HAW HAMBURG

BACHELOR THESIS
Soheil Nazari

# Erhöhung der Korrektheit des States in Frontend Webapplikationen mit Strikten 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

# Erhöhung der Korrektheit des States in Frontend Webapplikationen mit Strikten 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**

Erhöhung der Korrektheit des States in Frontend Webapplikationen mit Strikten Übergängen

**Stichworte**

State-Management, Webapplikationen, Frontend

**Kurzzusammenfassung**

Frontend-Applikationen sind ein wesentlicher Bestandteil jeder Webapplikation und sind für Teile der Geschäftslogik und die UI verantwortlich. Steigende Anforderungen in Geschwindigkeit, Responsiveness und Features erhöhen die Komplexität enorm. Um einen Teil dieser Komplexität zu verwalten, kommen State Management-Lösungen zum Einsatz. Diese übernehmen wichtige Aufgaben wie beispielsweise das Data-Fetching, die Datentransformation und die Datenspeicherung. Fehler im State können daher einen verhältnismäßig großen Einfluss auf das Nutzererlebnis und das operative Geschäft haben. Damit Fehler und Defekte in diesem Bereich reduziert und schnell erkannt werden, wird eine strikte Erweiterung für State Management im Allgemeinen vorgestellt und mit dem normalen Ansatz verglichen.

**Soheil Nazari**

**Title of Thesis**

Increasing Correctness in State of Frontend Web Applications with Strict Transitions

**Keywords**

State Management, Web Applications, Frontend

**Abstract**

Frontend applications are an essential part of any web application and are responsible for parts of the business logic and the UI. Increasing demands for speed, responsiveness,

and features significantly raise complexity. To manage part of this complexity, state management solutions are used. These handle important tasks such as data fetching, data transformation, and data storage. Errors in the state can, therefore, have a relatively large impact on the user experience and operational business. To reduce and quickly detect errors and defects in this area, a strict extension for state management, in general, is introduced and compared to the conventional approach.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungen

**DFA** Deterministischer endlicher Automat.

**DX** Developer Experience.

**IDE** Integrated Development Environment.

**JS** JavaScript.

**LOC** Lines of Code.

**POJO** Plain Old JavaScript Object.

**SM** State Management.

**SPA** Single Page Application.

**ST** Strict Transitions.

**TS** TypeScript.

**tsc** TypeScript Compier.

# 1 Einleitung

## 1.1 Die Rolle des State-Managements in Frontend-Webapplikationen

Moderne Webseiten folgen dem Single Page Application-Ansatz. Demnach bleibt die gleiche Instanz der Webapplikation bestehen, solange der Nutzer auf der Webseite ist. In der Regel sind mehrere Teile einer Applikation, beispielsweise bei der Komponenten-Architektur, von denselben 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. Mithilfe dieser Open-Source-JavaScript-Bibliotheken können Daten bei 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 in den Daten der Applikation, haben einen direkten Einfluss auf das Angezeigte. Wenn die Applikation sich in einem inkorrekten Zustand befindet und es keine Laufzeitfehler

gab, können 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 inkorrekten Zustands eliminiert wird. Dazu wird jeder zusammenhängende Teil des Zustands als ein Deterministischer 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 (ST), wie bei einem Deterministischer endlicher Automat, zu erweitern. Auf diesem Wege wird eine Reduzierung von Bugs in größeren Applikationen angestrebt. Dabei wird insbesondere auf die Lesbarkeit und Wartbarkeit des Quellcodes sowie die Developer Experience geachtet.

Folgende Forschungsfragen werden behandelt:

1. Können Bugs, die aufgrund eines falschen Zustands entstehen, mithilfe von Strict Transitions reduziert werden?
2. Steigt oder sinkt die DX durch die Einführung von Strict Transitions?
3. Steigt oder sinkt die Lesbarkeit und Wartbarkeit des Codes durch die Einführung von Strict Transitions?

