S) => boolean
```

Mit dieser Funktion, kann der Anwender für die Indentifizierbarkeit der Zustände sorgen. Bei JavaScript Klassen, kann der *instanceof* Operator genutzt werden, um auf die Instanz einer Klasse wie *Error* zu prüfen.[7] Desweiteren, können bei Objekten auf eindeutige Eigenschaften, wie die Präzens eines Feldes per *in* Operator geprüft werden.[6] Bei Arrays kann die *Array.isArray* Funktion verwendet werden.[5] Durch die Kombinition dieser und weiteren Funktionen und Operatoren können weitere Datentypen und Fälle identifiziert werden.

Die *Übergangsliste* lässt sich in einer Map wie folgt speichern:

```
type TransitionMap<S extends IdentityFn<S>, A> = Map<S, Array<A>>
```

Für die *Übergangsmap* gilt: *Identitätsfunktion* ist der Schlüssel, während Liste von Aktionen der Wert ist.

In der Übergangsfunktion wird der Zustandswechsel mit einer *Validierungsfunktion* validiert. Diese prüft mit Hilfe der *Übergangsmap* auf die Gültigkeit des Übergangs und wirft einen Laufzeitfehler bei ungültigen Aufrufen. Falls der Übergang gültig ist, wirft sie keinen Fehler und der Zustandswechsel kann stattfinden. Gültig ist der Übergang, wenn es für den aktuellen Zustand eine Identitätsfunktion gibt, die wahr zurückgibt und die aktuelle Aktion in der zugehörigen Liste enthalten ist. In allen anderen Fällen ist der Übergang ungültig. Der Laufzeitfehler sorgt dafür, dass der ungültige Aufruf berichtet wird und sich nicht zu einem langlebigen Bug entwicklen kann.

Die Validerungsfunktion V ist wie folgt definiert:

```
type ValidationFn<S, A> = (
    transitionMap: Map<S, A>,
    state: S,
    action A
  ) => boolean
```

4.3.1 Vorteile

1. **Übersichtlichkeit**: Damit die Validierung funktioniert, ist der Entwickler gezwungen die Übergangsmap zu definieren. So können fehlerhafte und überflüssige Übergänge schneller erkannt und korrigiert werden.
2. **Erkennung von Bugs**: Bei fehlgeschlagener Übergangsvalidierung wird ein Laufzeitfehler geworfen, der über die Monitoringsysteme die Entwickler über einen Bug informieren kann. Ebenfalls möglich ist es, die falschen Übergänge lediglich zu loggen. Auf diesem Wege können die Entwickler ebenfalls über den Bug in Kenntnis gesetzt werden. Die letztere Strategie erlaubt jedoch im Worst-Case Weiterausführung falscher Geschäftslogik.

4.3.2 Nachteile

1. **Mehr Aufwand**: Damit der Ansatz funktioniert muss die Übergangsmap definiert werden. Diese Voraussetzung kostet zusätzliche Aufwand.
2. **Erhöhte Ausfürungszeit**: Außerdem erhöht sich Ausführungszeit der gesamten Applikation durch die Validerung bei jedem Zustandswechsel. Diese hinzukommende Zeit ist jedoch zuvernachlässigen, wenn die Identitätsfunktion effizient ist und keine Nebeneffekte erzeugt, also eine *pure function* ist.

4.4 Implementierung

4.4.1 Redux

Im folgenden wird die Implementierung der obengenannten Funktionen und Konzepte für redux gezeigt.

Die TransitionMap ist die Grundlage des Ansatzes. Mit Blick auf die Developer Experience und die Lesbarkeit wird die TransitionMap als ein Array von Objekten mit zwei Feldern definiert. Nämlich *identityFn* und *actionTypes*:

```
type Transition<S> = {
  identityFn: (state: S) => boolean
  actionTypes: string[]
}

type Transitions<S> = Transition<S>[]
```

Im folgenden Beispiel wird die TransitionMap definiert.

```
type State = 'not-fetched' | 'fetching' | string[] | Error

type Action =
  | {
      type: 'fetch'
    }
  | {
      type: 'fetch-successful'
      payload: string[]
    }
  | {
      type: 'fetch-failed'
      payload: Error
    }

const transitions = [
  {
    identityFn: (state) => state === 'not-fetched',
    actionTypes: ['fetch'],
  },
  {
    identityFn: (state) => state === 'fetching',
    actionTypes: ['fetch-successful', 'fetch-failed'],
  },
]
```

Der Reducer kann wie von redux vorgegeben definiert werden:

```
function reducer(state = 'not-fetched', action) {
  switch (action.type) {
    case 'fetch':
      return 'fetching'
    case 'fetch-successful':
      return action.payload
    case 'fetch-failed':
      return action.payload
    default:
      return state
  }
}
```

Die definierten Übergänge werden mit Hilfe der folgenden Validierungsfunktion validiert:

```
function validateTransition(state, action, transitions) {
  for (const transition of transitions) {
    if (transition.identityFn && transition.identityFn(state)) {
      if (transition.actionTypes.includes(action.type)) {
        return
      }

      throw new IllegalTransitionError(state, action.type)
    }
  }

  throw new TransitionNotFoundError(state)
}
```

Die Beiden Laufzeitfehler *IllegalTransitionError* und *TransitionNotFoundError* erben von der *Error* Klasse und dienen der Unterscheidbarkeit.

Damit die Validerungsfunktion bei jedem Zustandswechsel ausgeführt wird, muss die Übergangsfunktion, bei redux *Store.dispatch*, überschrieben werden.

