Frontend Abstractions Live: Vue3 Pokemon App

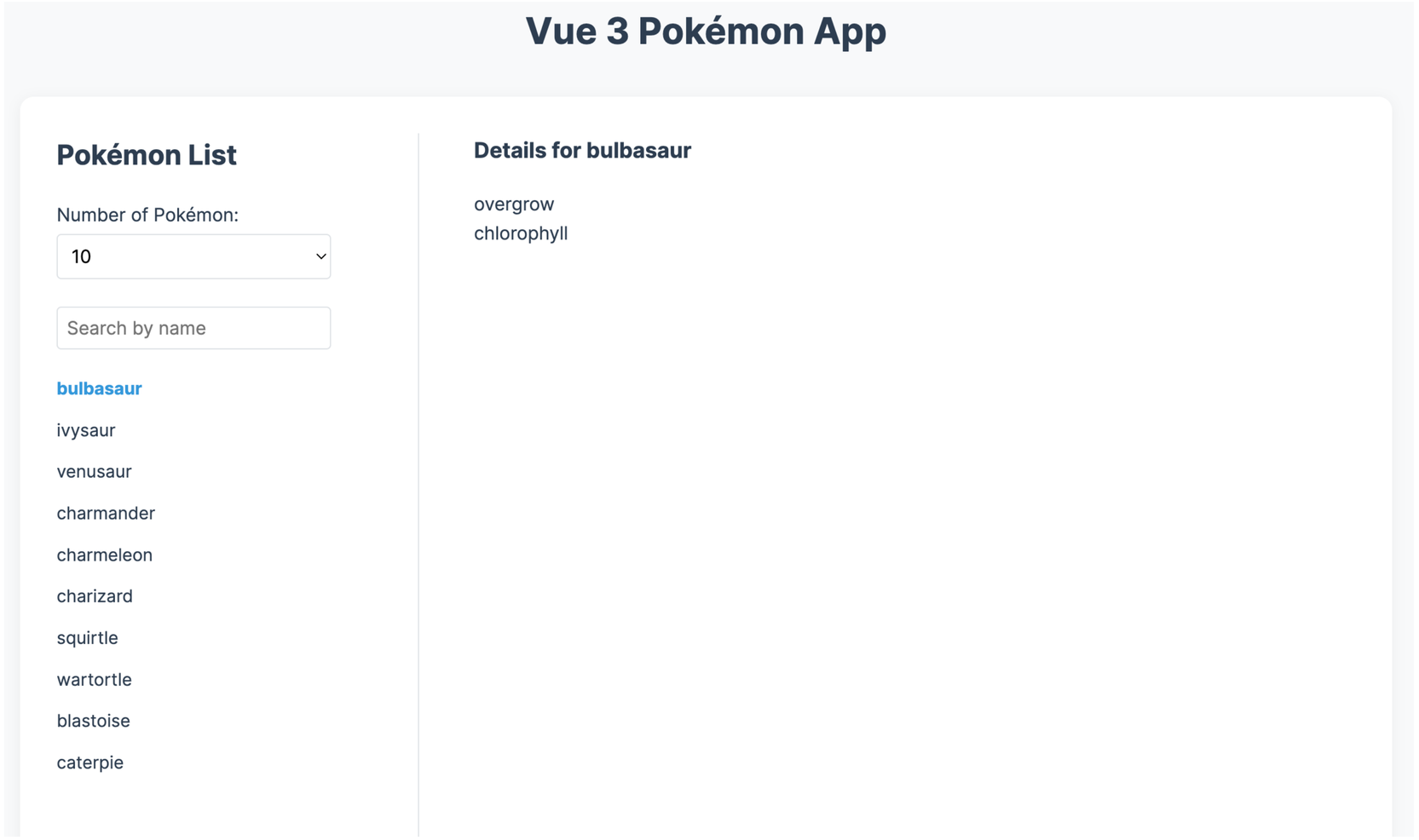
Web Engineering

INTERACTIVE PROGRAMMING & ANALYSIS LAB (IPA LAB @ TU WIEN)