## 1.3 Aufbau

In dieser Arbeit werden die bestehenden SM-Ansätze um Übergänge wie bei einem DFA erweitert. Um dies zu erreichen, ist es notwendig, die Funktionsweise bestehender Ansätze zu kennen. Diese werden in Kapitel 4 aufgeführt. Anschließend werden die Deterministischer endlicher Automat-Übergänge, angepasst an den Anwendungsfall, einschließlich der JavaScript-API zur Definition in Kapitel 5 dargestellt. Die Erkenntnisse aus Kapitel 4 und 5 werden kombiniert, um zwei konkrete Implementierungen für Redux und Pinia zu zeigen. Danach wird der ST-Ansatz in Kapitel 6 mit dem normalen Ansatz verglichen. Bei dem Vergleich werden die quantitativen und qualitativen Aspekte analysiert, wobei die Bewertung der qualitativen Aspekte auf den üblichen Coding-Standards, Konventionen und Konzepten basiert. Abschließend werden die oben aufgeführten Forschungsfragen in Kapitel 7 basierend auf dem Vergleich beantwortet und ein Fazit gezogen.

# 2 Verwandte Arbeiten

Im Bereich State Management gibt es viele Arbeiten, die meistens in Form von konkreten Implementierungen sind. Trotzdem gibt es kaum Arbeiten, die sich mit Erhöhung der Korrektheit im SM beschäftigen.

Die bekannteste dieser Arbeiten ist die XState Bibliothek. Diese verwendet unter anderem Konzepte aus event-basierter Programmierung und endlichen Automaten, um die Korrektheit des Applikationszustands zu gewährleisten. Genauso wie bei einem Deterministischer endlicher Automat, gibt es eine Übergangsfunktion und unerlaubte Übergänge werden blockiert. Es handelt sich um eine framework-agnostische Lösung mit hauptsächlichem Fokus auf React. Um die Developer Experience zu erhöhen wird zusätzlich noch ein visueller Editor zur Verfügung gestellt.[18]

Weitere relevante Arbeiten ergaben sich bei der Recherche nicht.

# 3 Methodologie

In dieser Arbeit kamen quantitative Methoden zur Analyse von LOC, Bundle Size und Performance zum Einsatz. Zur Analyse der Developer Experience, Fehleranfälligkeit, Lesbarkeit und Wartbarkeit wurden hingegen qualitative Methoden verwendet. Die drei Aspekte Fehlerquote, DX, Lesbarkeit und Wartbarkeit, mit denen sich die Forschungsfragen beschäftigen, sind weitgehend mithilfe qualitativer Methoden beantwortbar. Da der quantitative Aspekt der LOC auch zu den genannten qualitativen Aspekten beiträgt, wird er in dieser Arbeit inkludiert. Die ebenfalls berücksichtigten quantitativen Merkmale der Performance und Bundle Size sind vor allem im Web von hoher Bedeutung und korrelieren mit wichtigen wirtschaftlichen Kennzahlen wie der Conversion-Rate.[1]

## 3.1 Aufbau

Damit der Vergleich realitätsnah ist, wurde eine Webapplikation mit üblichen Anforderungen wie Data-Fetching und Filterung gebaut. Die Applikation besteht aus einer Seite, welche eine Liste von 194 Produkten und Filteroptionen beinhaltet. Die Applikation fetcht und speichert die Produktdaten im Store. Die Filteroptionen sind: Titel, Preisobergrenze, Mindestbewertung, Verfügbarkeit und Kategorie.

Die Applikation wurde in React mit Redux und analog in Vue mit Pinia gebaut. Anschließend wurden die beiden Implementierungen kopiert und die Kopien um Strict-Transitions erweitert.

Das öffentliche GitHub-Repository mit allen Applikationen ist im Anhang A.3 verlinkt.

### 3.1.1 Quantitative Methoden

Die quantitativen Kennzahlen wurden mithilfe des Unix-Utilitys *wc* (word count) für LOC, Playwright und Chrome-Profiling für Performance sowie dem Vite-Build-Tool für

Bundle Size gesammelt. Das Ergebnis der Analysen sind die relativen und absoluten Veränderungen der Kennzahlen.

### 3.1.2 Qualitative Methoden

Die qualitative Analyse basiert auf weitverbreiteten Code-Konventionen, Patterns und Empfehlungen, die zur Lesbarkeit und Wartbarkeit des Quellcodes beitragen und die Produktivität des Entwicklers oder der Entwicklerin sowie die Fehlerquote der Applikation unmittelbar beeinflussen. Es ist wichtig zu erwähnen, dass die Schlussfolgerungen hierbei nicht vollständig von Subjektivität befreit sind.

## 3.2 Code-Ausschnitte

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

# 4 State-Management Ansätze

Bei den populären SM Lösungen folgen Redux und NgRx dem Flux-Pattern[16][6], 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.

