

Web-Engineering II

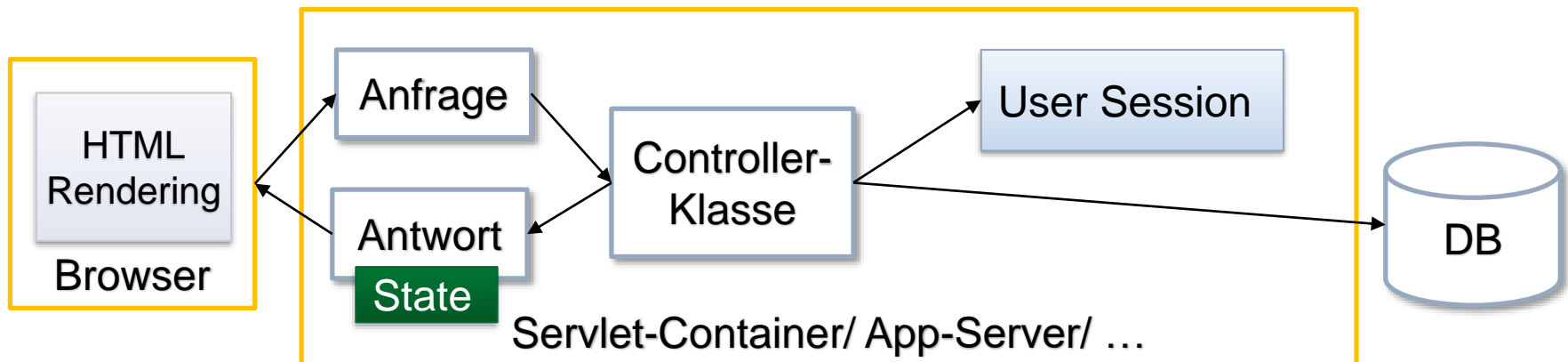
Prof. Dr. Sebastian von Klinski

React – State-Management und REST-Anbindung

- Wenn mit React Webanwendung umgesetzt wird, übernimmt...
 - React den dynamischen Aufbau der Seite
 - Das Layout wird mit CSS umgesetzt
- Zwei Aspekte fehlen aber noch...
 - Anwendungsweites State-Management
 - Stringente Anbindung von Backend
- Zentrale Fragen zum State-Management
 - Wie werden anwendungsweite Änderungen im State sinnvoll verwaltet und zwischen den Komponenten ausgetauscht?
 - Wie werden stets nur jene Komponenten bei Änderungen im State aktualisiert, für die die Änderung auch relevant sind?
- Zentrale Fragen zum Backend
 - Wie können die oft asynchronen Anfragen an das Backend sinnvoll in das State-Management eingebunden werden?

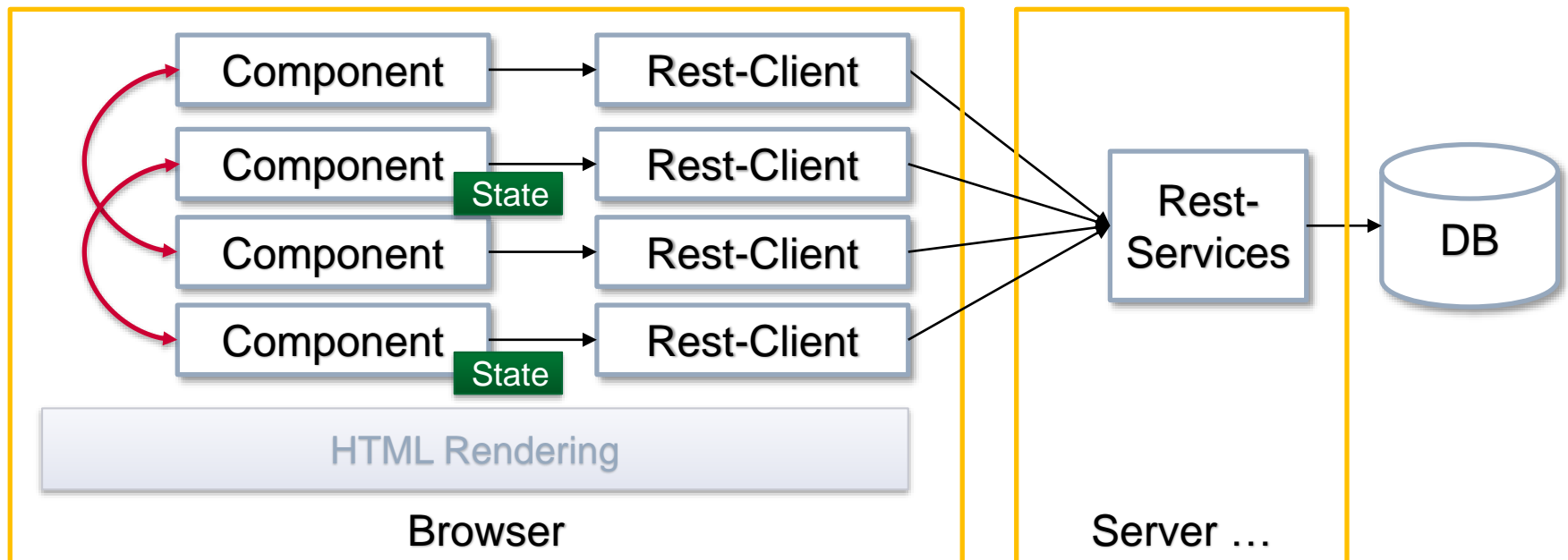
State-Management: Serverseitiges Rendering

- Bei serverseitigen Webanwendungen ist der State auf dem Server im Form einer User-Session
- Jede Anfrage vom Client wird auf dem Server bearbeitet, dort kann auf den gesamten State zugegriffen werden
- Bei Bedarf können Daten aus der Datenbank gelesen werden
- Die Antwort-Webseite kann mit allen notwendigen Daten des State ausgeliefert werden
- Eine Kommunikation zwischen den Komponenten auf Client-Seite ist nicht/ kaum notwendig, einen Client-State gibt es (fast) nicht



State-Management: Client-seitiges Rendering

- Mit SPAs und JavaScript-Web-Frameworks wie React, Vue, Angular ist ein wichtiger Teil der Kontrollstruktur zum Browser verschoben
- Insbesondere gibt es auf dem Server keinen State, nur auf Client-Seite
- Der State ist häufig über viele Komponenten verteilt
- Oft hat/benötigt jede Komponenten nur einen kleinen Teil des States