Jürgen Cito

TECHNISCHE UNIVERSITÄT WIEN

Pokemon Vue3 App



Pokemon Vue3 App

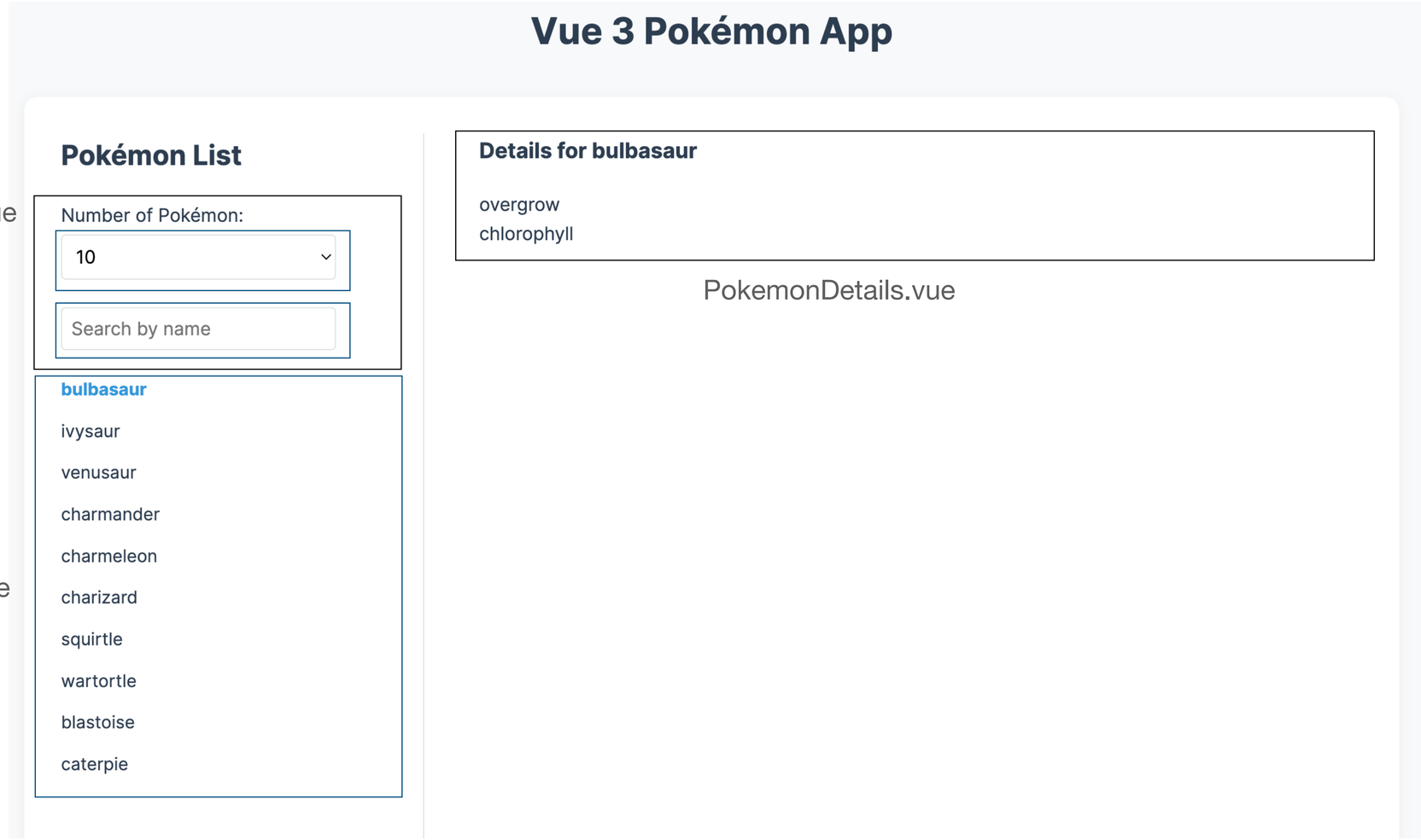
Component Design

PokemonControls.vue

LimitSelector.vue

SearchInput.vue

PokemonListDisplay.vue



PokemonDetails.vue

Open Questions:

- Granularity
- When to fetch data
- Managing state
- Binding
- Conditional rendering

Component Design Evolution

How it started

```
PokemonList.vue
├── <input type="number">       # Inline limit selector
├── <input type="text">         # Inline search input
├── <ul> with <li>              # Inline list rendering
├── <PokemonDetails>            # Detail view (separate)
```

Final Component Architecture

```
PokemonList.vue                ← container (data + layout)
├── PokemonControls.vue        ← v-model-driven control surface
│   ├── LimitSelector.vue      ← generic, reusable preset selector
│   └── SearchInput.vue        ← reusable text search
├── PokemonListDisplay.vue     ← list rendering + selection feedback
└── PokemonDetails.vue         ← display abilities of selected Pokémon
```

Observations:

- Start off simple with inline elements within components
- Only start abstracting once you “feel the pain (“Premature optimization is the root of all evil”)
- Exception to this rule: If you already come from experience and know which abstractions already make sense

Container: PokemonList.vue

View

```vue
<template>
  <div class="container">
    <!-- Left side: Controls + List -->
    <div class="list">
      <h2>Pokémon List</h2>

      <PokemonControls
        v-model:limit="pokemonLimit"
        v-model:search="search"
      />

      <PokemonListDisplay
        :pokemon="filteredList"
        :selectedName="selectedPokemonDetails?.name"
        @select="loadPokemonDetails"
      />
    </div>

    <!-- Right side: Details -->
    <div class="details">
      <PokemonDetails
        v-if="selectedPokemonDetails"
        :pokemon="selectedPokemonDetails"
        :key="selectedPokemonDetails.name"
      />
      <p v-else class="placeholder">Select a Pokémon to see abilities.</p>
    </div>

    <p v-if="loading" class="loading-indicator">Loading...</p>
  </div>
</template>
```

Binding

Events

ViewModel

```vue
<script setup>
import { ref, computed, onMounted, watch } from 'vue'
import { usePokemonStore } from '../stores/pokemonStore'
import PokemonControls from './PokemonControls.vue'
import PokemonListDisplay from './PokemonListDisplay.vue'
import PokemonDetails from './PokemonDetails.vue'

const store = usePokemonStore()

const pokemonLimit = ref(10)
const search = ref('')
const selectedPokemonDetails = ref(null)
const loading = ref(false)

onMounted(() => {
  store.fetchPokemonList(pokemonLimit.value)
})

watch(pokemonLimit, async (newLimit) => {
  selectedPokemonDetails.value = null
  loading.value = true
  await store.fetchPokemonList(newLimit)
  loading.value = false
})

const filteredList = computed(() =>
  store.pokemonList.filter(p =>
    p.name.toLowerCase().includes(search.value.toLowerCase())
  )
)

async function loadPokemonDetails(pokemon) {
  loading.value = true
  try {
    const res = await fetch(pokemon.url)
    const data = await res.json()
    selectedPokemonDetails.value = {
      name: pokemon.name,
      abilities: data.abilities.map(a => a.ability.name),
    }
  } finally {
    loading.value = false
  }
}
</script>
```

Lifecycle Methods

Computed Values Are Reactive

Observations:

- Keep components as stateless as possible (i.e., parameterize)
- Minimize global state
- High-level components serve as data orchestrators
- When to fetch data is a design decision

Control Surface — PokemonControls.vue

```vue
<template>
  <div class="controls">
    <LimitSelector
      v-model="limit"
      :options="[10, 20, 50, 100, 500]"
      label="Number of Pokémon:"
    />
    <SearchInput v-model="search" placeholder="Search by name" />
  </div>
</template>

<script setup>
import LimitSelector from './LimitSelector.vue'
import SearchInput from './SearchInput.vue'

const limit = defineModel('limit')
const search = defineModel('search')
</script>
```

Observations/questions:

- Choice of abstraction level is a design decision
- Generality vs. Specificity in components
- Where do we put configuration values (e.g., selection values 10, 20, etc.)?