## 4.1 Redux

Redux definiert sich durch folgende vier Eigenschaften:

1. Unveränderlichkeit (Immutability): Änderungen 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. Nachvollziehbarkeit (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.

### 4.1.1 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.[17]

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.

### 4.1.2 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-Function handelt, dürfen keine Seiteneffekte stattfinden. Wie eingangs erwähnt, ist der Zustand unveränderlich, daher dürfen hier keine direkten Veränderungen des Zustandes stattfinden. Es wird lediglich ein neues Objekt zurückgegeben. Falls keine Veränderungen stattfinden sollen, kann das ursprüngliche Objekt aus dem ersten Parameter unverändert zurückgegeben werden.[17]

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 das ursprüngliche Zustandsobjekt zu klonen.[5]

### 4.1.3 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 unter anderem *dispatch* und *getState*-Methoden. Mit diesen kann jeweils eine Action versendet und aus dem Store gelesen werden.

```
import { createStore } from 'redux'

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

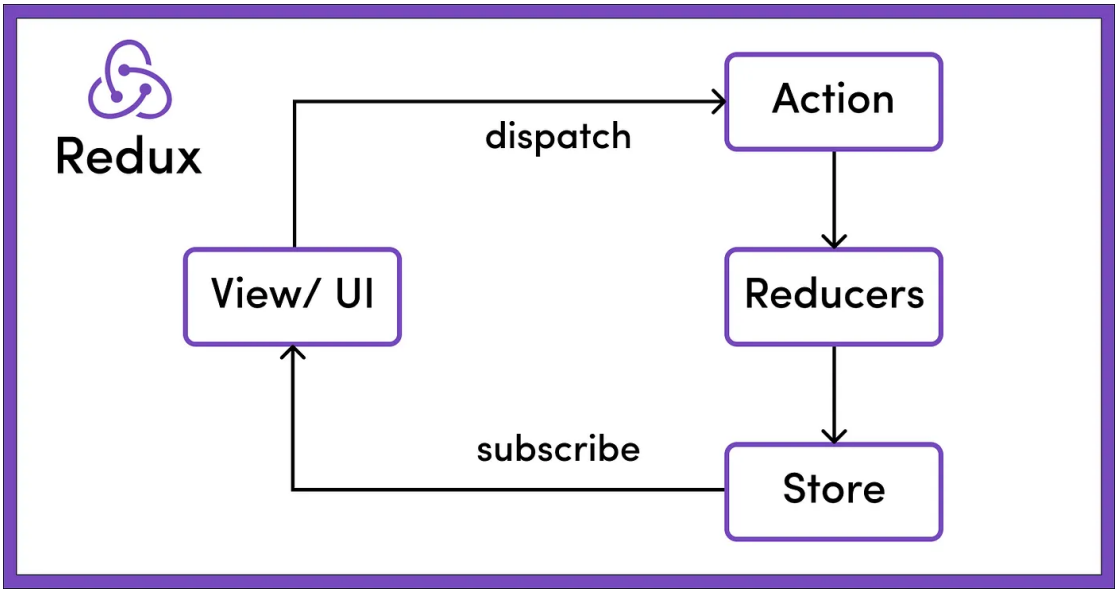


Abbildung 4.1: How to use Redux architecture — Part 1 [2024]

## 4.2 Pinia

Pinia ist sehr eng gekoppelt mit dem Vue-Framework und nutzt dessen Mechanismen der Reaktivität zur 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 dem Single-Store-Ansatz, bei dem alle Daten in einem zentralen Objekt leben, sondern es sind für Teile der Daten eigenständige Store-Instanzen zuständig. Pinia bietet zwei verschiedene APIs zur Definition von Stores an. In dieser Arbeit wird die *Options API* verwendet. Die Konzepte lassen sich auch auf die *Composition API* übertragen.[15] Die zwei essentiellen Konzepte sind *State* und *Action*.

### 4.2.1 State

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

### 4.2.2 Action

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

### 4.2.3 Definition eines Stores

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

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

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

### 4.2.4 Interaktion mit dem Store

Der Store kann in einer beliebigen 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 zu Membern des Store-Objekts.