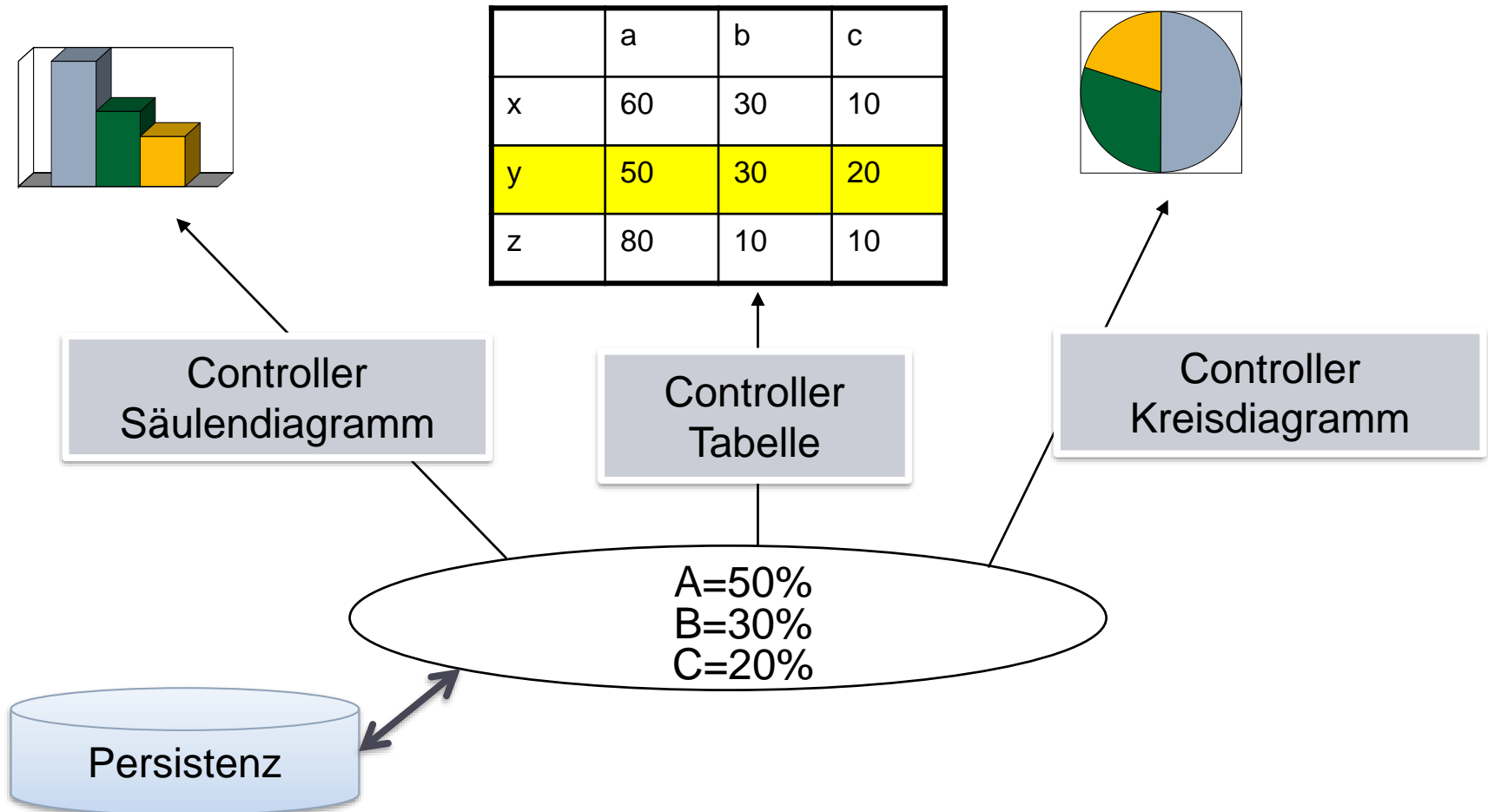
State-Management: Client-seitiges Rendering

- Der Austausch des State zwischen einzelnen Komponenten ist bei SPA häufig nicht leicht
- Das gilt für React, Angular, etc. gleichermaßen
 - Übliche Implementierungsansätze wie Singletons, Observer-Pattern, etc. sind nur mit Hilfsmitteln möglich → die Kommunikation zwischen Komponenten ist erschwert
 - Bei Software-Bibliotheken wie React wird Aktualisierung von Komponenten nur durch die Änderung des lokalen States ausgelöst → Weiterreichen des States unverzichtbar
 - Wenn der gleiche State an unterschiedlichen Orten verwendet wird oder innerhalb des States bidirektionale Abhängigkeiten bestehen, können schnell Endlosschleifen bei der Oberflächenaktualisierung entstehen

Separation of Concerns

- **Frage:** Wie müsste das State-Management umgesetzt werden?
- Für Antwort kann Konzept „Separation of Concerns“ herangezogen werden
- In jeder Anwendung gibt es in der einen oder anderen Form die folgenden Aspekte
 - **Rendering:** grafische Umsetzung der Benutzerschnittstelle
 - **Anwendungslogik/ State:** der logische Kern der Anwendung, häufig als Fachdomäne bezeichnet, umfasst auch den State
 - **Anwendungs-Controller:** Die Verbindung zwischen Darstellung und Anwendungslogik
 - **Persistenzschicht:** die Ebene, in der die Daten der Anwendungslogik gespeichert und wieder abgerufen werden kann
- Rendering, Anwendungslogik und Anwendungs-Controller entsprechen dem klassischen MVC-Pattern (Model-View-Controller)
- Je nach Software-Architektur werden diese Aspekte durch spezifische Software-Komponenten umgesetzt

Veranschaulichung von MVC

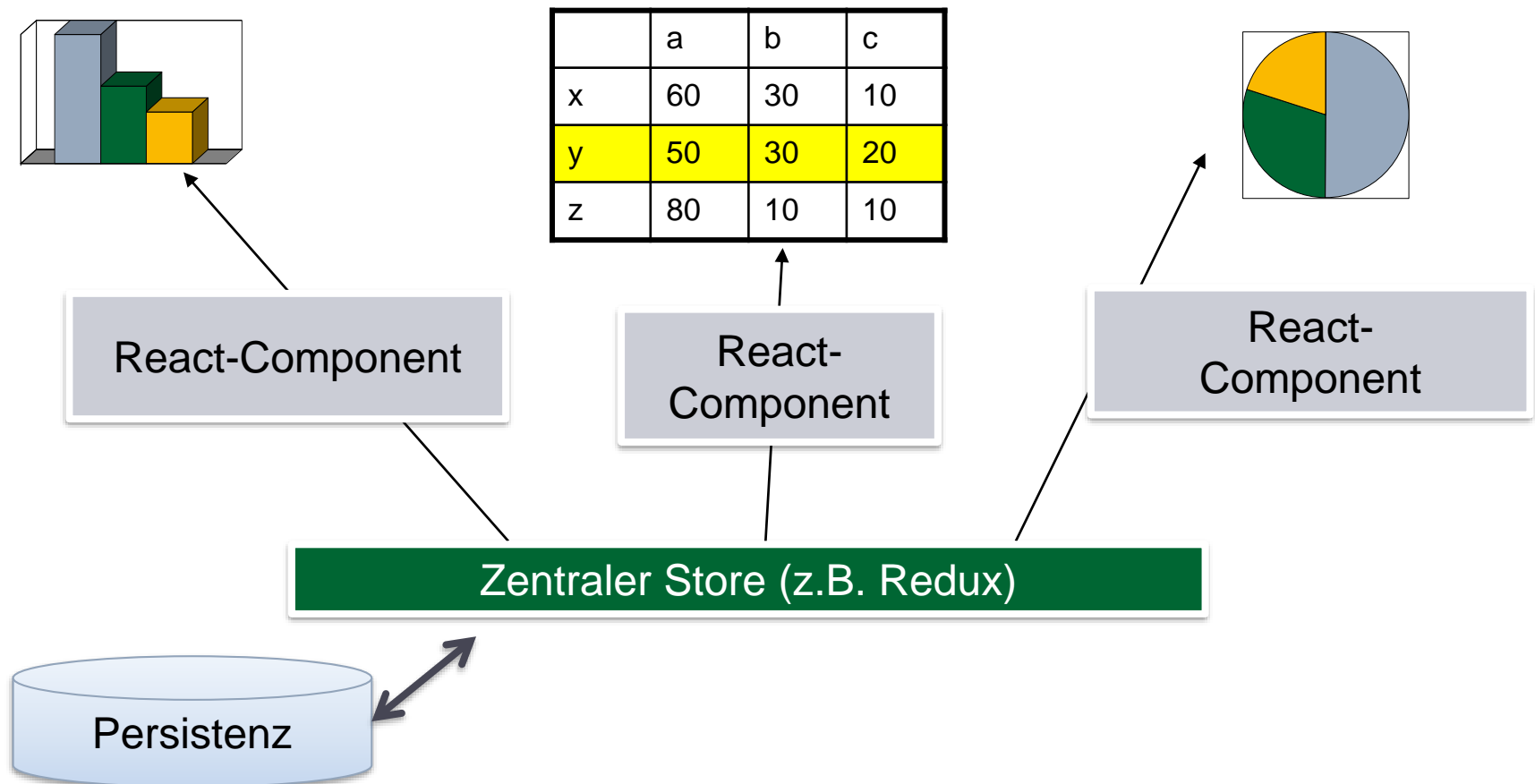


State-Management in SPAs

- Die Fragestellung zum State-Management ist in allen SPAs gegeben
- Häufige Antwort: zentrales State-Management
 - Es gibt einen zentralen Store, der den State beinhaltet
 - Komponenten holen sich den State vom Store
 - Komponenten melden Änderungen des State an den Store
 - Der Store informiert alle „interessierten“ Komponenten, wenn sich der State geändert hat
- Redux setzt genau dieses Konzept um
- Vergleichbare Umsetzung in Angular: NgRx

