Front-End: React – Teil 2

Web-Engineering II

Prof. Dr. Sebastian von Klinski

Formulare

Formulare

- Formulare sind in der Regel Klassenkomponenten, da State benötigt wird
- Komponenten, die Formulare umsetzen, müssen…
  - Änderungen in den Formularfeldern übernehmen
  - Den State ändern und die anderen Komponenten (und ggfls. das Backend) informieren, wenn Formular abgeschickt wird
- Formularfelder werden im State angelegt/ initialisiert

```
import React, { Component } from 'react'

class Form extends Component {
  initialState = {
    name: '',
    job: '',
  }
  state = this.initialState
}
```

Formulare: Übernehmen von Änderungen

- Automatisches Übernehmen von Werten in Formularfeldern durch onChange-Methode von HTML-Elementen
- Dazu muss…
  - Methode zum Umgang mit Änderungen in Komponente implementiert werden (z.B. handleChange())
  - Methode muss an HTML-Formularelemente gebunden werden
- Geänderte Werte werden mit setState() in den State übernommen

```
handleChange = event => {
  const { name, value } = event.target

  this.setState({
    [name]: value,
  })
}
```

Formulare

- An die beiden Input-Felder wird die Methode handleChange() gebunden
- Mit dem value-Attribut wird die Ausgabe im Formular automatisch an den Wert im State gebunden

```
render() {
  const { name, job } = this.state;

  return (
    <form>
      <label htmlFor="name">Name</label>
      <input type="text" name="name" id="name" value={name}
        onChange={this.handleChange} />
      <label htmlFor="job">Job</label>
      <input type="text„ name="job„ id="job„ value={job}
        onChange={this.handleChange} />
    </form>
  );
}
```

Formulare: Submit

- Wenn der Submit-Button im Formular gedrückt wird, muss der State in der Klasse geändert werden
- Notwendige Schritte
  - Methode zum Ändern des State implementieren
  - Methode muss zum Formular runtergereicht werden

```
handleSubmit = character => {
    this.setState({ characters: [...this.state.characters, character] })
}
render() {
    const { characters } = this.state
    return (
        <div className="container">
            <Table characterData={characters} removeCharacter={this.removeCharacter} />
            <Form handleSubmit={this.handleSubmit}/>
        </div>
    )
}
```

Formulare: Submit

- Bei einem Submit werden die eingegebenen Daten „hochgereicht“
- Anschließend werden die Formularfelder geleert

```
…
  submitForm = () => {
   this.props.handleSubmit(this.state)
   this.setState(this.initialState)
  }
…
 return (
      <form>
        …
        <input type="text" name="job„ id="job„ value={job}
          onChange={this.handleChange} />
          <input type="button" value="Submit" onClick={this.submitForm} />
      </form>
    );
…
```

Event-Handling: HTML und React

- Event-Handling in React ähnlich wie in HTML, aber leicht anders!
- HTML

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

- In React

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

- Methode in geschweiften Klammern und ohne runde Klammern

Event-Handling: HTML und React

- Standardverhalten von HTML-Elementen wird auch unterschiedlich unterdrück (z.B. Wechsel zu einer Seite bei einem Link)
- HTML: Rückgabe von „false“ bei Event-Funktion

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

- In React: Beim Event muss preventDefault() aufgerufen werden

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');  }
  return (
    <a href="#" onClick={handleClick}>      Click me
    </a>
  );
}
```

Event-Handling: Binding von Methode in React

- Bei Verwendung von JSX muss die Methode in Komponente mit bind() an „this“ (Instanz) gebunden werden! Ansonsten ist die Funktion „undefined“.

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
      this.setState(state => ({      isToggleOn: !state.isToggleOn    }));
  }
  render() {
    return (
      <button onClick={this.handleClick}>{this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

Event-Handling: Binding von Methode in React

- **Alternative 1 zum Binding: Arrow-Functions**
- Das Binding der Methode an die Instanz ist nur notwendig bei regulären Funktionsdeklarationen
- Wird die Methode als Arrow-Function deklariert, ist das Binding nicht notwendig
- Arrow-Functions werden automatisch an „this“ gebunden

```
class LoggingButton extends React.Component {
  handleClick = () => {
   console.log('this is:', this);
   }
  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

Event-Handling: Binding von Methode in React

- **Alternative 1 zum Binding: Arrow-Functions in HTML-Komponente**
- Wenn Methodenaufruf in HTML-Komponente als Arrow-Function geschrieben wird, ist ebenfalls ein Binding nicht notwendig
- Nachteil: Callbacks werden immer wieder neu erzeugt → unnötige Aufrufe für Aktualisierung von Web-Seite