```
  const userStore = useUserStore()

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

Der State im oberen Beispiel ist reaktiv und kann als *userStore.user* im Template der Komponente referenziert werden. Die Destrukturierung (destructuring) des Store-Objekts, im oberen Beispiel *userStore*, führt zum Verlust der Reaktivität. Aus diesem Grund wird die Punktnotation empfohlen.[15]

# 5 Strict Transitions

## 5.1 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 5.1 abgebildet, wird TypeScript von immer mehr Entwicklern genutzt, während die JavaScript-Nutzung abnimmt. 5.1 beinhaltet die tatsächliche Nutzung von TS nicht. Der TypeScript Compier 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.[7]

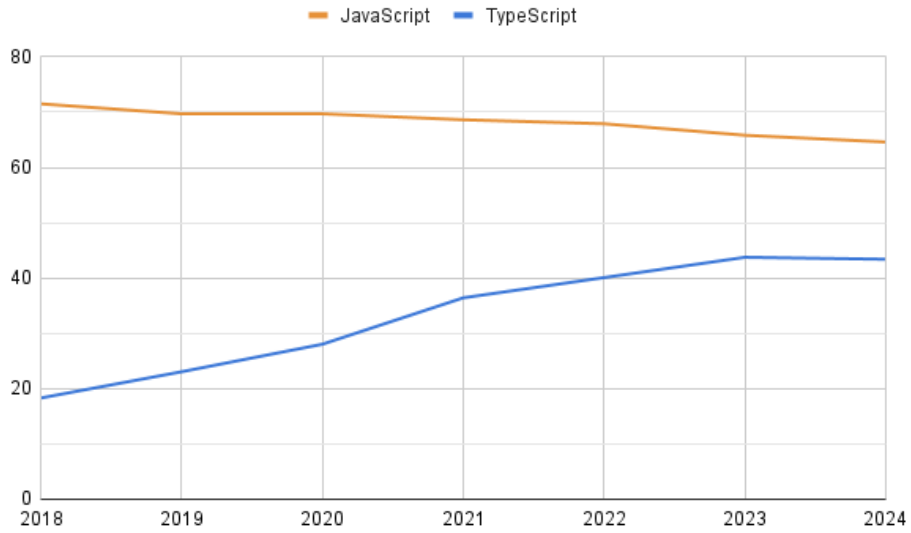


Abbildung 5.1: Stack Overflow Surveys der Jahre [2018], [2019], [2020], [2021], [2022], [2023] und [2024], Prozentuale Nutzung von JavaScript und TypeScript unter professionellen Entwicklern von 2018 bis 2024

## 5.2 Fehlende Garantie für Korrektheit des States

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* Action zu versenden, ohne vorher die *Fetch* Action versendet zu haben. Das heißt: „*items* wurden erfolgreich abgerufen“, ohne die Anfrage zuvor gemacht zu haben. Seitens Redux ist das Versenden einer Action immer, unbeachtet des aktuellen Zustandes, erlaubt. Dieser Faktor spricht gegen die Nachvollziehbarkeit und gilt für alle populären SM-Lösungen.

## 5.3 Korrektere Zustandsübergänge

Es wird vorgeschlagen, den Applikationszustand wie einen Zustand eines Deterministischer endlicher Automats zu behandeln. Im Falle von Redux werden die Actions als Übergänge und der Reducer als die Übergangsfunktion eines Deterministischer endlicher Automats gesehen. Restliche Eigenschaften des Quintupels eines Deterministischer endlicher Automats werden hierbei ignoriert.

Um die Übergangsfunktion zu definieren, wird pro Zustand eine *Übergangsliste* aller Actions benötigt, die bei diesem Zustand erlaubt sind. Ein Problem hierbei ist allerdings, dass die Identifizierbarkeit der einzelnen Zustände nicht garantiert ist. Abweichend von Deterministischer endlicher Automats 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* empfohlen, um zwischen verschiedenen Zuständen zu unterscheiden. Sie akzeptiert als Parameter den aktuellen Zustand und gibt ein Boolean zurück.

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

Mit dieser Funktion kann der Anwender für die Identifizierbarkeit der Zustände sorgen. Bei JavaScript-Klassen kann der *instanceof*-Operator genutzt werden, um auf die Instanz einer Klasse wie *Error* zu prüfen.[4] Des Weiteren können bei Objekten auf eindeutige Eigenschaften, wie die Präsenz eines Feldes per *in*-Operator geprüft werden.[3] Bei Arrays kann die *Array.isArray*-Funktion verwendet werden.[2] Durch die Kombination dieser und weiterer 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 Transition Map gilt: Identitätsfunktion ist der Schlüssel, während die Liste von Actions der Wert ist.