MVC in SPA mit Redux

- In SPAs übernimmt der zentrale Store die Funktion des Models



Separation of Concerns bei React

- Aufgabenteilung bei React
 - **Rendering:** Umgesetzt mit CSS und entsprechenden Bibliotheken
 - **Anwendungs-Controller:** Kernaufgabe von React
 - **Persistenzschicht:** Umgesetzt durch REST-Backend
 - **Anwendungslogik/ State:** Redux und...
- Redux setzt zentralen Store bei React um
 - Komponenten verbinden sich mit dem Store
 - Wenn der State geändert werden soll, schicken Komponenten entsprechende Befehle an den Store
 - Die Änderungen am State werden vom Store übernommen
 - Der Store informiert alle Komponenten über die Änderungen am State
 - Daraufhin können die Komponenten aktualisiert werden

React-Redux

Redux



- Redux ist ein State-Container für JavaScript-Anwendungen
- Quelloffene JavaScript-Bibliothek
- Zur Verwaltung von Anwendungszustand in einer Webanwendung
- Häufig Nutzung in Kombination mit React, Angular, etc.
- Ziel: alle Zustandsinformationen zentral an einer Stelle vorzuhalten und für alle Komponenten der Webanwendung zugänglich zu machen.
- Redux schließt zwei zentrale Design-Lücken von React
 - Die Kommunikation zwischen den Komponenten
 - Eine konsistente Verwaltung des Anwendungs-States

Quelle: [https://de.wikipedia.org/wiki/Redux_\(JavaScript-Bibliothek\)](https://de.wikipedia.org/wiki/Redux_(JavaScript-Bibliothek))

Redux

- Der zentrale Store wird mit `createStore()`-Methode von Redux angelegt
- Der Store wird dann allen Komponenten der React-Anwendung zur Verfügung gestellt, indem React-Komponenten in Redux-Komponente eingebettet werden
- Alle Komponenten können über Actions den State ändern
- Die eigentlichen Änderungen werden im Store von „Reducers“ ausgeführt
- Anschließend informiert der Store alle „interessierten“ Komponenten, dass sich der State geändert hat → die Oberfläche wird damit aktualisiert

```
import { createStore } from "redux";  
import rootReducer from "../reducers/index";  
const store = createStore(rootReducer);  
  
export default store;
```

Redux: Der Store

- Der Store verwaltet den State als Objekt, in dem beliebige Objekte abgelegt werden können
- Er koordiniert alle Änderungen am State über sogenannte Reducer
- Komponenten, die über Änderungen informiert werden wollen, können sich bei ihm registrieren
- Wenn Änderungen vorgenommen werden, informiert der Store alle Komponenten, die sich vorher bei ihm registriert haben
- Der Store hat die folgenden Funktionen, die von den Komponenten aufgerufen werden können
 - `getState()`: Zum Lesen des aktuellen States
 - `dispatch()`: Zum Versenden von Actions an den Store, damit Reducer die Action bearbeiten
 - `subscribe()`: Um über Änderungen im State informiert zu werden.
- Komponenten, die Änderungen am State vornehmen wollen, bekommen eine Referenz auf die `dispatch()`-Methode

Redux: Reducer

- Reducer sind Funktionen für das Anlegen und Ändern des States
- In Redux dürfen ausschließlich Reducer den State ändern
- Ein Reducer ist eine einfache JavaScript-Funktion, die 2 Parameter übergeben bekommt
 - Den aktuellen State
 - Die Aktion, die ausgeführt werden soll
- Der geänderte State wird zurückgegeben, wenn keine Änderungen vorgenommen werden sollen, wird der aktuelle State zurückgegeben

```
export default (state = 0, action) => {  
  switch (action.type) {  
    case 'INCREMENT':  
      return state + 1  
    case 'DECREMENT':  
      return state - 1  
    default:  
      return state  
  }  
}
```

Beispiel für
einen
Reducer, der
einen Counter
hoch- oder
runterzählt

Redux – Das ist anders...

- Ohne Redux ändern Komponenten ihren State selber (beispielsweise über `setState()`)
- Für einen konsistenten Einsatz von Redux darf das nicht mehr passieren!
- Wenn Redux verwendet wird, sollte der State nur noch zentral verwaltet werden
- Ansonsten besteht die Gefahr, dass der State inkonsistent wird
- Das bedeutet wiederum
 - Klassenkomponenten werden durch Redux weitgehend überflüssig
 - Der State ist der wesentliche Unterschied zwischen Klassen- und Funktionskomponenten bei React
 - Wenn der State zentral verwaltet wird, können auch (fast) alle Komponenten als Funktionen umgesetzt werden

Quelle: [https://de.wikipedia.org/wiki/Redux_\(JavaScript-Bibliothek\)](https://de.wikipedia.org/wiki/Redux_(JavaScript-Bibliothek))

Redux: Action

- Reducer ändern den Status auf Basis von Actions
- Actions sind wie Nachrichten, die den Reducern mitteilen, was zu ändern ist
- Actions sind einfache JavaScript-Objekte
- Das Attribut „type“ muss immer enthalten sein
- Das Attribut payload ist optional und kann den Inhalt für die Aktion umfassen
- **Hinweis:** Diese Namensgebung weicht von klassischen Actions in der Programmierung ab. Übliche Actions haben eine Methode execute() und führen die Logik selber aus

```
{  
  type: 'ADD_MESSAGE',  
  payload: { title: 'React Redux Tutorial', id: 1,  
             author: 'Sebastian',  
             messageText: 'Das ist Redux',  
          }  
}
```

Beispiel für
eine Action
zum Anlegen
einer
Nachricht

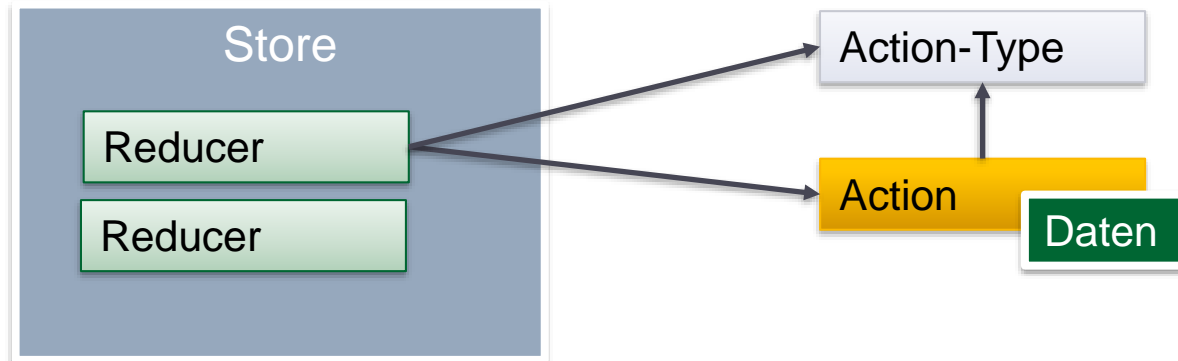
Redux: Action

- Actions beschreiben nur, was am State geändert werden soll
- Die eigentliche Änderung übernimmt der jeweils zuständige Reducer
- Über den Action-Type stellen Reducer fest, ob sie zuständig sind für das Bearbeiten der aktuellen Action
- Actions werden üblicherweise über entsprechende Methoden angelegt
- Vergleichbar zum Factory-Method-Pattern

```
import Constants from '.././Constants';  
  
export function addMessage(payload) {  
  return { type: Constants.ADD_MESSAGE, payload }  
};
```