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }
  render() {
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

Event-Handling: Parameter in Schleifen

- In Listen muss häufig Parameter zur Identifikation des Eintrags an Methode übergeben werden
- Mögliche Umsetzung:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

Event-Handling: Parameter in Schleifen

- Beispiel bei Tabelle mit Einträgen, die per Button gelöscht werden

```
const TableBody = props => {
    const rows = props.characterData.map((row, index) => {
        return (
            <tr key={index}>
                <td>{row.name}</td>
                <td>{row.job}</td>
                <td>
                    <button onClick={() => props.removeCharacter(index)}>Delete</button>
                </td>
            </tr>
        )
    })

    return <tbody>{rows}</tbody>
}
```

Hinweis: map-Funktion

- Die map() Methode wendet auf jedes Element des Arrays die bereitgestellte Funktion an und gibt das Ergebnis in einem neuen Array zurück.

```
const TableBody = props => {
    const rows = props.characterData.map((row, index) => {
        return (
            <tr key={index}>
                <td>{row.name}</td>
                <td>{row.job}</td>
                <td>
                    <button onClick={() => props.removeCharacter(index)}>Delete</button>
                </td>
            </tr>
        )
    })

    return <tbody>{rows}</tbody>
}
```

Nutzen von REST-APIs

REST-Einbindung

- Für das Gestalten von React-Anwendungen werden in der Regel REST-APIs verwendet
- Grundsätzlich können allle REST-APIs verwendet werden (z.B. Axios, jQuery AJAX, das im Browser vordefinierte window.fetch, etc.)
- Üblicherweise wird der AJAX-Call beim Aufbau der Klassenkomponente ausgeführt: **componentDidMount()**
- Das Ergebnis wird dann verwendet, um mit setState() den Zustand zu setzen
- Nach setState() wird die Komponente automatisch aktualisiert

REST-Einbindung

- State zunächst leer initialisieren
- In componentDidMound() wird REST-Services aufgerufen

```
import React, { Component } from 'react'
class App extends Component {
  state = {
    data: [],
  }
  componentDidMount() {
    const wikiURL = 'https://ghibliapi.herokuapp.com/films'
    fetch(wikiURL)
      .then(result => result.json())
      .then(result => {
        this.setState({
          data: result,
        })
      })
  }
```

REST-Einbindung

- Innerhalb von React-Komponenten kann die native Fetch-API des Browsers verwendet werden (Standardfunktion in JavaScript)
- fetch() gibt Promise zurück
- Wenn Spinner für das Laden angezeigt werden soll, kann vorher ein Flag im State gesetzt werden

```
…
componentDidMount() {
    this.setState({ isLoading: true });

    fetch('https://ghibliapi.herokuapp.com/films')
      .then(response => response.json())
      .then(data => this.setState({ hits: data.hits, isLoading: false }));
  }
…
```

REST-Einbindung

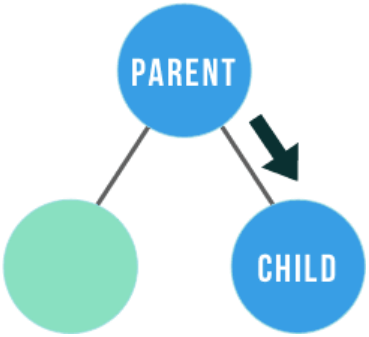
- Ergebnis des REST-Calls ist in der Regel ein Object-Array, der als JSON vom Server kam
- Mit dem Object-Array kann dann ein Element-Array erzeugt werden, der dann per JSX eingebaut werden kann
- Hinweis: in React sollten Elemente in Listen möglichst immer eine ID/ Key haben!

```
render() {
   const { data } = this.state

   const result = data.map((entry, index) => {
     return <li key={entry.id}>{entry.title}</li>
   })

   return <ul>{result}</ul>
 }
```

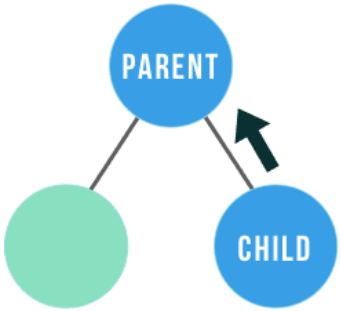
Kommunikation zwischen Komponenten

Kommunikation zwischen Komponenten

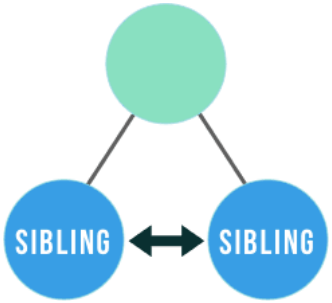
- Umsetzung der Kommunikation zwischen den Komponenten ist abhängig von der Konstellation der Kommunikationspartner



