tarent solutions GmbH

SPA mit vue.js -
ohne Node oder WebPack

tarent

Ausgangssituation

Ich wollte doch nur eine moderne Seite...

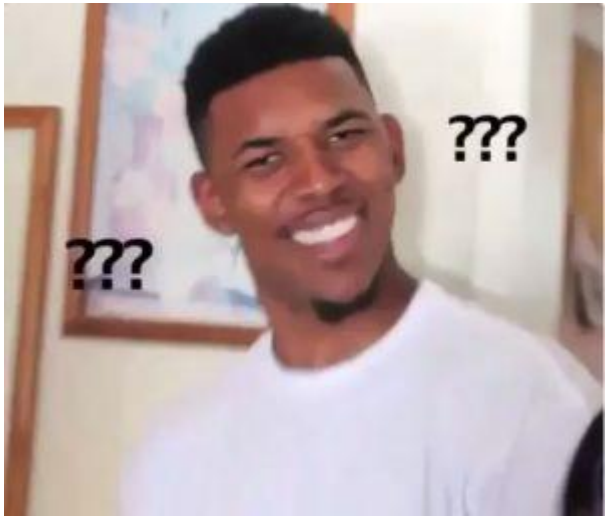
- Backend-Entwickler
  - Java Spring Anwendung
  - Thymeleaf als Template-Engine
  - Standard… :)
- Basics von HTML
- Basics von CSS
  - Bootstrap
- Basics von JavaScript
  - jQuery

Angesagte Frameworks, aber...

Angesagte Frameworks, aber...

Angesagte Frameworks, aber...

Angesagte Frameworks, aber...



Angesagte Frameworks, aber...

webpack

Was ich wollte...

- eine Library importieren
- etwas CSS-Zucker drüber streuen
- das via Java-Backend ausliefern (Ressourcen)
- und fertig!

Was ich nicht wollte...

- Mich erstmal mit NodeJS beschäftigen
- Fragen wie dieses NPM überhaupt funktioniert
- Verstehen was WebPack eigentlich macht
- Und was sind diese JSX-Dateien?!?

Vue.js

Vue.js

Warum?

- Konnte auf der **ersten** Seite überzeugen:
  - **<script src="https://unpkg.com/vue@3"></script>**
  - genauso wollte ich es haben!
- Dadurch schneller Einstieg
  - **keine** .jsx Dateien; nodeJS; npm; WebPack; ...
- Mittlerweile 204k Sterne auf Github
  - 210k Sterne für react
- Relativ leichtgewichtig
  - ~129Kb - vue3 minifiziert
  - ~86Kb - jQuery minifiziert
- Debug Tooling vorhanden

https://github.com/sschum/froscon-2023-demo/tree/main

Playground

- Alle Beispiele sind eingecheckt
- Jede Folie beinhaltet die URL zu dem entsprechenden Code
- Jede URL kann auch via QR-Code (unten rechts) gescannt werden
- Mittels start.sh wird ein Docker Container mit einem Nginx erstellt, welches die Website ausspielt
    - via http://localhost:1312 könnt ihr dann die Seite im Browser aufrufen
    - die Quelldateien sind in dem Container gemountet
        - Eine Quellcode Anpassung ist also direkt nach einem Browser-Reload sichtbar
- Jedes Vue-Kapitel ist in einem eigenen Branch eingecheckt
    - git checkout step-X

https://github.com/sschum/froscon-2023-demo/tree/main

Hello World

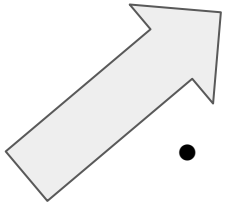
```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>FrOSCon - 2023 - Demo</title>

    <link href="./css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <h1>Hello World</h1>
</body>
</html>
```

- Nichts Besonderes:
  - index.html
  - bootstrap als CSS-Library - damit es nach etwas aussieht

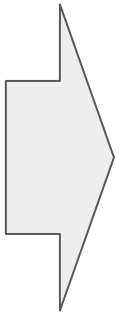
https://github.com/sschum/froscon-2023-demo/tree/step-01

Hello Vue

```
<head>
  …
  <script src="./js/vue.js"></script>
</head>

<h1>Hello World</h1>

<div id="app">
  {{ message }}
</div>

<script>
  const MyVueApp = Vue.createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }
    }
 })
 MyVueApp.mount('#app')
</script>
```