Zusammenspiel

- Der Store wird mit den Reducern angelegt
- Die Reducer entscheiden anhand des Action-Types, ob sie die Action bearbeiten und welche Änderungen sie vornehmen
- Eine Action wird mit einem Action-Type angelegt
- Eine Action hat bei Bedarf noch Daten (payload), die zum Bearbeiten der Aktion notwendig sind



Ablauf beim Ändern eines States

1. Von einer Komponente der Web-Seite wird ein Event ausgelöst (z.B. ein Button wird gedrückt)
2. Der Event ruft die Funktion zum Instanziiieren der Aktion auf, übergibt die relevanten Daten und sendet die Aktion an den Store
3. Der Store gibt die Aktion an die Reducer weiter
 1. Die Reducer, die sich aufgrund des Action-Types für die Aktion zuständig fühlen, führen die entsprechenden Änderungen am State aus
 2. Anschließend reichen Sie die Aktion mit next() weiter
4. Wenn alle Reducer abgearbeitet sind, leitet der Store den geänderten State an die Komponenten weiter, die sich registriert hatten
5. Die entsprechenden Komponenten werden aktualisiert

Redux: Änderungen am State

- Reducer sollten die Objekte im State nicht verändern, sondern immer ersetzen!
- Ansonsten bekommt der Store nicht mit, dass sich der State geändert hat
 - Der Store vergleicht die Referenzen im Store
 - Wenn die Referenz sich nicht geändert hat, sendet er kein Update an die entsprechenden Komponenten → die Oberfläche wird dann nicht aktualisiert
- **Beispiel:** Kombination von `Object.assign()` und `Array.concat()` für Liste von Nachrichten

```
// Falsch!!
```

```
state.articles.push(action.payload);
```

```
// Richtig
```

```
Object.assign({}, state, {messages: state.messages.concat(action.payload)
```

Hinweis: JavaScript-Array-Funktionen

- `Array.prototype.push()`
 - Verändert der Original-Array, Referenz bleibt die gleiche

```
var array = [1, 2, 3, 4, 5];  
array.push(6);
```

- `Object.assign()`
 - Kopiert die Werte aller aufzählbaren, eigenen Eigenschaften von einem oder mehreren Quellobjekten in ein neues Zielobjekt.
 - Es wird das neue Zielobjekt zurückgegeben.
 - Gleiche Attribute werden überschrieben

```
const target = { a: 1, b: 2 };  
const source = { b: 4, c: 5 };  
const returnedTarget = Object.assign(target, source);  
// Ergebnis: Object { a: 1, b: 4, c: 5 }
```

Hinweis: JavaScript-Array-Funktionen

- `Array.prototype.concat()`
 - Führt zwei oder mehr Arrays zu einem zusammen.
 - Methode ändert nicht die existierenden Arrays, sondern gibt stattdessen ein neues Array zurück

```
const array1 = ['a', 'b', 'c'];  
const array2 = ['d', 'e', 'f'];  
const array3 = array1.concat(array2);  
// Ergebnis: Array ["a", "b", "c", "d", "e", "f"]
```

Reducer: Hinzufügen von Messages

- Der Reducer erzeugt einen Array mit den aktuellen Nachrichten im State und der neu hinzugekommenen Message
 - In **action.payload** ist eine neue Message
 - **state.messages.concat(action.payload)** erzeugt einen neuen Array und fügt sowohl action.payload als auch state.messages ein
 - **Object.assign({}, state, {messages:...** erzeugt ein leeres Array von Objekten, fügt die Objekte vom state ein und fügt dann noch das Objekt „messages“ ein, das ebenfalls ein Array von Objekten ist

```
function rootReducer(state = initialState, action) {  
  if (action.type === Constants.ADD_MESSAGE) {  
    return Object.assign({}, state, {  
      articles: state.articles.concat(action.payload)} );  
  }  
  return state;  
}
```

Eine erste Anwendung

Schritte, um Redux in React-Applikation einzubinden

1. Beim Anlegen der React-Applikation wird der Redux-Store angelegt und die App wird im Redux-Provider eingebettet
 - dadurch können sich die Komponenten mit Redux verbinden
2. Beim Anlegen des Stores wird der (Root-)Reducer übergeben
 - Der Root-Reducer beinhaltet alle Reducer, die für das Ändern des States verantwortlich sind
3. Action-Factory-Methods werden umgesetzt
 - Sie instanziiieren die Aktions, die verschickt werden sollen
4. Die Action-Factory-Methods werden an die HTML-Komponenten angebunden
 - damit können HTML-Komponenten Actions auslösen
5. Komponenten werden mit dem Redux-Store verbunden
 - dadurch werden die Komponenten Actions verschicken, bzw. werden über Änderungen informiert

1. React-Redux: Anwendung mit Redux anlegen

- Installation: `npm i react-redux`

```
import React from 'react'
import ReactDOM from 'react-dom';
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './tutorials/Articles/ArticlesApp'
import rootReducer from './tutorials/Articles/reducers/index'
```

```
const store = createStore(rootReducer)
```

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

1. React-Redux: Anwendung mit Redux anlegen

- Von Redux wird die `createStore()`-Methode importiert, um den Store anzulegen
- Es wird die `Provider`-Komponente importiert, um die React-App darin einzubetten → Dadurch können sich eingebettete Komponenten mit Redux verbinden

```
...  
import { createStore } from 'redux'  
import { Provider } from 'react-redux'  
...  
const store = createStore(rootReducer)  
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>,  
  document.getElementById('root')  
)
```

1. React-Redux: Anwendung mit Redux anlegen

- Die „App“ kann zunächst eine normale React-Komponente sein
- Beim Anlegen vom Store wird der Root-Reducer übergeben

```
...  
import App from './tutorials/Articles/ArticlesApp'  
import rootReducer from './tutorials/Articles/reducers/index'  
  
const store = createStore(rootReducer)  
  
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>,  
  document.getElementById('root')  
>);
```

2. (Root-)Reducer anlegen

- Der Root-Reducer kann zunächst eine einfache Funktion sein
- Die Funktion bekommt den aktuellen State und die Aktion übergeben
- Mit der Zuweisung vom State im Funktionskopf (state = initialState) kann der State initialisiert werden
- Bei sehr kleinen Anwendungen kann es ausreichen, einen Reducer zu haben, der alle Änderungen vornimmt

```
const initialState = {  
  articles: []  
};  
  
function rootReducer(state = initialState, action) {  
  return state;  
};  
  
export default rootReducer;
```

2. (Root-)Reducer anlegen

- Üblicherweise setzt sich jedoch der Root-Reducer aus mehreren Reducern zusammen
- Zum Zusammenführen von Reducern wird `combineReducers()` verwendet

```
import { combineReducers } from 'redux';

import { authentication } from './authentication.reducer';
import { registration } from './registration.reducer';
import { users } from './users.reducer';

const rootReducer = combineReducers({
  authentication,
  registration,
  users
});
export default rootReducer;
```

2. (Root-)Reducer anlegen

- Beispiel: Reducer um Artikel (Artikelbezeichnung und ID) anzulegen

```
import { ADD_ARTICLE } from "../constants/actionTypes";
const initialState = {
  articles: []
};
function rootReducer(state = initialState, action) {
  if (action.type === ADD_ARTICLE) {
    return Object.assign({}, state, {
      articles: state.articles.concat(action.payload)
    });
  }
  return state;
};
export default rootReducer;
```

ADD_ARTICLE ist String, identifiziert Action Type

Reducer-Funktion bekommt den aktuelle State
und die auszuführende Action übergeben

2. (Root-)Reducer anlegen

- Der Reducer muss den aktualisierten State zurückgeben
- Ganz wichtig: die betreffenden Objekte im State dürfen nicht verändert, sondern müssen ersetzt werden!
 - Ansonsten wird stellt Redux keine Änderung im State fest und
 - Die Komponenten werden nicht aktualisiert

```
...  
function rootReducer(state = initialState, action) {  
  if (action.type === ADD_ARTICLE) {  
    return Object.assign({}, state, {  
      articles: state.articles.concat(action.payload)  
    });  
  }  
  return state;  
};  
...
```