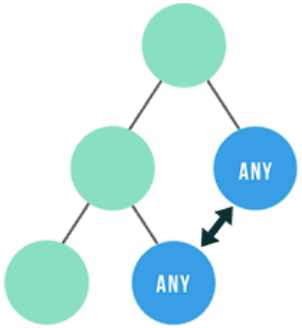
Parent to Child
- Props
- Instanz-Methoden



Child to Parent
- Callback
- Event-Bubbling



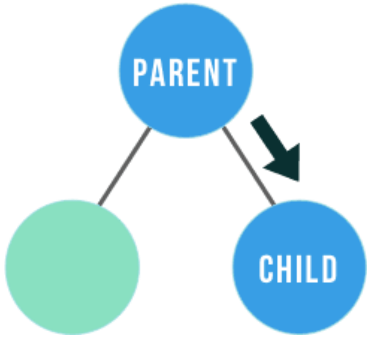
Child to Child
- Parent-Component



Any to Any
- Observer Pattern
- Globale Variablen
- Context

Quelle: https://www.javascriptstuff.com/component-communication/

Parent-To-Child: Props



- Zentraler Mechanismus in React für Weitergabe von Daten und Methoden

```
<Welcome name="Sara" />
```

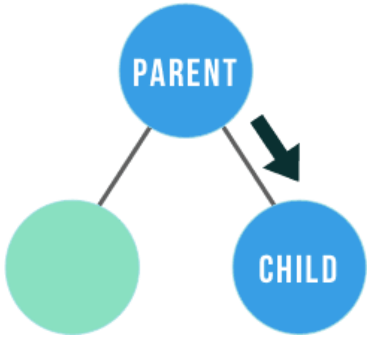
- Bei Funktionskomponenten

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- Bei Klassenkomponenten

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Parent-To-Child: Instanzmethoden

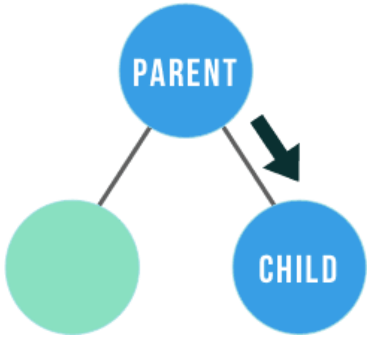


- Wenn der Parent per Methode mit dem Child kommunizieren soll
- Beim Anlegen der Children werden die Referenzen im State abgelegt und können aufgerufen werden
- Komponente mit Funktion, die aufgerufen werden soll:

```
class TheChild extends React.Component {
  myFunc() {
    return "hello";
  }
}
```

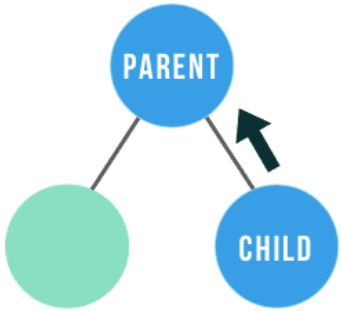
Parent-To-Child: Instanzmethoden

- Im Parent wird beim Anlegen mit „ref=„ eine Referenz angelegt und kann im State abgelegt werden
- Eher seltener Ansatz für Kommunikation




```
class TheParent extends React.Component {
  render() {
    return (
      <TheChild ref={foo => {this.foo = foo; }} />
    );
  }
  componentDidMount() {
    var x = this.foo.myFunc();
    // x is now 'hello'
  }
}
```

Child-To-Parent: Callback



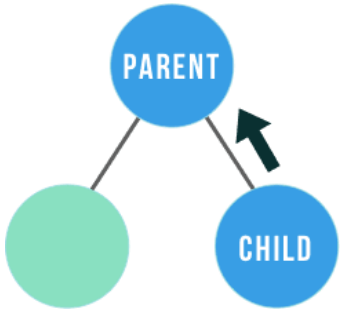
- Der gängige Kommunikationsansatz vom Child zum Parent
- Übergabe der Callback-Methode:
  - Klassenkomponente definiert Funktion
  - Beim Anlegen von Child-Komponenten wird denen die Referenz auf die Funktion übergeben

```
<MyChild myFunc={this.handleChildFunc} />
```