Simple stateless component — PokemonDetails.vue

```vue
<template>
  <div v-if="pokemon">
    <h3>Details for {{ pokemon.name }}</h3>
    <ul>
      <li v-for="ability in pokemon.abilities" :key="ability">
        {{ ability }}
      </li>
    </ul>
  </div>
</template>

<script setup>
defineProps({
  pokemon: {
    type: Object,
    required: true,
  },
})
</script>
```

Observations/questions:

- Keep low-level components as “stupid” as possible
- This makes them more easily comprehensible and testable

Simple component — PokemonListDisplay.vue

```vue
<template>
  <ul>
    <li
      v-for="pokemon in pokemon"
      :key="pokemon.name"
      @click="$emit('select', pokemon)"
      :class="['pokemon-name', { selected: pokemon.name === selectedName }]"
    >
      {{ pokemon.name }}
    </li>
  </ul>
</template>

<script setup>
defineProps({
  pokemon: {
    type: Array,
    required: true
  },
  selectedName: {
    type: String,
    default: null
  }
})

defineEmits(['select'])
</script>
```

Observations/questions:

- Sometimes, we need to propagate information from low-level components to other parts of the application
- Design decision boils down to whether we use direct propagation (*emit*) or global state (e.g., *Pinia store*)
- Try to avoid global state if you can, because it can become messy to reason about all the places state change affects the application

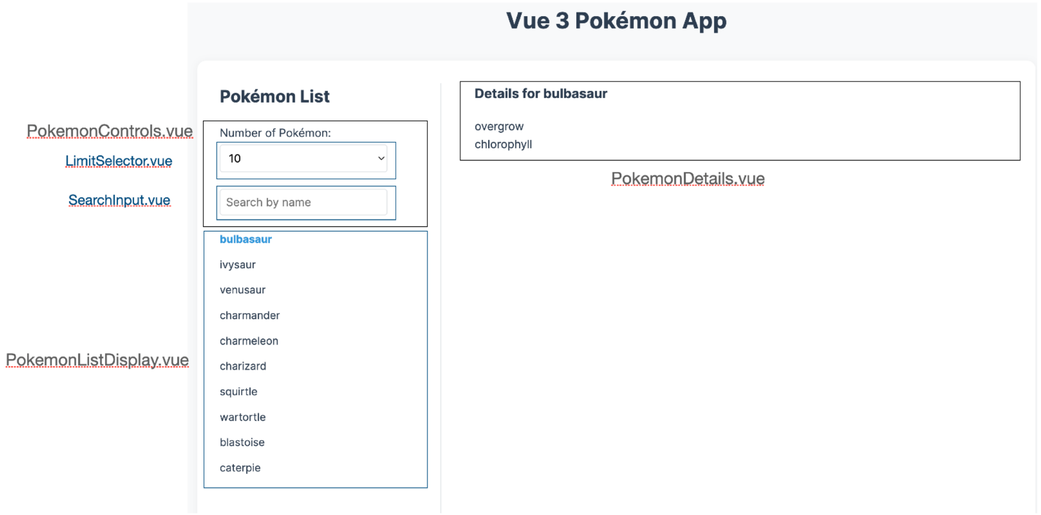
Global State in Pinia store — stores/pokemonStore.js

```
import { defineStore } from 'pinia'

export const usePokemonStore = defineStore('pokemon', {
  state: () => ({
    pokemonList: [], // { name, url }
    selectedPokemonDetails: null,
    loading: false,
  }),

  actions: {
    async fetchPokemonList(limit = 151) {
      this.loading = true
      try {
        const res = await fetch(`https://pokeapi.co/api/v2/pokemon?limit=${limit}`)
        const data = await res.json()
        this.pokemonList = data.results // includes name + url
      } finally {
        this.loading = false
      }
    }
  }
})
```

Observations/questions:

- Use for data that you need across the application
- Could have also been used to store the selected Pokemon if the information is needed across multiple components across the application

Pokemon Vue3 App



Summary:

- Avoid premature optimization when designing component architectures: Start simple and extract components when you “feel the pain"
- Avoid global state if you can (“emit" information/ events from lower-level to higher level components)
- However, if orchestration becomes too tedious because information is needed across components, consider global state (e.g., Pinia store)