In der Übergangsfunktion wird der Zustandswechsel mit einer *Validierungsfunktion* validiert. Diese prüft mit Hilfe der Transition Map 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 Action 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 entwickeln kann.

Die Validerungsfunktion V ist wie folgt definiert:

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

### 5.3.1 Vorteile

1. **Übersichtlichkeit**: Damit die Validierung funktioniert, ist der Entwickler gezwungen, die Transition Map 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 die Weiterausführung falscher Geschäftslogik.

### 5.3.2 Nachteile

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

## 5.4 Implementierung

### 5.4.1 Redux

Im Folgenden wird die Implementierung der oben genannten Funktionen und Konzepte für Redux gezeigt.

Die Transition Map ist die Grundlage des Ansatzes. Mit Blick auf die Developer Experience und die Lesbarkeit wird die Transition Map 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 wird die Transition Map 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 Validierungsfunktion 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)
}
```

Es wird ein Proxy für *createStore* eingeführt, der 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 *transitions* Parameter erstellt einen Store mit der *createStore*-API von Redux und fügt dem Store drei neue Felder hinzu: die Transition Map, die Validierungsfunktion und die dispatchTransition-Methode.

```
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
```

Der TransitionStore kann per *createTransitionStore* erstellt werden. Die Actions werden per *TransitionStore.dispatchTransition(action)* dispatched. Mit dieser Implementierung bleiben die APIs zum Lesen und Manipulieren des Stores identisch. Lediglich die Funktion zur Erstellung eines Stores nimmt einen zusätzlichen *transitions* Parameter.

### 5.4.2 Pinia

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

Die Transition Map hat die gleiche Struktur wie die bei Redux. Allerdings wird das Feld *actionTypes* zu *action* umbenannt und steht für den Namen der Action im *actions*-Objekt.

Das Plugin wird mit der *transitions*-Funktion instanziiert, diese nimmt eine Map mit der ID des Stores als Schlüssel und die Transition Map als Wert für den jeweiligen Store:

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

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

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

Die *transitions*-Funktion gibt eine anonyme Funktion zurück, die ein Objekt als Parameter erhält. In diesem befindet sich unter anderem das Store-Objekt. Über die *Store.$onAction*-Methode kann eine Callback-Funktion als Preprocessor für Actions registriert werden. Die Callback-Funktion erhält ein Objekt als Parameter. In diesem sind

unter anderem der Name der aktuellen Action und der aktuelle Zustand enthalten. In dem Preprocessor wird die Validierungsfunktion aufgerufen.

```
type PiniaUseCallbackArgs = Parameters<PiniaUseCallback>[0]

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

    if (transitions) {
      store.$onAction(({ name, store }) => {
        // name ist der Name der aktuellen Action
        validateTransition(store.$state, name, transitions)
      })
    }
  }
}
```

Die Validierungsfunktion 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)
}
```

Das Plugin wird im Bootstrap-Schritt der Vue-Applikation registriert:

```
const app = createApp(App)
const pinia = createPinia()

pinia.use(stateTransitions({
  [itemStoreId]: itemStoreTransitions
}))

app.use(pinia)
app.mount('#app')
```

Im Gegensatz zum TransitionStore für Redux ändern sich bei Pinia, dank des Plugin-Systems, die APIs zur Erstellung, zum Lesen und zur Manipulation der Stores nicht.

# 6 Vergleich der Ansätze

Der in vorangegangen Kapiteln vorgestellte Strict Transitions-Ansatz erweitert die interne Funktionsweise einer State Management-Lösung. Er erfordert die zusätzliche Definition einer TransitionMap und es werden leicht geänderte APIs dem Benutzer zur Verfügung gestellt. In diesem Kapitel werden die Standard Stores (unveränderte) zu den mit Strict Transitions verglichen. Es werden die quantifizierbaren Kennzahlen Lines of Code, Bundle Size und Performance untersucht. Außerdem werden die Aspekte Developer Experience, Fehleranfälligkeit, Wartbarkeit und Lesbarkeit analysiert.

## 6.1 Quantifizierbare Aspekte

### 6.1.1 Lines of Code

Bei React wurden die .js, .ts, .tsx und bei Vue die .ts und .vue Dateien untersucht. Alle anderen Dateitypen wurden ignoriert, da diese weder für die Geschäftslogik noch für die UI verantwortlich sind.