- Die Child-Komponenten kann sich die Referenz auf die Funktion aus den Props holen und aufrufen

```
this.props.myFunc();
```

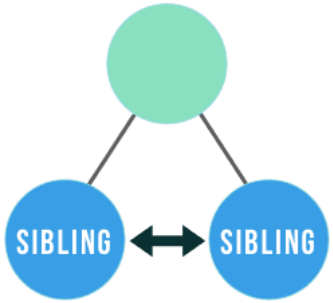
Child-To-Parent: Event Bubbling



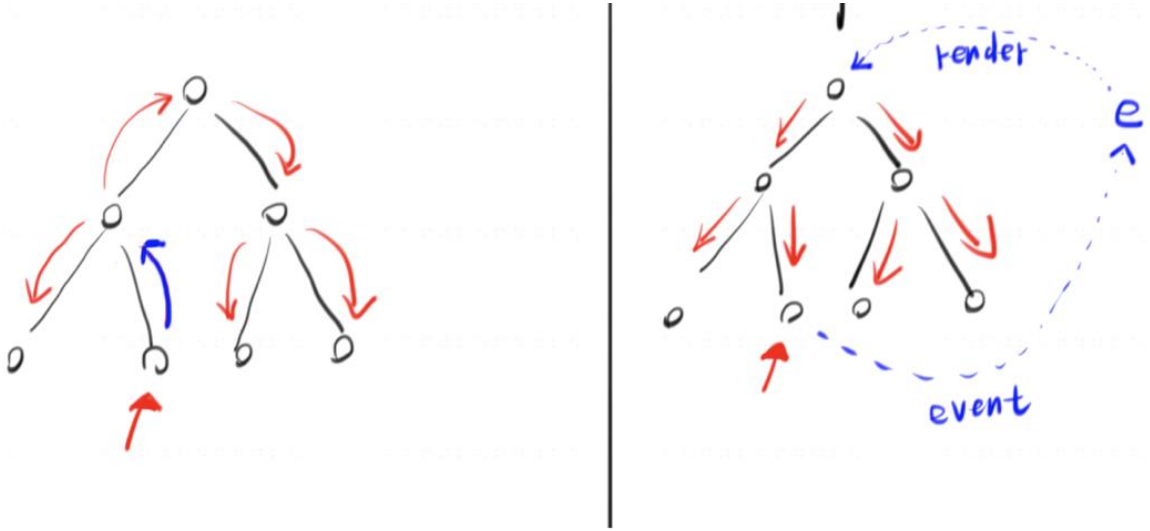
- Kein React-spezifischer Kommunikationsansatz, sondern klassischer DOM-Ansatz
- Im Parent können schon Event-Handler an Child-Komponente angehangen werden
- Eher selten verwendet

```
class ParentComponent extends React.Component {
  render() {
    return (
      <div onKeyUp={this.handleKeyUp}>
        …
      </div>
    );
  }
  handleKeyUp = (event) => {
    …
  }
}
```

Zwischen Child-Komponenten

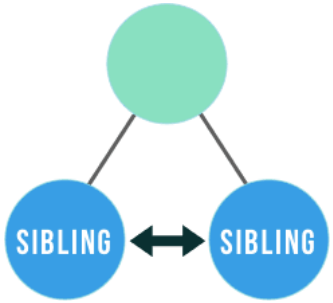


- Direkte Kommunikation zwischen Child-Komponenten schwierig, weil sie keine Referenzen aufeinander haben
- In der Regel übernimmt Root-Komponente die Kommunikation
  - Benachrichtigung, der Root-Komponente
  - Root-Komponente aktualisiert den eigenen State und bei Aktualisierung in render() die Props in den Child-Komponenten



Quelle: https://medium.com/@haixiang6123/react-js-component-communication-between-sibling-components-1fdd21328c64

Zwischen Child-Komponenten

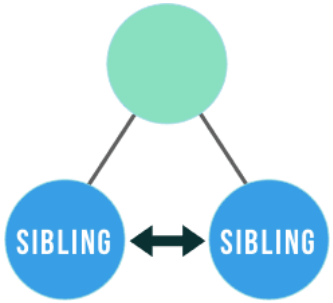


- Methode in der Parent-Komponente definieren und weiterreichen
- Wenn Komponenten informieren wollen, rufen Sie die Funktion auf
- setState() führt zu Rendering von Komponenten

```
class SessionApp extends Component {
   constructor(props) {
      super(props);
      this.state = { currentUser: "" };
      this.handleLogin = this.handleLogin.bind(this);
   }
   handleLogin () {
      if (localStorage != null) {
         const userID = localStorage.getItem(Constants.USER_ID);
         this.setState({currentUser: userID});
      }
   }
…
```