Gültigkeitsbereich von Vue



- Neue Vue-Instanz (**MyVueApp**) wird erstellt und an das DOM-Element mit der ID **app** verknüpft
- Diese Instanz beinhaltet eine Reihe von Daten, die in **data()** definiert sind
  - aktuell nur **message**
- Das DOM-Element wird geparst
  - es wird ein sog. virtueller DOM-Tree aufgebaut
  - Placeholder werden mit dem entsprechenden Wert ersetzt
    - {{ message }} -> ‘Hello Vue!’

https://github.com/sschum/froscon-2023-demo/tree/step-01

Hello Vue - Wo ist da jetzt die Magie?

https://github.com/sschum/froscon-2023-demo/tree/step-02

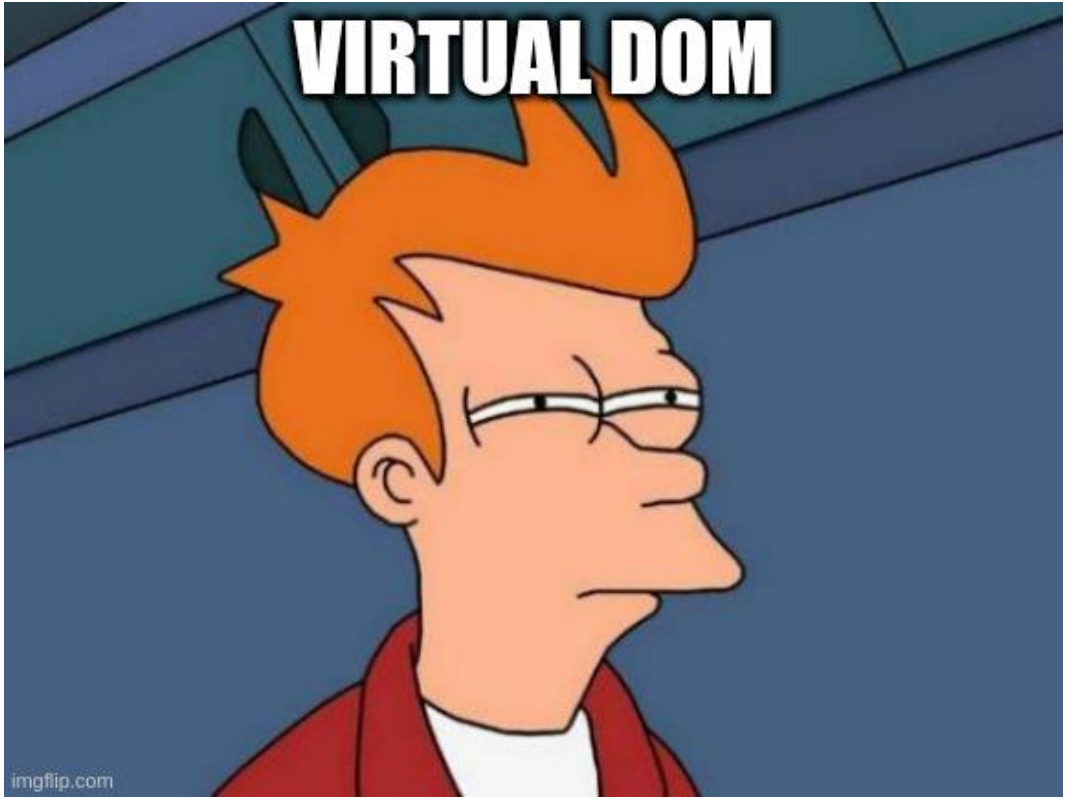
Erster Magic Moment

```
<div class="card" id="app">
  <div class="card-body">
    <h5 class="card-title">{{ message }}</h5>
    <p class="card-text">
      <input type="text" v-model="message" />
    </p>
  </div>
</div>

<script>
  const MyVueApp = Vue.createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }
    }
  })
  MyVueApp.mount('#app')
</script>
```