3. Action-Factories anlegen

- Actions sind einfache JavaScript-Objekte, die alle Informationen beinhalten müssen, damit die Reducer die Anfrage bearbeiten können
- Beispiel:

```
{  
  type: 'ADD_ARTICLE',  
  payload: { title: 'React Redux Tutorial', id: 1 }  
}
```

- Für ein konsistentes Anlegen von Actions werden die Actions über entsprechende Factory-Methods angelegt:

```
export function addArticle(payload) {  
  return { type: "ADD_ARTICLE", payload }  
};
```

3. Action-Factories anlegen: Action-Types

- Action-Types sind einfache Strings
- Um Verwechslungen und Fehler zu vermeiden, sollten Action-Types als Konstanten angelegt werden
- Sie können in einer zentralen Datei „constants.js“ abgelegt werden

```
export const ADD_ARTICLE = "ADD_ARTICLE";
```

- Angepasste Factory-Method:

```
import { ADD_ARTICLE } from "../constants/actionTypes";

export function addArticle(payload) {
  return { type: ADD_ARTICLE, payload };
}
```

3. Action-Factories anlegen: Reducer erweitern

- Für jeden Action-Type sind die entsprechenden Aktionen im Reducer umzusetzen

```
import { ADD_ARTICLE } from "../constants/actionTypes";
...
function rootReducer(state = initialState, action) {
  if (action.type === ADD_ARTICLE) {
    return Object.assign({}, state, {
      articles: state.articles.concat(action.payload)
    });
  }
  return state;
};
export default rootReducer;
```

4. HTML-Komponenten mit Actions verbinden

- Zum Ändern des States im Store aus einem Formular heraus müssen die folgenden Schritte ausgeführt werden:
 1. Die Änderungen im Formular werden in den State der Komponente übernommen
 2. Bei Drücken des „Submit“-Button wird eine Funktion aufgerufen, die eine Aktion erzeugt und
 3. ... die Aktion an den Store schickt.
 4. Der Store gibt die Aktion an den Reducer
 5. Der Reducer ändert den State

4. HTML-Komponenten mit Actions verbinden

- Formular zum Anlegen von Artikeln

```
import React, { Component } from 'react'
...
class ConnectedForm extends Component {
  render() {
    const { title, id } = this.state;
    return (
      <form>
        <label htmlFor="name">Artikelbezeichnung</label>
        <input type="text,, name="title,, id="title,, value={title}
          onChange={this.handleChange} />
      ...
        <input type="button" value="Submit" onClick={this.handleSubmit} />
      </form>
    );
    ...
  }
}
```

Quelle: <https://www.valentinog.com/blog/redux/>

4. HTML-Komponenten mit Actions verbinden

- Änderungen vom Form in den State übernehmen

```
import React, { Component } from 'react'
...
class ConnectedForm extends Component {
  constructor(props) {
    super(props);
    ...
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange = event => {
    const { name, value } = event.target
    this.setState({
      [name]: value,
    })
  }
  render() {
    const { title, id } = this.state;
    return (
      ...
      <input type="text" name="title" id="title" value={title} onChange={this.handleChange} />
      ...
    )
  }
}
```

Methode muss an
Instanz gebunden
werden

Das geänderte Attribute wird
aktualisiert

Methode wird an Input gehängt

Quelle: <https://www.valentinog.com/blog/redux/>

4. HTML-Komponenten mit Actions verbinden

- Zum Weiterleiten der geänderten Daten mit dem Submit-Button muss ...
 1. eine Aktion mit den Daten aus dem Formular angelegt werden
 2. die Aktion per `dispatch()` an den Store geschickt werden
- Damit Aktionen an den Store geschickt werden können, wird zunächst eine Referenz auf die `dispatch()`-Methode benötigt
- Die Referenz auf die `dispatch()`-Methode kann über zwei Wege bestimmt werden
 - Eine mit Redux verbundenen Komponenten bekommt die Referenz auf die Methode mit den Properties
 - Es wird die Funktion `mapDispatchToProps()` verwendet
- Der gängige Ansatz ist die Verwendung der Methode `mapDispatchToProps()`

4. HTML-Komponenten mit Actions verbinden

- Es muss zunächst die Factory-Methode für die Action/en geladen werden (z.B. `addArticle`), die später per `dispatch()` verschickt werden sollen
- Der Methode `mapDispatchToProps()` wird die `dispatch()`-Methode übergeben, wenn sie von Redux später aufgerufen wird
- Der `dispatch()`-Methode kann dann die Action übergeben, die über die Factory-Methode `addArticle()` erzeugt wird

```
import React, { Component } from 'react'
...
import { addArticle } from '../actions/index'

function mapDispatchToProps(dispatch) {
  return {
    addArticle: article => dispatch(addArticle(article))
  };
}
...
```

Factory-Methode zum Anlegen von einem Artikel

`mapDispatchToProps` ruft Factory-Methode auf

Quelle: <https://www.valentinog.com/blog/redux/>

4. HTML-Komponenten mit Actions verbinden

- Damit `mapDispatchToProps()` aufgerufen wird, muss die Komponente mit Redux verbunden werden
- Bei der `connect()`-Funktion wird `mapDispatchToProps()` als 2ter Parameter übergeben
- `Connect()` gibt eine Funktion zum Einbetten der eigentlichen Komponenten zurück, dieser wird dann das `ConnectedForm` übergeben

```
import React, { Component } from 'react'  
import { connect } from "react-redux";  
...  
const Form = connect(null,mapDispatchToProps)(ConnectedForm);  
export default Form;
```

Die `connect()`-Funktion wird von Redux importiert

Die Funktion `mapDispatchToProps` wird beim Aufruf der `connection`-Funktion als 2ter Parameter übergeben

Quelle: <https://www.valentinog.com/blog/redux/>

4. HTML-Komponenten mit Actions verbinden

- Als letzten Schritt muss noch an das Submit das Erzeugen und Versenden der Aktion umgesetzt werden

```
...
class ConnectedForm extends Component {
...
  constructor(props) {
    super(props);
    ...
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleSubmit(event) {
    event.preventDefault();
    this.props.addArticle({ title: this.state.title, id: this.state.id });
    this.setState({ title: "" });
  }
...
  <input type="button" value="Submit" onClick={this.handleSubmit} />
</form>
...
```

Die handleSubmit-Funktion mit an die Komponente gebunden werden

In der handleSubmit()-Funktion wird die Factory-Method addArticle() mit den Daten aus dem Formular aufgerufen

An den Button wird der Aufruf von handleSubmit() angehängen

Quelle: <https://www.valentinog.com/blog/redux/>

5. Komponenten mit Store verbinden

- Komponenten, die den zentralen Store verwenden wollen, müssen mit dem Store verbunden werden
- Zum Anbinden einer Komponenten an den Store gibt es die `connect()`-Methode von Redux:
 - `connect()`-Methode importieren
 - Anstelle von der Komponente wird `connect()`-Methode aufgerufen
 - `connect()` gibt „Higher-Order“-Komponente zurück, in die dann die eigentliche Komponente eingebettet wird
 - Es wird die Higher-Order-Komponente zurückgegeben

```
...  
import { connect } from 'react-redux';  
...  
class LoginButton extends Component {  
  ...  
}  
export default connect()(LoginButton)
```

5. Komponenten mit Store verbinden

- Durch das Verbinden der Komponente mit dem Store erhalten die Komponenten in ihren Properties die `dispatch()`-Methode vom Store
- Mit der `dispatch()`-Methode können Komponenten Action an Store schicken
- Sie erhalten aber noch nicht den geänderten State, wenn sich der State im Store ändert

```
...
class LoginButton extends Component {
  ...
  showLoginDialog() {
    const dispatch = this.props.dispatch
    dispatch(getShowLoginDialogAction())
  }
  render() {
    return (
      <div>
        <Button variant="light" onClick={this.showLoginDialog}>Login</Button>
      </div>
    )
  }
}
```