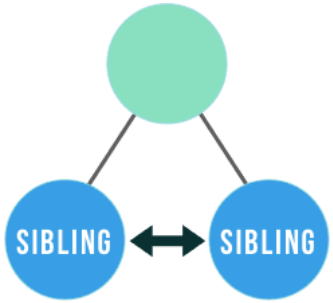
Hinweis: Umsetzen von Nachrichten

- Wenn sich Komponenten gegenseitig informieren, sollten Konstanten verwendet werden, um die Nachrichtenart eindeutig zu identifizieren
- Diese Konstanten sollten in zentraler Datei ausgelagert werden.
- Mögliche Umsetzung: Modul mit entsprechenden Exports




```
module.exports = {
    USER_LOGGED_IN: 'UserLoggedIn',
    THEME_CHANGED: 'ThemeChanged',
    SELECTION_CHANGED_EVENT: 'SelectionChanged',
}
```

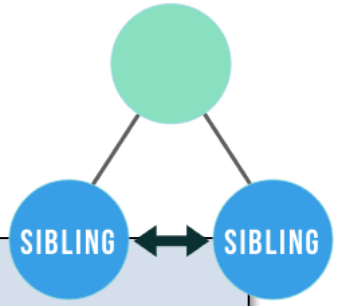
Zwischen Child-Komponenten



- Funktionen zum Ändern des States werden an Komponenten mit Interaktionsmöglichkeiten weitergereicht
- Wenn Methode durch Child-Komponente aufgerufen wird, wird der State in Parent-Komponente geändert
- Dadurch wird Komponente aktualisiert und Änderungen an Child-Komponenten weitergereicht

```
…
render() {
        const userName = this.state.currentUser;
        return (
            <div>
                <SessionLoginComponent loginChanged={this.handleLogin} />
                <SessionWelcomeText currentUser={userName} />
            </div>
        )}
 …
```

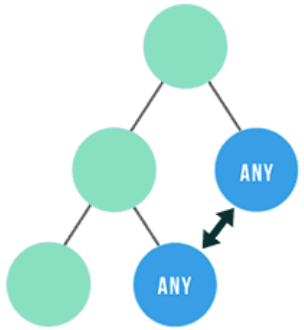
Zwischen Child-Komponenten



- Aufrufen der Parent-Methode

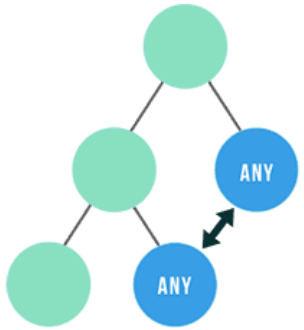
```
class LoginComponent extends Component {

  handleLogin(e) {
    … //Hier ist der Login-Prozess
    if (localStorage != null) {
      localStorage.setItem(Constants.USER_ID, this.state.userID);
    }
    this.loginChanged();
  }
  render() {
    …
    return (
      <form>
        …
        <input type="button" value="Submit" onClick={this.handleLogin} />
      </form>
    …
```

