Vue.js Framework

Vue.js - Agenda

- Vue Einführung
- Vue CLI & Quick Start
- Deklaratives Rendering
- Attribute Bindings
- Event Listeners
- Form Bindings
- Conditional Rendering & Listen
- Computed Properties
- Lifecycle und Template Refs
- Watchers
- Components
- Props
- Emits

Was ist Vue.js

Ein JavaScript Framework um Benutzeroberflächen zu erstellen. Es baut auf Standard HTML, CSS und JavaScript auf und bietet ein deklaratives und komponentenbasiertes Programmiermodell, um Benutzeroberflächen effizient zu entwickeln, egal ob sie einfach oder komplex sind.

Warum Vue.js?

- Man arbeitet mit HTML, JavaScript und CSS
- Komponenten Basiert
- Schnelle Lernkurve
- Große Community

Single File Components (SFC)

- Vue Komponenten bestehen aus einer HTML-Ähnlichen Datei (*.vue)
- Template (`<template>`): Der Aufbau der Komponente. Quasi wie HTML.
- Logik (`<script>`): JavaScript Code aufgeführt als ES Modul.
- Styling (`<style>`): Styling im CSS Stil.

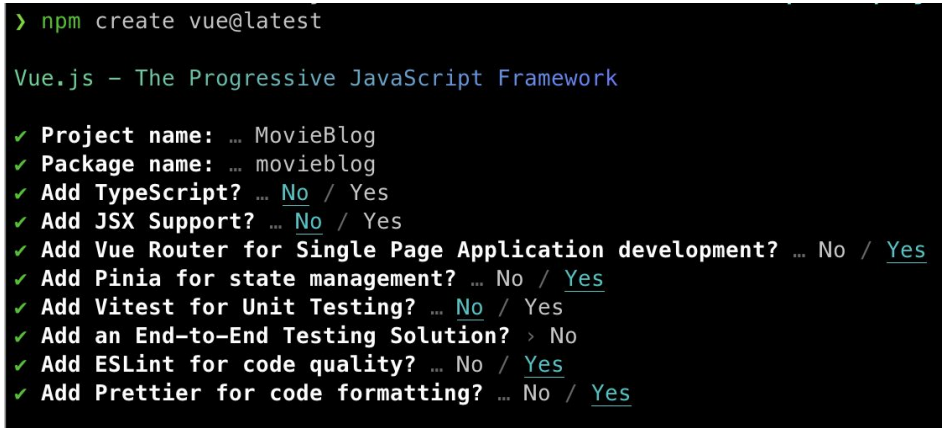
```
<script setup>
import { ref } from 'vue'
const count = ref(0)
</script>

<template>
 <button @click="count++">Count is: {{ count }}</button>
</template>

<style scoped>
button {
 font-weight: bold;
}
</style>
```

Vue CLI & QuickStart

- Aktuelle Node.js Version notwending
- Projekt mit npm aufsetzen: `$ npm create vue@latest`
- Erstellt ein Projekt über das Vue.js Gerüstbau Tool:
- In den Ordner wechseln: `cd MovieBlog`
- JS dependencies installieren: `npm install`
- Lokalen Test Server starten: `npm run dev`



Deklaratives Rendering

- Kernmerkmal von Vue: deklaratives Rendern.
- Vorlagen Syntax erweitert HTML basierend auf JavaScript-Zustand.
- HTML aktualisiert sich automatisch bei Zustandsänderungen.
- Reaktive Zustände lösen Updates bei Änderungen aus.
- reactive()-API von Vue für reaktiven Zustand.
- reactive() funktioniert mit Objekten, Arrays, Map, Set.

```
import { reactive } from 'vue'

const counter = reactive({
 count: 0
})

console.log(counter.count) // 0
counter.count++
```