5. Komponenten mit Store verbinden

- Komponenten müssen mit dem Store verbunden werden, wenn...
 - a) ... Komponenten über Änderungen im State informiert werden wollen
 - b) ... die Komponenten den State verändern wollen
- Wenn Komponenten über Änderungen im State informiert werden sollen, ...
 - ... muss die Komponenten bei Änderungen informiert werden, um sich neu zu rendern
 - ... für das Rendern werden der aktuelle State benötigt
- Wenn Komponenten den State verändern wollen, brauchen sie eine Referenz auf die `dispatch()`-Methode, damit ...
 - ... lokale Funktionen mit `dispatch()` Actions verschicken können.
 - ... um Funktionen, die nicht in der Komponente definiert sind, denen die `dispatch()`-Methode weiterzureichen

5. Komponenten mit Store verbinden

- Damit Komponenten über Änderungen im State informiert werden, muss beim `connect()` Redux mitgeteilt werden, welcher Teil des State in die Properties kopiert werden sollen
- Das Kopieren der Daten aus dem State wird über eine Funktion mit dem Namen `mapStateToProps()` gemacht
- `mapStateToProps()` kopiert sich aus dem State jene Daten, die für die Komponente von Relevanz sind
- Hinweis: Die Properties setzen sich damit aus den Properties der Parent-Komponenten und den von Redux kopierten Properties zusammen und der `dispatch()`-Methode
- Der Impuls zum Rendern von Komponenten kommt nun auch vom Store, nicht mehr nur von der Parent-Komponente

5. Komponenten mit Store verbinden: MapStateToProps

- **Beispiel:** Die Komponente soll die aktuellen Artikel im Store auflisten
- Die Liste soll automatisch aktualisiert werden, wenn ein neuer Artikel dazukommt

```
import React from "react";
import { connect } from "react-redux";
const mapStateToProps = state => {
  return { articles: state.articles };
};
const ConnectedList = ({ articles }) => (
  <ul>
    {articles.map(el => (
      <li key={el.id}>{el.title}</li>
    ))}
  </ul>
);

const List = connect(mapStateToProps)(ConnectedList);
export default List;
```

In den Array articles wird der articles-Array aus dem State kopiert

Zum Rendern holt sich die Komponente den kopierten Array aus den Properties

Beim Verbinden der Komponente wird die Funktion übergeben, die das Kopieren des States ausführt

5. Komponenten mit Store verbinden: MapStateToProps

- `ConnectedList` eine Funktionskomponente
- Sie bekommt die Artikel als Properties übergeben
- Sie gibt als HTML eine Liste zurück, in der die Artikel als List-Items enthalten sind.
- Jedes List-Item hat als “key”-Attribute die ID des Artikels
- React empfiehlt für Listen, immer ein “key”-Attribut zu setzen!
- `ConnectedList` entspricht einer normale React-Funktionskomponente

```
...  
const ConnectedList = ({ articles }) => (  
  <ul>  
    {articles.map(el => (  
      <li key={el.id}>{el.title}</li>  
    ))}  
  </ul>  
);  
...
```

5. Komponenten mit Store verbinden: MapStateToProps

- mapStateToProps ist eine Funktion, die als Parameter den State übergeben bekommt
- Sie holt sich aus dem State nur die Artikel raus und gibt ein Objekt mit den Artikeln als einziges Attribut zurück
- Dieses Objekt wird zu den Properties hinzugefügt, die die Komponente beim nächsten Rendern erhält

```
...  
import { connect } from "react-redux";  
  
const mapStateToProps = state => {  
  return { articles: state.articles };  
};  
  
...  
const List = connect(mapStateToProps)(ConnectedList);  
export default List;
```

5. Komponenten mit Store verbinden: MapStateToProps

- Vor dem Exportieren der Komponente wird die `connect()`-Methode von Redux aufgerufen, als Parameter wird die `mapStateToProps()`-Methode übergeben
- Mit dem `connect()` merkt sich der Store, dass diese Komponente an Änderungen interessiert ist
- Wenn sich der State ändert, übergibt der Store an `mapStateToProps()` den aktuellen State
- `mapStateToProps()` gibt den relevanten Teil des States zurück
- Dieses zurückgegebene Objekt kopiert dann der Store in die Properties der Komponente und ruft die `render()`-Methode der Komponenten auf

```
...
import { connect } from "react-redux";

const mapStateToProps = state => {
  return { articles: state.articles };
};

...
const List = connect(mapStateToProps)(ConnectedList);
export default List;
```

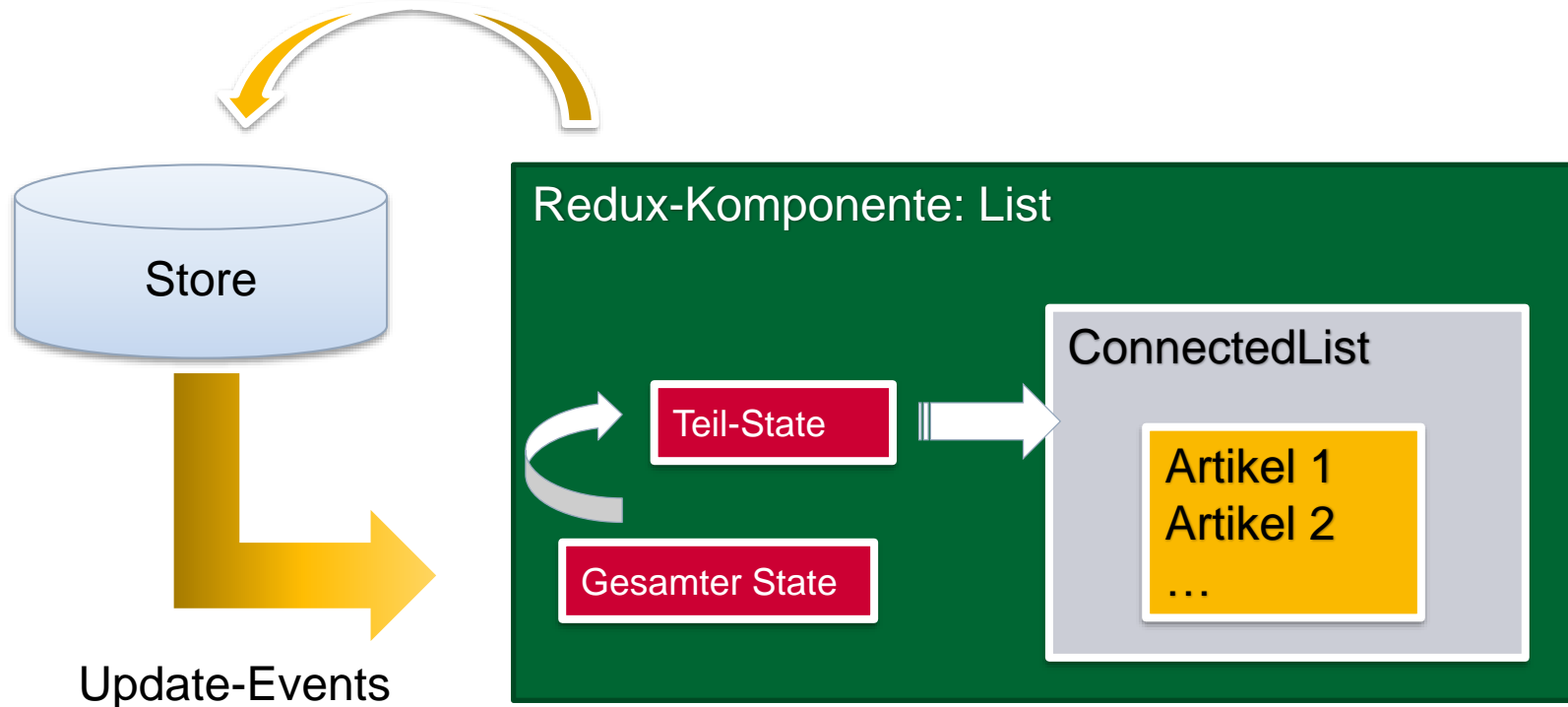
5. Komponenten mit Store verbinden: MapStateToProps