- v-model bindet das Feld **message** an das Input Element
- Sobald sich der Wert des Input-Elementes ändert, wird er in das Feld **message** übertragen
- Da sich sich der Wert des Feld **message** geändert hat, wird der Inhalt des h5 Elements neu gerendert
- Das ist die **Reaktivität**, von der alle sprechen :)
  - Auf eine **Änderung** wird **reagiert**

https://github.com/sschum/froscon-2023-demo/tree/step-02

Virtueller DOM



https://github.com/sschum/froscon-2023-demo/tree/step-02

Virtueller DOM

- Vue.js arbeitet mit einem virtuellen DOM
- Das kann man sich als Abbild des “echten” DOM vorstellen
- Änderungen in das “echte” DOM zurückspielen ist “teuer”
  - ganz besonders, wenn mehrere Änderungen stattfinden
- Somit werden Änderungen zuerst in den virtuellen DOM aufgenommen
- Nachdem “alle” Änderungen einer Aktion durchgespielt wurden, findet ein Abgleich zwischen virtuellen und “echten” DOM statt
- Aus diesem Grund sollte man keine Libraries mit benutzen, die den DOM direkt manipulieren

https://github.com/sschum/froscon-2023-demo/tree/step-02

Virtueller DOM

https://github.com/sschum/froscon-2023-demo/tree/step-03

Eventlistener

```
<div class="card" id="app">
  <div class="card-body">
    ...
    <a href="#" class="btn btn-primary"
       v-on:click="addCard">Add Card
    </a>
  </div>
</div>

<script>
  const MyVueApp = Vue.createApp({
    data(){ … },
    methods: {
      addCard() {
        alert(this.message)
      }
    }
  })
  MyVueApp.mount('#app')
</script>
```

- Eigene Methoden können in dem **methods**-Block definiert werden
- Diese können u.a. als Event-Handler verwendet werden
- Das Attribut **v-on:click** registriert einen Click-Handler für das a-Element
- Methoden haben Zugriff auf ihre Vue-Instanz
  - somit auch Zugriff auf die Daten
    - mittels **this** kann man darauf zugreifen

https://github.com/sschum/froscon-2023-demo/tree/step-04

Schleifen

```
<div class="card" id="app">
  …
  <div class="row">
    <div class="col-6" v-for="card of cards">
      <div class="card">
        <div class="card-body">
          <h5 class="card-title">{{ card }}</h5>
        …
</div>

<script>
  const MyVueApp = Vue.createApp({
    data() { return { … , cards: [] } },
    methods: {
      addCard() {
        this.cards.push(this.message)
      }
    }
  })
  MyVueApp.mount('#app')
</script>
```

- Listen können mittels **v-for** gerendert werden
- Dazu verwendet man einfach ein Array, über das man iterieren möchte
  - in diesem Fall **cards**
- Sobald sich die Element(e) in dem Array ändern, wird das entsprechende DOM-Element
  - erzeugt
  - verändert
  - oder gelöscht

https://github.com/sschum/froscon-2023-demo/tree/step-05

Schleifen (2)

```
<div class="card" id="app">
  …
  <div class="col-6" v-for="(card, i) of cards">
    <div class="card">
      <div :class="i % 2 === 0 ? 'bg-primary' : 'bg-secondary'">
        <div class="card-body">
          <h5 class="card-title">{{ card }}</h5>
        </div>
      </div>
    </div>
  </div>
  …
</div>
```

- Die Zählervariable kann optional verwendet werden
- “kleinerer” Operationen können **inline** definiert werden
- Ein Element-Attribut, welches mit “**:**” beginnt, wird als **Ausdruck** behandelt
  - ansonsten wird es 1:1 als Text wahrgenommen
- In diesem Fall wird dieser Mechanismus für das hin- und herschalten der CSS-Klassen verwendet

https://github.com/sschum/froscon-2023-demo/tree/step-06

Schleifen (3)

```
…
<div class="col-6" v-for="(card, i) of cards" :key="card.id">
  <div class="card" v-on:click="removeCard(i)">
    …
  </div>
</div>
…
<script>
  const MyVueApp = Vue.createApp({
  …
  methods: {
    addCard() {
      this.cards.push({
        id: new Date().getTime(),
        content: this.message
      })
    },
    removeCard(i) { this.cards.splice(i, 1) }
  }
 }) …
</script>
```