Zwischen beliebigen Komponenten

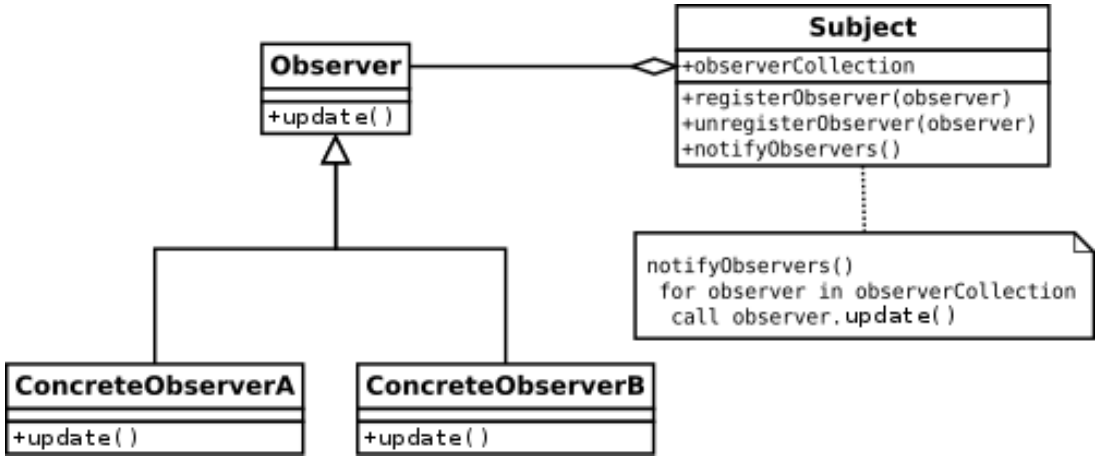


- Wenn der Komponentenbaum sehr tief ist und/ oder die Child-Komponenten keine unmittelbare Parent-Komponente gemeinsam haben, wird Kommunikation über Referenzen schwierig
- In diesem Fall wird eine Kommunikationsmöglichkeit unabhängig von den Parent-Komponenten benötigt
- Mögliche Umsetzungsoptionen
  - Observer Pattern
  - Global Variables
  - Context

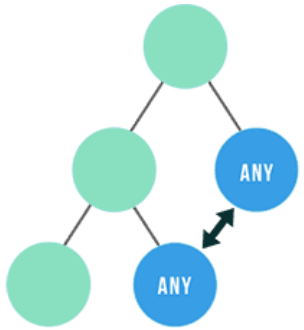
Observer-Pattern



- Gängiges Programmiermuster (Pattern) in objektorientierter Programmierung
- Lose gekoppelte Kommunikation zwischen
  - Observable (Objekt, das sich ändert) und
  - Observer (Objekt, das über Änderungen informiert wird)
- Observer ist Interface mit update()-Methode
- Observable ist Klasse, bei dem sich Observer per addObserver() registrieren kann.

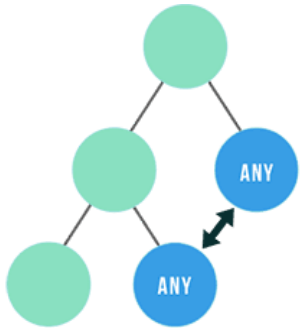


Observer-Pattern



- Umsetzung von Observer-Pattern über Bibliotheken möglich
- Gängige Bibliotheken: EventEmitter, PubSubJS, MicroEvent.js, mobx
- Am weitesten verbreitet: EventEmitter
  - Zentrales Objekt zum Registrieren von Listenern
  - Methode zum Versenden von Events
  - Emitter verteilt Events an entsprechende Listener
- Schritte
  1. Definieren eines Singleton-Modules mit dem Event-Emitter
  2. Registrieren von Listenern beim Event-Emitter
  3. Versenden von Events von den Komponenten, die einen Zustandswechsel veranlassen

Hinweis: Singleton

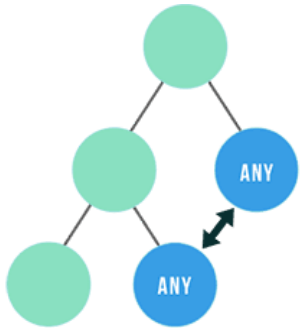


- Singleton ist Software-Pattern, bei dem von einem Objekt nur eine Instanz angelegt werden soll
- Andere Objekte können sich direkt die Instanz holen, ohne dass die Referenz rumgereicht werden muss

Singleton in Java

```
public class Singleton {
   static private Singleton instance_ = null;
   static public Singleton instance()
   {
      if (null == instance_)
      {
         instance_ = new Singleton();
      }
      return instance_;
   }
   protected Singleton()
   {
    // nur um Zugriff zu verhindern
   }
}
```

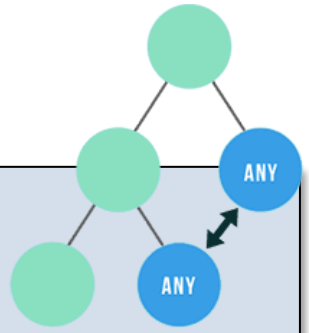
Observer-Pattern mit EventEmitter



- Methoden Hinzufügen und Entfernen von Listenern sowie dem Versenden von Listener
- Object.freeze() stellt sicher, dass Emitter nicht geändert wird

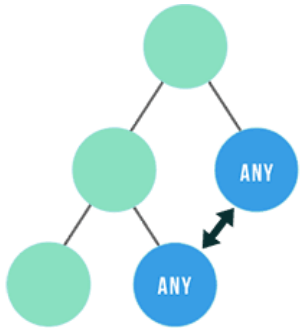
```
import EventEmitter from 'eventemitter3';

const eventEmitter = new EventEmitter();

const Emitter = {
  addListener: (event, fn) => eventEmitter.on(event, fn),
  removeListener: (event, fn) => eventEmitter.off(event, fn),
  sendEvent: (event, payload) => eventEmitter.emit(event, payload)
}

Object.freeze(Emitter);

export default Emitter;
```