- Beim `connect()` wird die eigentliche Komponente in eine Higher-Order-Komponenten eingebettet
- Die Higher-Order-Komponente erweitert die Funktionen der Komponente durch Redux-spezifische Funktionen wie das Anreichern des State
- Exportiert wird nicht die ursprüngliche Komponente, sondern die Higher-Order-Redux-Komponente mit der eingebetteten React-Komponente

```
...  
import { connect } from "react-redux";  
  
const mapStateToProps = state => {  
  return { articles: state.articles };  
};  
...  
const List = connect(mapStateToProps)(ConnectedList);  
export default List;
```

5. Komponenten mit Store verbinden: MapStateToProps

Registriert sich als Observer



5. Komponenten mit Store verbinden: MapDispatchToProps

- Damit Komponenten Actions an den Store, bzw. die Reducer schicken kann, braucht die Komponenten eine Referenz auf die dispatch()-Methode
- Eine Komponente, die per connect() mit dem Store verbunden wurde, erhält standardmäßig eine Referenz in den Properties

```
showLoginDialog() {  
  const dispatch = this.props.dispatch  
  dispatch(getShowLoginDialogAction())  
}
```

- Dieser Ansatz ist sinnvoll für alle Funktionen, die in der Komponente selber definiert werden

5. Komponenten mit Store verbinden: MapDispatchToProps

- Wenn Funktionen aus anderen Modulen dispatch() verwenden sollen, müssen diese Funktionen per mapDispatchToProps() verknüpft werden
- Diese Funktionen werden mit connect() dem Store übergeben
- Der Store wrappt diese Funktion in Funktionen, die den Funktionen die Dispatch-Methode übergeben

```
...
handleClose() {
  const { hideLoginDialogAction } = this.props;
  hideLoginDialogAction();
}
...
const mapDispatchToProps = dispatch => bindActionCreators({
  showLoginDialogAction: authenticationService.showLoginDialog,
  hideLoginDialogAction: authenticationService.hideLoginDialog,
  authenticatUserAction: authenticationService.authenticationUser
}, dispatch)
const ConnectedUserSessionWidget = connect(mapStateToProps,
mapDispatchToProps)(UserSessionWidget);
```

5. Komponenten mit Store verbinden: MapDispatchToProps

- Wenn bei connect() eine mapDispatchToProps()-Funktion übergeben wird, wird die dispatch()-Methode nicht mehr im State mitgeliefert!
- bindActionCreatorsCreators() ermöglicht das einfache Einbinden von mehreren Funktionen

```
...  
import { bindActionCreators } from 'redux'  
...  
const increment = () => ({ type: 'INCREMENT' })  
...  
function mapDispatchToProps(dispatch) {  
  return bindActionCreators({ increment, decrement, reset }, dispatch)  
}  
  
connect(  
  null,  
  mapDispatchToProps  
) (Counter)
```

5. Komponenten mit Store verbinden: MapDispatchToProps

- Wenn den Funktionen abweichende Namen gegeben werden sollen, kann diese auch bei `bindActionCreators()` gemacht werden

```
...  
const mapDispatchToProps = dispatch => bindActionCreators({  
  showLoginDialogAction: authenticationService.showLoginDialog,  
  hideLoginDialogAction: authenticationService.hideLoginDialog,  
  authenticaUserAction: authenticationService.authenticationUser  
}, dispatch)  
...
```

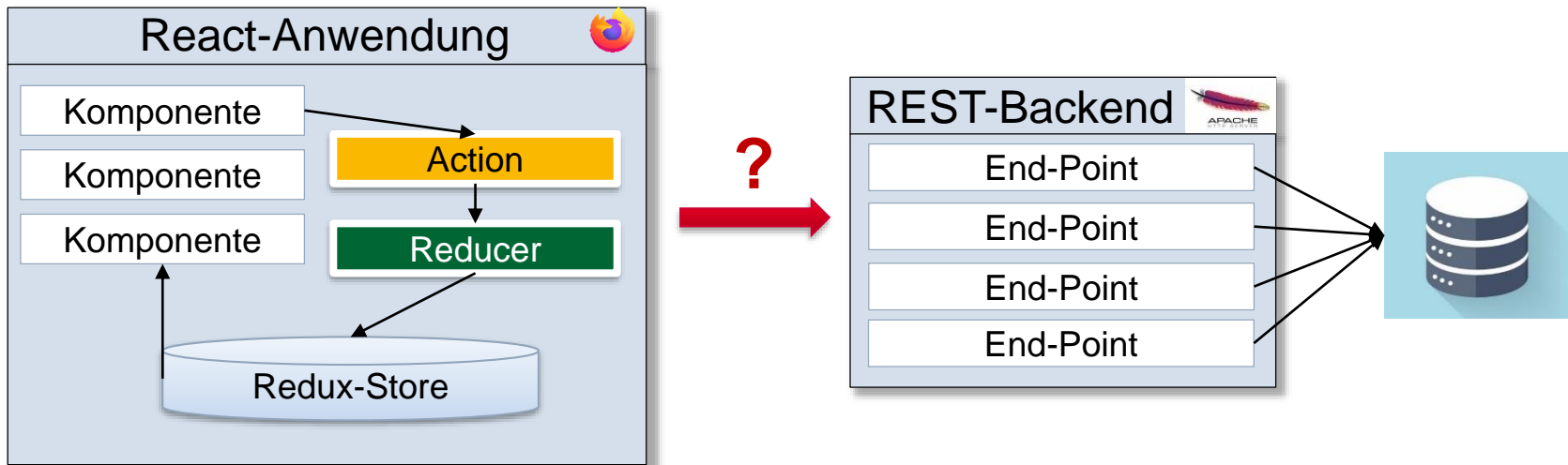
Redux- und React-Einbindung

- Das erste Aufsetzen der Redux- und React-Einbindung ist aufwändig und etwas komplex
- Jedoch die Erweiterung um neue Aktionen und Reducer geht in der Regel sehr schnell
 - Die Struktur von Actions und Reducer ist immer sehr ähnlich
 - Neue Funktionen können durch Kopieren und Anpassen umgesetzt werden

Anbindung eine REST-Backends

Anbindung von REST-Backend

- Redux setzt zentralen Store um, sieht aber keine Funktionen für das Anbinden von REST-Backend vor
- Anbinden von Backend beinhaltet asynchronen Aufruf ans REST-Backend und anschließend asynchrone Aktualisierung der Web-Oberfläche
- **Frage:** Wann und wo sollte Aufruf ausgeführt werden?



Anbindung von REST-Backend

- Die Frage ist: Wo sollten die asynchronen Aufrufe gestartet werden?
- Die naheliegende Antwort: in den Actions
 - Die asynchronen Aufrufe an das Back-end werden immer durch eine Action ausgelöst
 - Für jede Action muss auch überwacht werden, ob die Action erfolgreich ausgeführt wurde
 - Je nach Rücklauf vom Backend müssen unterschiedliche angepasste Folgeaktivitäten ausgeführt werden
- **Beispiel:** Schritte einer Action zum Abrufen einer Liste von Produkten
 1. Anfrage an den Server wird abgeschickt, der Status ist „**pending**“
 2. Die Antwort vom Server kommt zurück. Mögliche Ergebnisse:
 1. Die Anfrage war erfolgreich → die Produkte werden dargestellt
 2. Die Anfrage ist fehlgeschlagen → eine entsprechende Nachricht wird dargestellt
- Dieser Prozess kann am besten innerhalb der Action abgebildet werden

Redux-Thunk

- Thunk ist eine alternative Bezeichnung für Funktion
- In Redux sind Actions nur Objekte, mit einem Type und dem Payload
- Die Software-Komponente Redux-Thunk ist eine Erweiterung für Redux, um anstelle von Objekten auch Funktionen an den Store zu senden
- Die einzige Aufgabe von Redux-Thunk ist: wenn in der Action eine Funktion ist, führt Thunk die Funktion aus