- Wenn sich das Array ändert muss die komplette Liste neu gerendert werden
  - teuer!
- Um das zu vermeiden muss jedem Element ein **eindeutiger Schlüssel** mitgegeben werden
  - dafür gibt es das **:key** Attribut
- Somit wird nur das Element aus dem DOM entfernt, welches auch wirklich aus dem Array entfernt wurde

https://github.com/sschum/froscon-2023-demo/tree/step-07

Computed Properties

```
…
<p>
  Cards: {{ cards.length }}
  Charcounter: {{ amountOfChars }}
</p>
…
<script>
  const MyVueApp = Vue.createApp({
    …
    computed: {
      amountOfChars() {
        if(this.cards.length === 0) return 0
        return this.cards
                .map(c => c.content.length)
                .reduce((pv, cv) => pv + cv)
      }
    },
  …
</script>
```

- Eine Computed Property ist eine **Funktion** innerhalb des **computed**-blocks, die ein Wert zurückliefert
- Diese hat Zugriff auf alle Daten der Vue-Komponente
- Der Wert, welcher geliefert wird, wird **zwischengespeichert**
- Sollten sich die **relevanten** zugrunde liegenden Daten ändern, wird die Funktion erneut aufgerufen
  - und (natürlich) alle **betroffenen** Teile neu gerendert
- In diesem Fall wird die Methode also **nicht** aufgerufen, wenn sich die message ändert

https://github.com/sschum/froscon-2023-demo/tree/step-08

Computed Properties (2)

```
…
<p>
  Cards: {{ cards.length }}
  Charcounter: {{ amountOfChars }}
  Average Chars: {{ averageCharsAmount }}
</p>
…
<script>
  const MyVueApp = Vue.createApp({
    …
    computed: {
      …
      averageCharsAmount() {
        if(this.amountOfChars === 0) return 0
        return this.amountOfChars / this.cards.length
     }
    },
  …
</script>
```

- Haben auch **Zugriff** auf alle **anderen** Computed Properties
- Auch diese Computed Property wird neu “berechnet” wenn sich die zugrunde liegenden Daten/Computed Properties **ändern**

https://github.com/sschum/froscon-2023-demo/tree/step-09

Watcher

```
<script>
  const MyVueApp = Vue.createApp({
    …
    data() { return {
      message: 'Hello Vue!',
      …
    }},
    watch: {
      message(newValue) {
        console.log("Message changed: ", newValue)
      }
    }
  …
</script>
```

- (Computed) Properties können mittels **Watcher** beobachtet werden
- Dabei muss eine **Methode** in dem **watch**-Block definiert werden
  - sie muss **exakt** so heißen, wie die zu beobachtende Property
- Diese Methode wird immer nach einer Änderung aufgerufen
- Der Rückgabewert spielt dabei keine Rolle

Was bisher geschah...

- **Trennung** zwischen Model und View
  - Es gibt zwar eine “render”-Methode, die Elemente dynamischer generieren kann (so wie in react.js)
    - aber ich empfehle es nicht :)
- Eine Model-Veränderung sorgt für das neu Rendern der entsprechenden View-Elemente
- Es ist klar definiert, was Daten und was Methoden sind
- **Vue** übernimmt die Aufgabe, die Änderungen darzustellen

Weiter geht’s

Components

- Möchte man Teile wieder verwenden, benutzt man Components
- Diese haben ihre eigenen
  - Daten
  - Methoden
  - Computed Properties
  - Watcher

https://github.com/sschum/froscon-2023-demo/tree/step-10

Components

```
MyVueApp.component('card', {
  template: `
<div class="card">
  <div>
    <div class="card-body">
      <h5 class="card-title"></h5>
    </div>
  </div>
</div>`
})
```

card.js

```
…
<div class="row">
  <div class="col-6"
       v-for="(card, i) of cards"
       :key="card.id">
    <card></card>
  </div>
</div>
…
<script>
  const MyVueApp = Vue.createApp({...})
</script>
<script src="./js/card.js"></script>
<script>
  MyVueApp.mount('#app')
</script>
```

index.html

- Via **component**-Funktion der **Vue-Instanz** können Komponenten registriert werden
- Dabei benötigt die Komponente einen **Namen** und ein **Template**
- Das Template ist **explizit** nur ein String und **keine** Funktion!
  - Denn sonst bricht man mit der strikten Trennung zwischen Model und View
- Diese Komponenten können dann in anderen Templates verwendet werden