Observer-Pattern: Registrieren von Listener




```
import Emitter from './SessionEventEmitter';
…
class WelcomeText extends Component {
  constructor(props)
  {
    super(props);
    this.handleEmitterEvent = this.handleEmitterEvent.bind(this);
    this.state = { username: props.userID};
  }
  componentDidMount() {
    Emitter.addListener(Constants.LOGIN_CHANGED_EVENT, (newValue) =>
          this.handleEmitterEvent(newValue));
  }
  componentWillUnmount() {
    Emitter.removeListener(Constants.LOGIN_CHANGED_EVENT);
  }
  handleEmitterEvent(newValue) {
      this.setState({username: newValue});
  }
…
```

Observer-Pattern: Versenden von Events

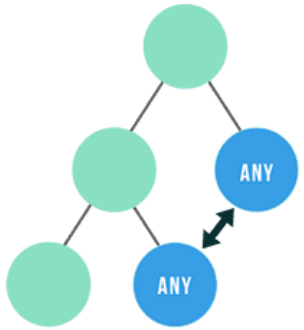


- Beim Versenden der Events werden die Art des Events und die relevanten Daten übergeben
- Listener sollten nur auf spezifische Events reagieren

```
import Emitter from './SessionEventEmitter';
…
class LoginComponent extends Component {
  constructor(props) {
    super(props);
    this.handleLogin = this.handleLogin.bind(this);
}
 handleLogin(e) {
  …
  Emitter.sendEvent(Constants.LOGIN_CHANGED_EVENT, this.state.userID);
  }
```

- Der Empfang eines Events sollte keine weiteren Events auslösen, sonst besteht die Gefahr von Endlosschleifen!

Alternative 2: Globale Variablen

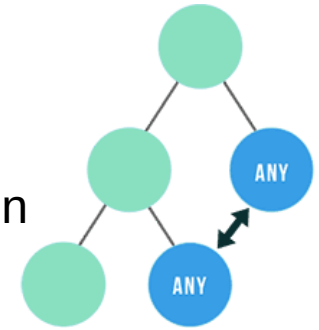


- Globale Variablen können im Webanwendungen beispielsweise im window-Objekt angelegt werden.

```
…
componentWillMount: function () {
    window.MyVars = {
        theme: „dark“,
    };
}
…
```

- Es wird jedoch grundsätzlich davon abgeraten, globale Variablen zu verwenden
- Seiteneffekte sind bei größeren Anwendungen schwer abzuschätzen
- Außerdem: Änderungen an Variablen lösen kein redraw() bei relevanten Komponenten aus

Alternative 3: Context

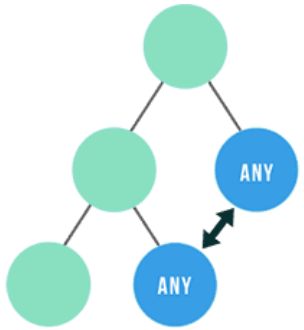


- Context ist React-Konzept vergleichbar zu globalen Variablen
- Context wird mit einem Standardwert für einen Teil des Komponentenbaum deklariert

```
…
const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {
     return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
…
```

- Child-Komponenten können auf Context-Objekt zugreifen, ohne dass es per Props weitergeleitet wurde

Alternative 3: Context



- In Child-Komponenten kann über „this.context“ auf Werte zugegriffen werden
- Einsatz von Context nur in Ausnahmefällen empfohlen

```
…
class ThemedButton extends React.Component {
   static contextType = ThemeContext;
   render() {
    return <Button theme={this.context} />;
   }
}
…
```

- Einsatz nur empfohlen, wenn viele Komponenten die Daten benötigen und ein Weiterreichen aufwändig wäre (z.B. Festlegen von Theme)
- Achtung: Änderungen im Context führt zu einem Neu-Rendern von allen betroffenen Komponenten

Kommunikation zwischen Komponenten

- Je nach Art der Kommunikation können unterschiedliche Ansätze verwendet werden
- Ein einheitlicher Ansatz ist in React zunächst nicht vorgesehen
- Eine direkte Kommunikation zwischen verteilten Komponenten, die keine Referenzen aufeinander haben, ist ohne Zusatz-Software-Komponenten schwierig
- In der Regel Umsetzung über andere Software-Komponenten wie Redux