Der relative Anstieg in LOC beträgt bei React $\sim 6\%$ und bei Vue $\sim 12\%$. Darüber hinaus beträgt der absolute Anstieg jeweils 28 Lines und 32 Lines. Siehe (Tab. 6.1)

### 6.1.2 Bundle Size

Analysiert wurden die Production Bundles der Applikationen. Diese wurden mit dem Build-Tool Vite erstellt. Alle Applikationen nutzen ausschließlich Client Side Rendering. Daher ist es wichtig, dass die Bundles so klein wie möglich bleiben.

Der relative Anstieg beträgt bei React $\sim 0,6\%$ und bei Vue $\sim 1\%$. Darüber hinaus beträgt der absolute Anstieg jeweils 0,98kB und 0,73kB. Der Anstieg in Bundle Size ist vernachlässigbar. Siehe (Tab. 6.2)

Tabelle 6.1: Statische Analyse bei Redux und Pinia mit und ohne ST

| Dateityp | Anzahl | LOC | Szenario |
|---|---|---|---|
| js | 1 | 25 | React ohne ST |
| ts | 14 | 256 | React ohne ST |
| tsx | 5 | 187 | React ohne ST |
| js | 1 | 25 | React mit ST |
| ts | 17 | 284 | React mit ST |
| tsx | 5 | 187 | React mit ST |
| ts | 5 | 140 | Vue ohne ST |
| vue | 5 | 133 | Vue ohne ST |
| ts | 5 | 172 | Vue mit ST |
| tsx | 5 | 133 | Vue mit ST |

Tabelle 6.2: Bundle Size Analyse bei Redux und Pinia mit und ohne ST

| Size in kB | Gziped | Szenario |
|---|---|---|
| 156,26 | 51,22 | React ohne ST |
| 157,24 | 51,54 | React mit ST |
| 70,43 | 28,23 | Vue ohne ST |
| 71,16 | 28,5 | Vue mit ST |

### 6.1.3 Performance

Um den Unterschied in Performance zu messen, wurde das Testing Tool Playwright verwendet. Mit Hilfe von Playwright wurde ein Szenario definiert. In diesem wurden alle Features der Webseite verwendet, welche Actions in Stores verursachen. Das Szenario wurde pro Applikation 20 Mal ausgeführt. Es lief im Chrome Browser und wurde mit Hilfe des Performance Tabs in den Chrome DevTools analysiert. Es wurden die Mittelwerte für Ausführungszeit in Millisekunden für die Browsertasks Scripting, Painting und Rendering ermittelt.

In allen Browsertasks, ausgenommen Scripting bei React, kann ein Rückgang in der Ausführungszeit beobachtet werden. Allerdings ist der Unterschied vernachlässigbar, da dieser sehr gering ist. Auch der Anstieg im Scripting bei React ist mit 0,89% ebenfalls vernachlässigbar. Siehe (Tab. 6.3)

Tabelle 6.3: Performance Analyse Redux und Pinia mit und ohne ST

| Task | ms ohne ST | mit ST | Delta | Library |
|---|---|---|---|---|
| Scripting | 1.111,45 | 1.121,35 | +0,89% | Redux |
| Painting | 858,05 | 841,05 | -1,98% | Redux |
| Rendering | 613,25 | 611,05 | -0,36% | Redux |
| Scripting | 1.680,15 | 1.677,95 | -0,13% | Pinia |
| Painting | 777,65 | 763,65 | -1,80% | Pinia |
| Rendering | 651,65 | 642,9 | -1,34% | Pinia |

Die Gerätespezifikation und Versionen der verwendeten Technologien sind aus der Tabelle A.1 im Anhang zu entnehmen.

## 6.2 Qualitative Aspekte

### 6.2.1 Developer Experience

Die DX wird durch die zusätzliche Aufgabe der Definition einer Transition Map beeinflusst. Sie führt zu mehr Code und somit zu zusätzlichem Aufwand.

### 6.2.2 Fehleranfälligkeit

Vorausgesetzt, die Transition Map bildet die zulässigen Übergänge vollständig und korrekt ab, kann sich die Applikation nicht in einem unzulässigen Zustand befinden. Obwohl sich hiermit die Fehlerstelle verlagert, ist diese zentral und nicht an vielen Orten verteilt. Falls sich die Applikation in einem falschen Zustand befindet, ist die Transition Map an einem zentralen Ort zu überprüfen, statt die ausgelösten Actions an vielen Orten.

Darüber hinaus bildet die Transition Map die Abläufe in der Applikation ab und kann für eine bessere Nachvollziehbarkeit sorgen. Außerdem ist die Transition Map ein POJO und kann somit ohne weiteres Mocking getestet werden.