WO SIND DIE DATEN?
imgflip.com

https://github.com/sschum/froscon-2023-demo/tree/step-11

Component - Properties

```
MyVueApp.component('card', {
  props: ['cssClass'],
  template: `
<div class="card">
  <div :class="cssClass">
    <div class="card-body">
      <h5 class="card-title"></h5>
    </div>
  </div>
</div>`
})
```

card.js

- In der Komponente kann man ein Array mit Properties (**props**) definieren
- Diese kann man dann via Attribut an der Komponente übergeben
  - CamelCase wird durch kebab-case ersetzt
- Die Property kann dabei wie eine **Computed Property** behandelt werden
  - ReadOnly

```
<div class="row">
  <div class="col-6" v-for="(card, i) of cards" :key="card.id">
    <card :css-class="i % 2 === 0 ? 'bg-primary' : 'bg-secondary'"></card>
  </div>
</div>
```

index.html

https://github.com/sschum/froscon-2023-demo/tree/step-12

Component - Slots

```
MyVueApp.component('card', {
  props: ['cssClass'],
  template: `
<div class="card">
  <div :class="cssClass">
    <div class="card-body">
      <h5 class="card-title">
        <slot></slot>
      </h5>
    </div>
  </div>
</div>`
})
```

card.js

- Mittels **Slots** kann in einer Komponente definiert werden, wo der **Komponenten-Body** eingefügt werden soll
- Slots können auch einen Namen haben
  - So kann man mehrere Bereiche definieren
    - in diesem Beispiel z.b. Header und Body der Card
  - In diesem Fall muss man explizit angeben, welcher Teil in welchen Slot gehört
  - In dem Beispiel hat der Slot keinen Namen -> Default-Slot
    - Es wird keine weitere Angabe benötigt

```
<div class="row">
  <div class="col-6" v-for="(card, i) of cards" :key="card.id">
    <card :css-class="i % 2 === 0 ? 'bg-primary' : 'bg-secondary'">
      {{ card.content }} ({{ card.id }})
    </card>
  </div>
</div>
```

index.html

https://github.com/sschum/froscon-2023-demo/tree/step-13

Component - Events

```
MyVueApp.component('card', {
  props: ['cssClass'],
  template: `
<div class="card" v-on:click="$emit('card-click')">
  <div :class="cssClass">
    <div class="card-body">
      <h5 class="card-title">
        <slot></slot>
      </h5>
    </div>
  </div>
</div>`
})
```

card.js

- Um Events an die Eltern-Komponente(n) auszulösen, benutzt man die **$emit** Funktion
- Diese erwartet als **ersten** Parameter den Event-**Namen**
- Gefolgt von den Event-Daten (optional)
- Um dann auf dieses Event zu reagieren, benutzt man das zuvor kennengelernte **v-on:** gefolgt von den Event-**Namen**
- In diesem Beispiel ist in der Komponente ein Listener, der auf das Click-Event lauscht
- Dies sorgt dafür, dass via **$emit** ein **neues** Event ausgelöst wird

```
<div class="row">
  <div class="col-6" v-for="(card, i) of cards" :key="card.id">
    <card :css-class="i % 2 === 0 ? 'bg-primary' : 'bg-secondary'"
          v-on:card-click="removeCard(i)">
      {{ card.content }} ({{ card.id }})
    </card>
  </div>
</div>
```

index.html

Single Page Application

Was ist eine SPA?

- Bei einem Seitenwechsel findet kein Neuladen statt
- Stattdessen werden Unterseiten “virtuell” geladen
- Dies ist **nicht** Teil des Vue-Kerns
- Dafür gibt es das Plugin **vue-router**
  - ist von den Machern von Vue
    - damit offiziell unterstützt
  - man möchte nur die Kernkomponente möglichst klein halten

https://github.com/sschum/froscon-2023-demo/tree/step-14

Vue Router

```
<head>
  …
  <script src="./js/vue.js"></script>
  <script src="./js/vue-router.js"></script>
  <script src="./js/page/hello.js"></script>
  <script src="./js/page/age.js"></script>
</head>
…
<div class="card" id="app">
  <ul class="list-group">
    <li class="list-group-item">
       <router-link to="/hello">Hello Vue!</router-link>
    </li>
    <li class="list-group-item">
       <router-link to="/age">Alter?</router-link>
    </li>
  </ul>
  <router-view></router-view>
</div>
…
<script>
  const router = VueRouter.createRouter({
    history: VueRouter.createWebHashHistory(),
    routes: [
      { path: "/hello", component: pages['hello'] },
      { path: "/age",   component: pages['age']   },
    ],
  })
  const MyVueApp = Vue.createApp({})
  MyVueApp.use(router)
</script>
…
```