Redux-Thunk

- Redux ist eine sehr einfache Middleware-Funktion (siehe Source-Code in node_modules)

```
function createThunkMiddleware(extraArgument) {  
  return ({ dispatch, getState }) => next => action => {  
    // This gets called for every action you dispatch.  
    // If it's a function, call it.  
    if (typeof action === 'function') {  
      return action(dispatch, getState, extraArgument);  
    }  
  
    // Otherwise, just continue processing this action as usual  
    return next(action);  
  };  
}  
  
const thunk = createThunkMiddleware();  
thunk.withExtraArgument = createThunkMiddleware;  
  
export default thunk;
```

Redux-Thunk

- In Redux sind Actions nur Objekte, mit einem Type und dem Payload
- Die Software-Komponente Redux-Thunk ist eine Erweiterung für Redux, um anstelle von Objekten nun Funktionen an den Store zu senden
- Die einzige Aufgabe von Redux-Thunk ist: wenn in der Action eine Funktion ist, führt Thunk die Funktion aus
- Dafür ist das Registrieren von Thunk erforderlich

```
import { applyMiddleware, createStore } from 'redux';
import thunk from 'redux-thunk';

import rootReducer from './rootReducer';
import initialState from './initialState';

const middlewares = [thunk];

createStore(rootReducer, initialState, applyMiddleware(...middlewares));
```


Redux-Thunk: Beispiel

- Das Abrufen von Produkten wird in mögliche 3 Actions unterteilt
 1. Das Abrufen wird gestartet, der Status ist „pending“
 2. Das Abrufen war erfolgreich
 3. Beim Abrufen ist ein Fehler aufgetreten

```
export const FETCH_PRODUCTS_PENDING = 'FETCH_PRODUCTS_PENDING';
export const FETCH_PRODUCTS_SUCCESS = 'FETCH_PRODUCTS_SUCCESS';
export const FETCH_PRODUCTS_ERROR = 'FETCH_PRODUCTS_ERROR';

function fetchProductsPending() {
  return { type: FETCH_PRODUCTS_PENDING }
}
function fetchProductsSuccess(products) {
  return { type: FETCH_PRODUCTS_SUCCESS products: products
  }
}
function fetchProductsError(error) {
  return { type: FETCH_PRODUCTS_ERROR error: error}
}
```

Redux-Thunk: Beispiel

- Der zuständige Reducer
 - Abrufen wird gestartet: Der aktuelle State wird zurückgegeben, der Status ist „pending“
 - Der Abruf war erfolgreich: die Produkte werden zum State hinzugefügt, „pending“ ist false
 - Beim Abruf gab es einen Fehler: der State bleibt gleich, „pending“ ist false

```
export function productsReducer(state = initialState, action) {  
  switch(action.type) {  
    case FETCH_PRODUCTS_PENDING:  
      return { ...state, pending: true }  
    case FETCH_PRODUCTS_SUCCESS:  
      return { ...state, pending: false, products: action.payload }  
    case FETCH_PRODUCTS_ERROR:  
      return { ...state, pending: false, error: action.error }  
    default:  
      return state;  
  }  
}
```

Redux-Thunk: Beispiel

- Die Funktion: die `dispatch()`-Methode wird 3 mal aufgerufen und bildet den Prozess ab

```
import {fetchProductsPending, fetchProductsSuccess, fetchProductsError} from 'actions';

function fetchProducts() {
  return dispatch => {
    dispatch(fetchProductsPending());
    fetch('https://exampleapi.com/products')
    .then(res => res.json())
    .then(res => {
      if(res.error) { throw(res.error); }
      dispatch(fetchProductsSuccess(res.products));
      return res.products;
    })
    .catch(error => {dispatch(fetchProductsError(error)); })
  }
}

export default fetchProducts;
```

Redux-Thunk: Beispiel

- Registrieren der Action-Function

```
const mapDispatchToProps = dispatch => bindActionCreators({
  fetchProducts: fetchProductsAction
}, dispatch)

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(ProductView);
```

- Wenn Thunk noch nicht registriert ist, sehen Sie diese Antwort

Error: Actions must be plain objects. Use custom middleware for async actions.

► 2 stack frames were collapsed.

componentWillMount
C:/Git/de.coservices.daif.core.react/de.coservices.daif.core.react/src/tutorials/products/ProductView.js:21

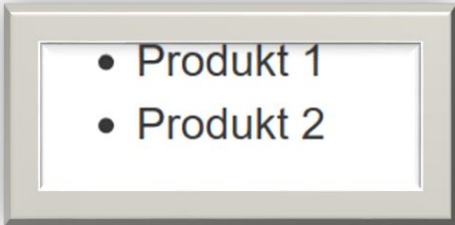
```
18 |  
19 |   componentWillMount() {  
20 |     const {fetchProducts} = this.props;  
> 21 |     fetchProducts();  
22 |   }  
23 |  
24 |   shouldComponentRender() {
```

Redux-Thunk: Beispiel

- Damit Funktionen in Actions ausgeführt werden können, muss Redux-Thunk als Middleware hinzugefügt werden

```
import { applyMiddleware, createStore } from 'redux';  
import thunk from 'redux-thunk';  
  
import rootReducer from './rootReducer';  
import initialState from './initialState';  
  
const middlewares = [thunk];  
  
createStore(rootReducer, initialState, applyMiddleware(...middlewares));
```

Redux-Thunk: Beispiel

- 
- Produkt 1
 - Produkt 2

- Es soll eine Produktliste dargestellt werden
- Sobald die Seite dargestellt wird, soll ein Request an den REST-Server geschickt werden, um die Produkte abzurufen
- Solange der Request läuft, soll „Lade Daten“ angezeigt werden
- Sobald das Ergebnis da ist, soll die Liste dargestellt werden
- Umsetzungsbeispiel: die Imports in der Komponente

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { bindActionCreators } from 'redux';

import fetchProductsAction from './FetchProducts';
import { getProductsError, getProducts, getProductsPending } from './Reducer';

import LoadingSpinner from './SomeLoadingSpinner';
import ProductList from './ProductList';
```

Redux-Thunk: Beispiel

- Es soll eine Produktliste dargestellt werden
- Die Methode `shouldComponentRender` soll unterscheiden, ob der Spinner oder die Liste dargestellt wird: sie muss an die Komponente gebunden werden
- `componentDidMount()` wird aufgerufen, sobald die Komponente dargestellt wurde
- Sobald das passiert ist, sollen die Produkte abgerufen werden

```
...  
class ProductView extends Component {  
  constructor(props) {  
    super(props);  
    this.shouldComponentRender = this.shouldComponentRender.bind(this);  
  }  
  componentDidMount() {  
    const { fetchProducts } = this.props;  
    fetchProducts();  
  }  
}
```

Redux-Thunk: Beispiel

- Die Methode `shouldComponentRender()` prüft, ob im State das „pending“-Flag gesetzt ist
- Wenn die Anfrage läuft („pending“) oder kein Wert gesetzt ist, wird „false“ zurückgegeben, ansonsten true

```
...  
  shouldComponentRender() {  
    const { pending } = this.props;  
    if (pending === false) {  
      return true;  
    }  
    else {  
      return false;  
    }  
  }  
}  
...
```


Redux-Thunk: Beispiel

- Wenn die Anfrage läuft, wird die Spinner-Komponente zurückgegeben, ansonsten die Produktliste, der die Produkte übergeben werden

```
...
render() {
  const { products } = this.props;

  if (!this.shouldComponentRender()) {
    console.log('ProductView: Render the Spinner')
    return <LoadingSpinner />
  }
  else {
    console.log('ProductView: Render the product list')
    return (
      <ProductList products={products} />
    )
  }
}
...
```

Redux-Thunk: Beispiel

- Da die Komponenten Actions verschickt und die Ergebnisse erhält, muss sowohl `mapStateToProps` als auch `mapDispatchToProps` registriert werden

```
...
const mapStateToProps = state => ({
  error: getProductsError(state),
  products: getProducts(state),
  pending: getProductsPending(state)
})

const mapDispatchToProps = dispatch => bindActionCreators({
  fetchProducts: fetchProductsAction
}, dispatch)

export default connect(mapStateToProps, mapDispatchToProps )(ProductView);
```