### 6.2.3 Lesbarkeit

Die Lesbarkeit des gesamten Applikationscodes bleibt unverändert, ausgenommen ist die hinzukommende Transition Map. Die Lesbarkeit der Transition Map wird hauptsächlich durch die enthaltenen Identitätsfunktionen beeinflusst. Für diese wird der Einsatz von pure Functions mit geringen Abzweigungen und Funktionsaufrufen empfohlen. Wenn dies eingehalten wird, ist die zyklomatische Komplexität gering, was in der Regel eine bessere Lesbarkeit impliziert. Jedoch hängt die Lesbarkeit stark von den Konventionen und dem Code Style des Authors ab.

### 6.2.4 Wartbarkeit

Jede hinzukommende Action oder State muss in der Transition Map berücksichtigt werden. Daher steigt die Wartbarkeit.

# 7 Fazit

## 7.1 Beantwortung der Forschungsfragen

Die Hauptgegenstände des Strict Transitions-Ansatzes sind die Erhöhung der Produktivität des Entwicklers oder der Entwicklerin und das Erkennen von Bugs in frühen Phasen der Entwicklung sowie des Testings im Umgang mit dem Applikationszustand. Um diese Ziele zu erreichen, werden die Zustandsübergänge übersichtlicher an einem zentralen Ort definiert und Laufzeitfehler bei Verstößen geworfen. Die Definition der Zustandsübergänge ist inspiriert von der Übergangsfunktion eines DFAs. Das liegt an der intuitiven Natur der DFAs und der starken Übereinstimmung im Aufbau mit dem Zustand eines Web-Frontends.

Die drei zentralen Fragen, mit denen sich diese Arbeit beschäftigt, sind:

1. Können Bugs, die aufgrund eines falschen Zustands entstehen, mit Hilfe von Strict Transitions reduziert werden?
2. Steigt oder sinkt die DX durch die Einführung von Strict Transitions?
3. Steigt oder sinkt die Lesbarkeit und Wartbarkeit des Codes durch die Einführung von Strict Transitions?

### 7.1.1 Reduzierung von Fehlern

Damit Fehler im Zustand auf ein Minimum reduziert werden, ist die Definition der zulässigen Zustandsübergänge (Transition Map) erforderlich. Folglich können keine unerlaubten, also undefinierten, Zustandsübergänge stattfinden. Allerdings ist der ST-Ansatz ungeschützt vor fehlerhaften oder unvollständigen Definitionen. Eine Applikation mit einer Transition Map, die die gültigen Geschäftsprozesse und Nutzerinteraktionen nicht

widerspiegelt, ist anfällig für Bugs, die im Zusammenhang mit dem State stehen. Diesem Faktor kann die leichte Testbarkeit der Transition Maps entgegenwirken. Trotz der erhöhten Übersichtlichkeit aufgrund des Aufbaus der Transition Maps sind Fehler nach wie vor möglich, auch wenn potenziell in geringerem Umfang. Aus diesem Grund kann keine definitive Antwort auf diese Frage mit den Methoden dieser Arbeit geliefert werden. Eine zutreffendere Antwort sollte auf den Einsatz des ST in realen Applikationen und umfangreiches Testen basieren.

### 7.1.2 Developer Experience

Auf den ersten Blick kann von einer Verschlechterung der DX ausgegangen werden. Der Grund hierfür ist die zusätzliche Aufgabe der Definition der Transition Map. Dieser zusätzliche Aufwand kann jedoch potenziell zukünftige Bugs verhindern und somit den Gesamtaufwand für Bugfixes reduzieren. Jedoch kann auch bei diesem Punkt keine definitive Schlussfolgerung mit den Methoden dieser Arbeit gezogen werden.

### 7.1.3 Lesbarkeit und Wartbarkeit

Die Lesbarkeit und Wartbarkeit der gesamten Applikation bleiben unverändert. Die Lesbarkeit und Wartbarkeit der Transition Map sind hoch, da es sich hierbei um ein einfach testbares POJO handelt. Jeder dazukommende State muss in der Transition Map ergänzt werden, und genauso muss jeder entfernte State aus der Transition Map gelöscht werden.

## 7.2 Ausblick

Um die ersten beiden Forschungsfragen endgültig zu beantworten, sollten eine weitreichendere Analyse und Umfragen stattfinden.