index.html

```
if(!pages) var pages = {}

pages['hello'] = {
  template: `<div>...</div>`
  data() { return { … } },
  computed: { … },
  methods: { … },
  watch: { … }
}
```

hello.js

```
if(!pages) var pages = {}

pages['age'] = {
  template: `<div>...</div>`
  data() { return { … } },
  computed: { … },
  methods: { … },
  watch: { … }
}
```

age.js

- vue-router Abhängigkeit direkt nach dem vue.js laden
- Der Inhalt der index.html wurde in die hello.js als Komponente ausgelagert

https://github.com/sschum/froscon-2023-demo/tree/step-14

Vue Router

```
<head>
  …
  <script src="./js/vue.js"></script>
  <script src="./js/vue-router.js"></script>
  <script src="./js/page/hello.js"></script>
  <script src="./js/page/age.js"></script>
</head>
…
<div class="card" id="app">
  <ul class="list-group">
    <li class="list-group-item">
       <router-link to="/hello">Hello Vue!</router-link>
    </li>
    <li class="list-group-item">
       <router-link to="/age">Alter?</router-link>
    </li>
  </ul>
  <router-view></router-view>
</div>
…
<script>
  const router = VueRouter.createRouter({
    history: VueRouter.createWebHashHistory(),
    routes: [
      { path: "/hello", component: pages['hello'] },
      { path: "/age",   component: pages['age']   },
    ],
  })
  const MyVueApp = Vue.createApp({})
  MyVueApp.use(router)
</script>
…
```

index.html

```
if(!pages) var pages = {}

pages['hello'] = {
  template: `<div>...</div>`
  data() { return { … } },
  computed: { … },
  methods: { … },
  watch: { … }
}
```

hello.js

```
if(!pages) var pages = {}

pages['age'] = {
  template: `<div>...</div>`
  data() { return { … } },
  computed: { … },
  methods: { … },
  watch: { … }
}
```

age.js

- Es wird eine neue **VueRouter-Instanz** erstellt
- Hier definiert man die **Routen** und deren **Komponenten**
- Dabei habe ich die Components jeweils in eine eigene Variable innerhalb des **pages** Objekts gespeichert
  - dieses **pages** Objekt ist global erreichbar
- Die router-Instanz wird dann an die Vue-Instanz übermittelt

https://github.com/sschum/froscon-2023-demo/tree/step-14

Vue Router

```
<head>
  …
  <script src="./js/vue.js"></script>
  <script src="./js/vue-router.js"></script>
  <script src="./js/page/hello.js"></script>
  <script src="./js/page/age.js"></script>
</head>
…
<div class="card" id="app">
  <ul class="list-group">
    <li class="list-group-item">
       <router-link to="/hello">Hello Vue!</router-link>
    </li>
    <li class="list-group-item">
       <router-link to="/age">Alter?</router-link>
    </li>
  </ul>
  <router-view></router-view>
</div>
…
<script>
  const router = VueRouter.createRouter({
    history: VueRouter.createWebHashHistory(),
    routes: [
      { path: "/hello", component: pages['hello'] },
      { path: "/age",   component: pages['age']   },
    ],
  })
  const MyVueApp = Vue.createApp({})
  MyVueApp.use(router)
</script>
…
```

index.html

```
if(!pages) var pages = {}

pages['hello'] = {
  template: `<div>...</div>`
  data() { return { … } },
  computed: { … },
  methods: { … },
  watch: { … }
}
```

hello.js

```
if(!pages) var pages = {}

pages['age'] = {
  template: `<div>...</div>`
  data() { return { … } },
  computed: { … },
  methods: { … },
  watch: { … }
}
```

age.js

- Für Links in eine Unterseite gibt es die Komponente **router-link**
- Wo diese Unterseiten-Komponente schlussendlich dargestellt werden soll, wird durch die Komponente **router-view** mitgeteilt