Deklaratives Rendering

- ref() erstellt Objekte für jeden Werttyp, Zugriff über .value-Eigenschaft.
- Reaktiver Zustand im <script setup>-Block der Komponente für templates nutzbar
- Im Template: Dynamischer Text basierend auf Counter-Objekt und Message-Referenz
- Verwendung der Mustache-Syntax in Template:

```
<h1>{{ message }}</h1>
<p>count is: {{ counter.count }}</p>
```

```
import { ref } from 'vue'

const message = ref('Hello World!')

console.log(message.value) // "Hello World!"
message.value = 'Changed'
```

Attribute Bindings

- In Vue werden Mustaches nur für Textinterpolation verwendet.
- Zum Binden eines Attributs an einen dynamischen Wert wird die v-bind-Direktive verwendet:

```
<div v-bind:id="dynamicId"></div>
```

- Direktiven sind spezielle Attribute mit v-Präfix, Teil von Vues Vorlagen Syntax.
- Direktiven Werte sind JavaScript-Ausdrücke mit Zugriff auf den Komponenten Zustand.
- Der Teil nach dem Doppelpunkt (:id) ist das "Argument" der Direktive; synchronisiert das id-Attribut des Elements mit der dynamicId-Eigenschaft des Zustands.
- v-bind hat eine eigene Kurzschreibweise aufgrund häufiger Nutzung:

```
<div :id="dynamicId"></div>
```

Event Listeners

- Lauschen auf DOM-Events mittels v-on-Direktive
- Kurzschreibweise für v-on aufgrund häufiger Nutzung:
  - `<button @click="increment">{{ count }}</button>`
- *increment* bezieht sich auf eine Funktion im *<script setup>*:

- Innerhalb der Funktion Aktualisierung des Komponentenzustands durch Mutation von refs.

```
<script setup>
import { ref } from 'vue'
const count = ref(0)

function increment() {
  // update component state
  count.value++
}
</script>
```

Form Bindings

- Kombination von *'**v-bind**'* und *'**v-on**'* für Zwei-Wege-Bindungen in Formularelementen:
  - `<input :value="text" @input="onInput">`
- Funktion *onInput* empfängt natives DOM-Event als Argument, aktualisiert *text.value*:

```
function onInput(e) {
  text.value = e.target.value
}
```

- Tippen im Eingabefeld führt zu Aktualisierung des Textes im <p>-Element.
- Vereinfachung durch Vue-Direktive '***v-model***', syntaktischer Zucker für oben Genanntes:
  - `<input v-model="text">`
- v-model synchronisiert automatisch den Wert des <input> mit dem gebundenen Zustand.
- v-model funktioniert nicht nur bei Texteingaben, sondern auch bei Checkboxen, Radio-Buttons und Auswahllisten.

Conditional Rendering

- Einsatz der v-if-Direktive für bedingte Elementdarstellung:
  - `<h1 v-if="awesome">Vue is awesome!</h1>`
- **<h1>** wird nur gerendert, wenn der Wert von awesome wahr ist. Bei einem falschen Wert wird es aus dem DOM entfernt.
- Verwendung von **v-else** und **v-else-if** für weitere Bedingung Zweige
  - `<h1 v-if="awesome">Vue is awesome!</h1>`
  - `<h1 v-else>Oh no 😢</h1>`

Listen

```
<ul>
  <li v-for="todo in todos" :key="todo.id">
    {{ todo.text }}
  </li>
</ul>
```

- Verwendung der v-for-Direktive zum Rendern einer Liste basierend auf einem Array:

- ***todo*** ist eine lokale Variable, die das aktuell iterierte Array-Element repräsentiert. Zugänglich nur im oder am **v-for**-Element, ähnlich einem Funktionsumfang.
- Jedes Todo-Objekt erhält eine eindeutige ID, gebunden als spezielles Schlüsselattribut für jedes **<li>**. Der Schlüssel ermöglicht es Vue, jedes **<li>** genau der Position seines entsprechenden Objekts im Array zuzuordnen.
- Zwei Wege zur Aktualisierung der Liste: `todos.value.push(newTodo)`
- Aufruf von mutierenden Methoden auf dem Quellarray: `todos.value = todos.value.filter(/* ... */)`