Die DX kann durch Linting Rules, z. B. für ESLint, gesteigert werden. So könnte beispielsweise darauf überprüft werden, dass jede definierte Action in mindestens einer Transition Map referenziert wird. Außerdem könnte mit Hilfe eines Language Servers überprüft werden, dass jeder State mindestens einer Identitätsfunktion zugeordnet werden kann. Ein visueller Editor für die Transition Map könnte die Nachvollziehbarkeit zusätzlich erhöhen

und, wenn es um die Geschäftsprozesse geht, als Diskussionshilfe zwischen den Entwicklern und Product-Managern / Product-Ownern dienen.

Im Rahmen dieser Arbeit wurden drei Implementierungen für die Libraries Redux, NgRx und Pinia entwickelt. Diese Auswahl könnte um Libraries wie Zustand und Redux Toolkit erweitert werden.

# Literaturverzeichnis


[1] Daniel An, Google: Find out how you stack up to new industry benchmarks for mobile page speed. (2018). – URL https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

[2] 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

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

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

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

[6] NgRx, Brandon Roberts (.: *Getting Started*. 2024. – URL https://ngrx.io/guide/store. – official documentation

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

[8] Overflow, Stack: *Most popular technologies*. 2018. – URL https://survey.stackoverflow.co/2018#technology-_-programming-scripting-and-markup-languages. – Stack Overflow Survey

[9] Overflow, Stack: *Most popular technologies*. 2019. – URL https://survey.stackoverflow.co/2019#technology-_-programming-scripting-and-markup-languages. – Stack Overflow Survey

[10] Overflow, Stack: *Most popular technologies*. 2020. – URL https://survey.stackoverflow.co/2020#technology-programming-scripting-and-markup-languages. – Stack Overflow Survey

[11] Overflow, Stack: *Most popular technologies*. 2021. – URL https://survey.stackoverflow.co/2021#programming-scripting-and-markup-languages. – Stack Overflow Survey

[12] Overflow, Stack: *Most popular technologies*. 2022. – URL https://survey.stackoverflow.co/2022/#programming-scripting-and-markup-languages. – Stack Overflow Survey

[13] Overflow, Stack: *Most popular technologies*. 2023. – URL https://survey.stackoverflow.co/2023/#programming-scripting-and-markup-languages. – Stack Overflow Survey

[14] Overflow, Stack: *Most popular technologies*. 2024. – URL https://survey.stackoverflow.co/2024/technology. – Stack Overflow Survey

[15] Pinia, Eduardo San Martin Morote (.: *Defining a Store*. 2024. – URL https://pinia.vuejs.org/core-concepts/. – official documentation

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

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

[18] XState, David Khourshid (.: *XState*. 2024. – URL https://stately.ai/docs/xstate. – official documentation

# A Anhang

## A.1 Gerätspezifikation und Versionen der verwendeten Technologien

Tabelle A.1: Gerätspezifikation und Versionen der verwendeten Technologien

| Gerät | Apple Macbook Pro 2023 mit M3 CPU und 24GB Speicher |
|---|---|
| Betriebssystem | MacOS 15.3 |
| Docker | 27.4.1 |
| OrbStack | 1.9.5 |
| Nginx | 1.27.3 |
| Chrome Canary | 134.0.6994.0 |
| Playwright | 1.50.0 |
| Node | 20.17.0 |
| Vite | 6.0.5 |
| React | 18.3.1 |
| Redux | 5.0.1 |
| React Redux | 9.2.0 |
| Vue | 3.5.13 |
| Pinia | 2.3.1 |

## A.2 Beispiele für Implementierung des ST-Ansatzes

Im Rahmen dieser Arbeit resultierten zwei Implementierung des ST-Ansatzes. Eine für Redux und andere für Pinia. Diese sind im öffentlichen Repository auf Github unter https://github.com/s0h311/strict-transitions zu finden.

## A.3 Im Vergleich genutzte Projekte

Die, im Vergleich genutzten React und Vue Projekte sind im öffentlichen Repository auf Github unter https://github.com/s0h311/strict-transitions-benchmark zu finden.

## A.4 Verwendete Hilfsmittel

In der Tabelle A.2 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.2: Verwendete Hilfsmittel und Werkzeuge

| Tool | Verwendung |
|---|---|
| LaTeX | Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments |
| OpenAI ChatGPT 4 | Rechtschreib- und Grammatikprüfung |

## 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 |
|---|---|---|