https://github.com/sschum/froscon-2023-demo/tree/step-14

Vue Router

```
<head>
  …
  <script src="./js/vue.js"></script>
  <script src="./js/vue-router.js"></script>
  <script src="./js/page/hello.js"></script>
  <script src="./js/page/age.js"></script>
</head>
…
<div class="card" id="app">
  <ul class="list-group">
    <li class="list-group-item">
       <router-link to="/hello">Hello Vue!</router-link>
    </li>
    <li class="list-group-item">
       <router-link to="/age">Alter?</router-link>
    </li>
  </ul>
  <router-view></router-view>
</div>
…
<script>
  const router = VueRouter.createRouter({
    history: VueRouter.createWebHashHistory(),
    routes: [
      { path: "/hello", component: pages['hello'] },
      { path: "/age",   component: pages['age']   },
    ],
  })
  const MyVueApp = Vue.createApp({})
  MyVueApp.use(router)
</script>
…
```

index.html

```
if(!pages) var pages = {}

pages['hello'] = {
  template: `<div>...</div>`
  data() { return { … } },
  computed: { … },
  methods: { … },
  watch: { … }
}
```

hello.js

```
if(!pages) var pages = {}

pages['age'] = {
  template: `<div>...</div>`
  data() { return { … } },
  computed: { … },
  methods: { … },
  watch: { … }
}
```

age.js

- Sobald man auf eine Unterseite wechselt, wird auch die URL angepasst
  - Aber es findet kein Neuladen statt!
- Ein direkter Aufruf dieser Unterseite ist aber dennoch möglich
- Der Router kann anhand der Browser-URL erkennen, welche Unterseite angezeigt werden soll
- Dennoch wird immer die index.html geladen!


DA WÄRE
NOCH EINE SACHE
imgflip.com

Komponenten übergreifende Daten

- Daten können via **Properties** an **Kind-Komponenten** gegeben werden
- Daten können via **Events** an die **Eltern-Komponenten** gegeben werden
- Es wird aber mit der Zeit umständlich diese Daten zwischen den Komponenten zu “synchronisieren”
- Vue bietet mit dem Plugin **Vuex** einen zentralen Store an, auf den alle Komponenten Zugriff haben

https://github.com/sschum/froscon-2023-demo/tree/step-15

Vuex

```
<head>
  …
  <script src="./js/vue.js"></script>
  <script src="./js/vuex.js"></script>
  …
</head>
…
  <p>Store-Counter: {{ storeCount }}</p>
…
<script>
  const store = Vuex.createStore({
    state(){
      return { count: 0 }
    },
    mutations: {
      increment (state) {
        state.count++
      }
    }
  })
  …
  const MyVueApp = Vue.createApp({
    computed: {
      storeCount(){
        return this.$store.state.count
      }
    }
  })
  MyVueApp.use(store)
</script>
```

index.html

```
…
methods: {
  addCard() {
    this.$store.commit(
      'increment'
    )
    …
  },
}
```

hello.js

```
…
watch: {
  days(){
    this.$store.commit(
      'increment'
    )
  }
}
```

age.js

- vuex Abhängigkeit direkt nach dem vue.js laden
- Eine neue Store-Instanz erzeugen
- Und der Vue-Instanz übergeben
- Dieser Store ist nun von überall aus erreichbar

https://github.com/sschum/froscon-2023-demo/tree/step-15

Vuex

```
<head>
  …
  <script src="./js/vue.js"></script>
  <script src="./js/vuex.js"></script>
  …
</head>
…
  <p>Store-Counter: {{ storeCount }}</p>
…
<script>
  const store = Vuex.createStore({
    state() {
      return { count: 0 }
    },
    mutations: {
      increment (state) {
        state.count++
      }
    }
  })
  …
  const MyVueApp = Vue.createApp({
    computed: {
      storeCount(){
        return this.$store.state.count
      }
    }
  })
  MyVueApp.use(store)
</script>
```

index.html

```
…
methods: {
  addCard() {
    this.$store.commit(
      'increment'
    )
    …
  },
}
```

hello.js

```
…
watch: {
  days(){
    this.$store.commit(
      'increment'
    )
  }
}
```

age.js

- Ein Store hat - ähnlich wie eine Komponente - sein eigenen **State**/Daten
- Darin ist der **initiale Zustand** des Stores beschrieben
  - In diesem Fall haben wir ein **count**er