Computed Properties

- Weiterentwicklung der Todo-Liste mit einer Umschaltfunktion für jedes Todo:
  - Hinzufügen einer done-Eigenschaft zu jedem Todo-Objekt
  - Bindung an ein Kontrollkästchen mit v-model:

```
<li v-for="todo in todos">
  <input type="checkbox" v-model="todo.done">
```

- Möglichkeit, bereits erledigte Todos auszublenden:
  - Button schaltet hideCompleted-Zustand um. `const hideCompleted = ref(false)`
  - Unterschiedliche Listenelemente basierend auf diesem Zustand rendern.
- Einführung von computed():
  - Erstellung eines berechneten Refs, der seinen .value auf Basis anderer reaktiver Datenquellen berechnet:

```
const filteredTodos = computed(() => {
  // Rückgabe gefilterter Todos basierend auf
  // `todos.value` & `hideCompleted.value`
})
```

Lifecycle und Template Refs

- Verwendung der onMounted()-Funktion, um Code nach dem Einbinden auszuführen:

```
import { onMounted } from 'vue'

onMounted(() => {
  // Komponente ist jetzt eingebunden.
})
```

- onMounted ist ein Lebenszyklus-Hook: ermöglicht Registrierung eines Callbacks, der zu bestimmten Zeiten des Komponenten-Lebenszyklus aufgerufen wird.
- Weitere Hooks wie **onUpdated** und **onUnmounted** verfügbar.
- Für mehr Details siehe das Lebenszyklus-Diagramm von Vue.

Watchers

- Manchmal ist es nötig, "Nebeneffekte" reaktiv auszuführen - zum Beispiel, eine Zahl zu protokollieren, wenn sie sich ändert.
- Dies kann mit Beobachtern (Watchern) erreicht werden:

- watch() kann direkt eine Ref überwachen, und der Callback wird ausgelöst, wann immer sich der Wert von count ändert.

```
import { ref, watch } from 'vue'

const count = ref(0)

watch(count, (newCount) => {
  // ja, console.log() ist ein Nebeneffekt
  console.log(`neuer Zählerstand: ${newCount}`)
})
```

Components

- Auslagern von Funktionalitäten in Child-Komponenten
- Vereinfacht die Lesbarkeit und Struktur von Programmen
- Ermöglicht das wiederverwenden von Komponenten

```
<script setup>
import ChildComp from './ChildComp.vue'
</script>

<template>
  <ChildComp />
</template>
```

Props

- Ein Kind Komponente kann Eingaben vom Elternteil über **Props** akzeptieren.
- Zuerst muss sie die akzeptierten Props deklarieren:
- **defineProps()** ist ein Makro zur Kompilierungszeit und muss nicht importiert werden.
- Einmal deklariert, kann das **msg** Prop im Template der Kindkomponente verwendet werden. Es ist auch im JavaScript über das zurückgegebene Objekt von **defineProps()** zugänglich.
- Der Elternteil kann das Prop an das Kind wie Attribute übergeben. Um einen dynamischen Wert zu übergeben, kann auch die v-bind-Syntax verwendet werden:

```
<!-- ChildComp.vue -->
<script setup>
const props = defineProps({
  msg: String
})
</script>
```

```
<template>
    <ChildComp :msg="greeting" />
</template>
```

Emits

```
<script setup>
// Deklaration der ausgesendeten Ereignisse
const emit = defineEmits(['response'])
// Ereignis mit Argument senden
emit('response', 'hello from child')
</script>
```

- Neben dem Empfang von Props kann eine Kind Komponente auch Ereignisse an die Eltern Komponente zurück senden
- Das erste Argument von emit() ist der Ereignisname. Zusätzliche Argumente werden an den Event-Listener weitergegeben.
- Die Eltern Komponente kann auf von der Kind Komponente gesendete Ereignisse mit ***v-on*** hören - hier erhält der Handler das zusätzliche Argument aus dem Emit-Aufruf des Kindes und weist es einem lokalen Zustand zu:

```
<template>
  <ChildComp @response="(msg) => childMsg = msg" />
</template>
```