```
function dispatchTransition(this, action) {
  validateTransition(this.getState(), action, this.transitions)
```

```
  this.dispatch(action)
}
```

Die neue *dispatchTransition* Funktion wird dem *Store.dispatch* Feld zugewiesen. Hierfür wird eine neue API *createTransitionStore* eingeführt, die wie folgt implementiert ist.

```
function createTransitionStore<S>(
  transitions: Transitions<S>,
  ...args: Parameters<typeof createStore>
): TransitionStore<S> {
  const store = createStore(...args)

  Object.defineProperty(store, 'transitions', {
    value: transitions,
  })

  Object.defineProperty(store, 'validateTransition', {
    value: validateTransition,
  })

  Object.defineProperty(store, 'dispatchTransition', {
    value: dispatchTransition,
  })

  return store as TransitionStore<S>
}
```

Die *createTransitionStore* API mit dem zusätzlichen TransitionsMap Parameter erstellt einen Store mit der *createStore* API von redux und fügt dem Store drei neue Felder, die TransitionMap, Validierungsfunktion und die dispatchTransition Methode hinzu.

```
type TransitionStore<S, A> = {
  validateTransition: (
    state: S,
    action: A,
    transitions: Transitions<S>
```

```
  ) => void
  transitions: Transitions<S>
  dispatchTransition: (
    this: TransitionStore<S, A>,
    action: BasicAction
  ) => void
} & Store
```

4.4.2 Pinia

Pinia verfügt über ein Plugin System. Über dieses bekommt man unter anderem Zugriff auf die Zustände und Actions der aktiven Stores. Die Plugins werden beim Start der Applikation über *Pinia.use* API registriert.

Die Übergangsmap hat die gleiche Struktur wie die bei Redux. Allerdings wird das Feld *actionTypes* zu *action* umbenennt, da die Actions hierbei keine POJOs sind und stehen für den Namen der Action in dem *actions* Objekt.

Das Plugin wird mit der *transitions* Funktion instaniziert, diese nimmt eine Map mit der StoreId als Schlüssel und die Übergangsmap als Wert für den jeweiligen Store:

```
type transitions<S> = (
  transitionsByStoreId: TransitionsByStoreId<S>
) => PiniaUseCallback

type TransitionsByStoreId<S> = {
  [storeId: string]: Transitions<S>
}
```

Die transitions Funktion gibt eine Anonymefunktion zurückt, diese bekommt ein Objekt als Parameter. In diesem befindet sich unter anderem das Store Objekt. Über die *Store.$onAction* kann eine Callbackfunktion als Preproccessor für Actions registriert werden. Die Callbackfunktion bekommt ein Objekt als Parameter. In diesem sind unter anderem der Name der aktuellen Actions und der aktuelle Zustand. In dem Preproccessor wird die Validierungsfunktion aufgerufen.

```
type PiniaUseCallback = Parameters<
  ReturnType<
    typeof createPinia
  >['use']
>[0]
type PiniaUseCallbackArgs = Parameters<PiniaUseCallback>[0]

function transitions<S>(
  transitionsByStoreId: TransitionsByStoreId<S>
): PiniaUseCallback {
  return ({ store }: PiniaUseCallbackArgs) => {
    const transitions = transitionsByStoreId[store.$id]

    if (!transitions) {
      return
    }

    store.$onAction(({ name, store }) => {
      validateTransition(store.items, name, transitions)
    })
  }
}
```

Die Validerungsfunktion ist auf die leicht geänderte Struktur der Actions angepasst.

```
function validateTransition<S, A extends string>(
  state: S,
  action: A,
  transitions: Transitions<S>
): void {
  for (const transition of transitions) {
    if (transition.identityFn && transition.identityFn(state)) {
      if (transition.actions.includes(action)) {
        return
      }

      throw new IllegalTransitionError(state, action)
```

```
    }
  }

  throw new TransitionNotFoundError(state)
}
```

5 Vergleich der Ansätze

Literaturverzeichnis

[1] (BRANDONROBERTS), Brandon R.: *Getting Started*. 2024. – URL https://ngrx.io/guide/store. – official documentation

[2] (JOSH-CENA), Joshua C.: *Spread syntax (...)*. 2024. – URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax. – official documentation

[3] MARK ERIKSON (MARKERIKSON), Eng Zer Jun (.: *A (Brief) History of Redux*. 2023. – URL https://redux.js.org/understanding/history-and-design/history-of-redux. – official documentation

[4] (MARKERIKSON), Mark E.: *Redux Fundamentals, Part 3: State, Actions, and Reducers*. 2024. – URL https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers. – official documentation

[5] MDN, Joshua Chen (Josh-Cena): *Array.isArray()*. 2023. – URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray. – official documentation

[6] MDN, Joshua Chen (Josh-Cena): *in*. 2024. – URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in. – official documentation

[7] MDN, Joshua Chen (Josh-Cena): *instanceof*. 2024. – URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof. – official documentation

[8] ORIGINS, OfferZen: *TypeScript Origins: The Documentary*. 2023. – URL https://youtu.be/U6s2pdxebSo?si=BKrgCSGIzmSf4GeB. – Documentary with creators of TypeScript at Microsoft

[9] (POSVA), Eduardo San Martin M.: *Defining a Store*. 2024. – URL https://pinia.vuejs.org/core-concepts/. – official documentation

A Anhang

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
LaTeX	Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original