https://github.com/sschum/froscon-2023-demo/tree/step-15

Vuex

```
<head>
  …
  <script src="./js/vue.js"></script>
  <script src="./js/vuex.js"></script>
  …
</head>
…
  <p>Store-Counter: {{ storeCount }}</p>
…
<script>
  const store = Vuex.createStore({
    state() {
      return { count: 0 }
    },
    mutations: {
      increment (state) {
        state.count++
      }
    }
  })
  …
  const MyVueApp = Vue.createApp({
    computed: {
      storeCount(){
        return this.$store.state.count
      }
    }
  })
  MyVueApp.use(store)
</script>
```

index.html

```
…
methods: {
  addCard() {
    this.$store.commit(
      'increment'
    )
    …
  },
}
```

hello.js

```
…
watch: {
  days(){
    this.$store.commit(
      'increment'
    )
  }
}
```

age.js

- Diesen State kann man dann via **Computed Property** für die View bereitstellen
  - Es wäre auch **möglich** in der View direkt auf **$store.state.count** zuzugreifen
- Durch die Computed Property wird die View auch **aktualisiert**, sobald sich der Zustand des Stores (count) **ändert**!

https://github.com/sschum/froscon-2023-demo/tree/step-15

Vuex

```
<head>
  …
  <script src="./js/vue.js"></script>
  <script src="./js/vuex.js"></script>
  …
</head>
…
  <p>Store-Counter: {{ storeCount }}</p>
…
<script>
  const store = Vuex.createStore({
    state() {
      return { count: 0 }
    },
    mutations: {
      increment (state) {
        state.count++
      }
    }
  })
  …
  const MyVueApp = Vue.createApp({
    computed: {
      storeCount(){
        return this.$store.state.count
      }
    }
  })
  MyVueApp.use(store)
</script>
```

index.html

```
…
methods: {
  addCard() {
    this.$store.commit(
      'increment'
    )
    …
  },
}
```

hello.js

```
…
watch: {
  days(){
    this.$store.commit(
      'increment'
    )
  }
}
```

age.js

- Der **Zustand** des Stores **darf nicht** direkt **verändert werden**!
- Dafür werden **Mutationen** genutzt
- Diese haben Zugriff auf den State und können diesen auch Verändern
- So bekommt vuex Änderungen am Zustand mit, und **propagiert** dies an alle “Zuhöhrer”
- Diese Mutationen können dann von Vue-Komponenten via **commit** aufgerufen werden


UND DAS WAR
VUEJS
imgflip.com

Ausblick

- Aktuell muss beim Seitenladen die Komponente(n) initialisiert werden
  - Das (Haupt-)Template muss geparsed werden
  - Die Vue-Komponenten müssen in den Speicher geladen werden
- Das dauert …
  - und während dessen wird dem Nutzer eine (fast) leere Seite angezeigt

Ausblick

- Dadurch, dass nur eine Seite geladen wird (SinglePageApp), wird auch am Anfang die komplette Anwendung geladen
  - Alle benötigten *js-Dateien
  - Indem alle möglichen Komponenten/Unterseiten enthalten sind
  - Auch wenn der Benutzer niemals auf die jeweilige Unterseite
- Auch das dauert …
- IDE Unterstützung ist nur bedingt gegeben
  - Im Falle der Templates “versteht” die IDE nicht, dass es sich um HTML-Elemente handelt

Ausblick

- Diese Probleme können gelöst werden mit der Verwendung von
  - **nodejs**
  - **npm**
  - **webpack**
- Es gibt dadurch
  - Gute IDE Unterstützung
  - Vorkompilierte Komponenten/Templates
  - Minifizierung der JS-Dateien
  - Aufteilung in Chunks (Lade nur das, was aktuell gebraucht wird)
- … aber das ist eine Geschichte für ein anderes Mal ;)
- Ein Udemy-Kurs den ich sehr empfehlen kann
  - **Vue - The Complete Guide - Maximilian Schwarzmüller**
    - ich bekomme kein Geld -> dennoch gut :)

tarent solutions GmbH

Vielen Dank!

Sven Schumann
Softwareentwickler

s.schumann@tarent.de

Am Dickobskreuz 10
53121 Bonn

Telefon: 0228 54881-0
Telefax: 0228 54881-235

info@tarent.de
www.tarent.de