Bedingte Ausgabe

Bedingte Ausgabe

- Häufig soll die Ausgabe von Komponenten an bestimmte Bedingungen geknüpft werden
- Beispielsweise: ein User ist eingeloggt oder kein User ist eingeloggt

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}
function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn)
  {
     return <UserGreeting />;
  }
  return <GuestGreeting />;
}
```

Bedingte Ausgabe für Login-Button

- Mögliche Umsetzung für Login-Komponente

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}
```

Bedingte Ausgabe für Login-Control

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  …
```

Bedingte Ausgabe für Login-Control

```
…
  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;
    if (isLoggedIn)
    {
       button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else
    {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }
    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />{button}
      </div>
    );
  }
}
```

Bedingte Ausgabe in JSX

- Es können auch direkt im JSX-Code bedingte Ausdrücke integriert werden.

```
function Mailbox(props) {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

```
return (
    <div>
      {isLoggedIn ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />      }
   </div>
);
```

Bedingte Ausgabe in JSX

- Durch die Rückgabe von null können Komponenten auf Basis von Bedingungen ausgeblendet werden.

```
function WarningBanner(props) {
  if (!props.warn)
  {
     return null;
  }
  return (
    <div className="warning">
      Warning!
    </div>
  );
}
```

React Router

React Routen

- React setzt grundsätzlich Single Page Applicaton um
  - Seite wird einmal geladen
  - Inhalt der Seiten wird auf Basis der Interaktionen geändert
  - URL bleibt immer gleich
- Zuweilen ist aber Verwendung von Routen sinnvoll
  - www.seite.de/impressum
  - www.seite.de/ueberuns
- Besonderheit bei SPA
  - Wechsel zwischen Routen soll nicht zu einem Reload der Seiten führen
  - Keine Links mit href möglich
- Zum Umsetzen solcher Routen kann man React-Router verwenden
  - Installation: npm install react-router-dom

React-Router

- Anwendung muss in BrowserRouter-Tag eingeschlossen werden
- Notwendig, damit Sub-Komponenten Links zum Wechsel zwischen Routen verwenden können

```
…
import { BrowserRouter } from 'react-router-dom';
…
ReactDOM.render(
  <BrowserRouter>
      <App />
  </BrowserRouter>,
  document.getElementById('root')
);
```

React-Router

- Zum Wechsel zwischen unterschiedlichen Routen wird Switch-Tag verwendet

```
…
import Home from './components/Home';
import About from './components/About';
import Shop from './components/Shop';
import { Switch, Route } from 'react-router-dom';
…
function App() {
    return (
        <main>
            <Switch>
                <Route path="/" component={Home} exact />
                <Route path="/about" component={About} />
                <Route path="/shop" component={Shop} />
            </Switch>
        </main>
    )
}
….
```

React-Router

- Für die Links müssen Router-Links verwendet werden!
- Keine normalen Links mit href möglich, weil sonst Seite neu geladen wird (bei SPA nicht erwünscht/ problematisch)

```
…
import { Link } from 'react-router-dom';
…
function Navbar() {
  return (
    <div>
      <Link to="/">Home </Link>
      <Link to="/about">About Us </Link>
      <Link to="/shop">Shop Now </Link>
    </div>
  );
};
```

React-Router und Bootstrap

- Für Nutzung von Bootstrap und React-Router wird weiteres Modul benötigt
  - Design soll von Bootstrap, Link-Handling von Router kommen
  - npm install -S react-router-bootstrap
- Komponenten, die Link-Actions haben, werden in Link-Container eingebettet

```
…
import { LinkContainer } from 'react-router-bootstrap‘
…
<LinkContainer to="/foo/bar">
  <Button>Foo</Button>
</LinkContainer>
…
```

React-Router und Bootstrap Navigation-Bar

- Für Navigation-Bar sind 2 Ansätze möglich
  - Einbettung eines Router-Links in Navbar.Brand-Tag
  - Einbettung von Nav.Link in Router-LinkContainer

```
…
 render() {
        return (
                <Navbar bg="light" expand="lg">
                   <Navbar.Brand href="/">
                      <Link to="/">Home </Link>
                   </Navbar.Brand>
                   <Navbar.Toggle aria-controls="basic-navbar-nav" />
                   <Navbar.Collapse id="basic-navbar-nav">
                      <Nav className="mr-auto">
                         <LinkContainer to="/aboutUs">
                            <Nav.Link eventKey={1}>Über uns</Nav.Link>
                         </LinkContainer>
                         …
                      </Nav>
```