Veebirakenduse projekt (ITI0302)

Loeng 5

Frontend: Vue · Siim Rebane

The problem

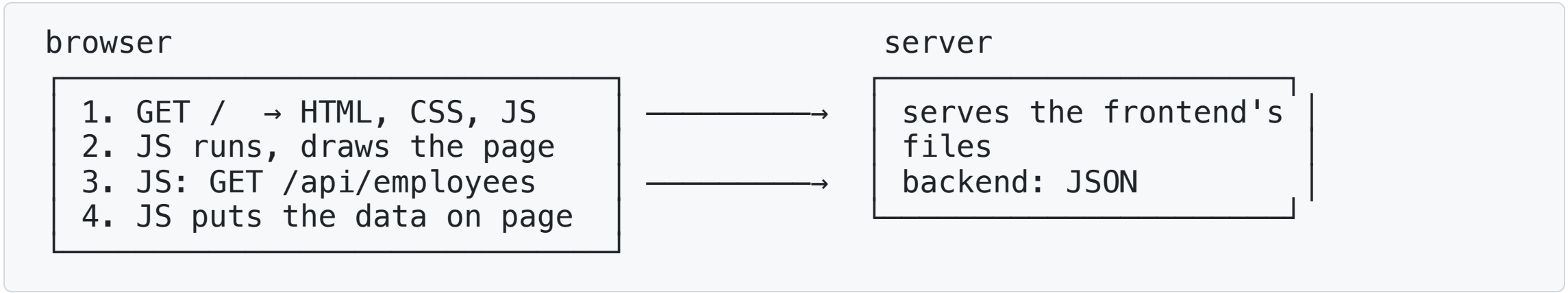
The backend from lecture 4 answers requests with JSON:

```
$ curl -s localhost:8080/api/employees
[{"id":1,"name":"John Smith","email":"john@example.com","departmentName":"Development"},
 {"id":2,"name":"Jane Doe","email":"jane@example.com","departmentName":"Sales"}]
```

A person using the application does not type `curl`. They open an address in a browser and expect a page: a list, a form, a button that deletes.

That page is the **frontend**: a second program, written for the browser, that calls the backend's endpoints and shows the results. Milestone 1 asks for at least one working view per team member that uses a backend endpoint.

What runs where



- The frontend's code runs **in the browser**, on the user's computer, not on the server
- The server only hands out its files, then answers the `/api` requests
- Three languages share the work: **HTML** for the content, **CSS** for the look, **JavaScript** for the behaviour

HTML

HTML (HyperText Markup Language) describes the content and structure of a page. It is not a programming language: it marks up text.

```
<!DOCTYPE html>
<html>
<head>
    <title>Staff</title>
</head>
<body>
    <h1>Employees</h1>
    <p class="intro">Everyone in the company.</p>
    <a href="/employees/1">John Smith</a>
</body>
</html>
```

Tags, elements, attributes

```
<h1>Employees</h1>
<p class="intro">Everyone in the company.</p>
<a href="/employees/1">John Smith</a>
```

Term	Here
tag	`<h1>` opens, `</h1>` closes
element	a tag pair with its content: `<h1>Employees</h1>`
attribute	a setting on an element: `class="intro"`, `href="/employees/1"`
`<head>`, `<body>`	information about the page; the content shown

Common elements

Element	Is
`<div>`, `<span>`	a generic block; a generic piece of text
`<h1>` … `<h6>`, `<p>`	headings; a paragraph
`<a href="…">`	a link
`<img src="…" alt="…">`	an image
`<ul>`, `<ol>`, `<li>`	a list without numbers, with numbers; a list item
`<table>`, `<tr>`, `<th>`, `<td>`	a table, a row, a header cell, a cell
`<form>`, `<input>`, `<select>`, `<option>`, `<button>`	a form and its fields
`<nav>`, `<header>`, `<main>`	the parts of a page, by role

The DOM

The browser reads the HTML and builds a tree of objects from it: the **DOM** (Document Object Model). Each element is one node, with its attributes and children.

```
document
└─ html
   ├─ head
   │  └─ title: "Staff"
   └─ body
      ├─ h1: "Employees"
      ├─ p.intro: "Everyone in the company."
      └─ a[href=/employees/1]: "John Smith"
```

What the user sees is the DOM, not the HTML file. JavaScript changes the page by changing the DOM: adding, removing and editing nodes.

CSS

CSS (Cascading Style Sheets) sets how elements look and where they are placed. A CSS **rule** is a selector and a list of declarations:

```
.intro {
    color: #555;
    font-size: 18px;
}
```

Part	Here
selector	`.intro` : which elements the rule applies to
declaration	`color: #555;` : a property and its value

A rule can be written in a `style="…"` attribute on one element, in a `<style>` element in the page, or in a separate `.css` file. When several rules set the same property on an element, the more specific selector wins: that is the "cascading".

CSS selectors

```css
/* class: every element with class="container" */
.container { }
.btn-primary { }

/* id: the one element with id="header" */
#header { }

/* element: every <h1> */
h1 { }

/* descendant: every <td> inside a <table> */
table td { }
```

```html
<div class="container">
<button class="btn-primary">

<div id="header">

<h1>Styled by the h1 rule</h1>

<table><tr><td>…</td></tr></table>
```

- a **class** can be on many elements, and an element can have several: `class="btn btn-primary"`
- an **id** is unique in the page

Common CSS properties

Layout and spacing:

```
display: block;   /* on its own line  */
display: inline;  /* inside the text  */
display: flex;    /* lays out its children in a row or a column */
padding: 20px;    /* space inside the border  */
margin: 10px;     /* space outside the border */
width: 100%;
```

Colours and text:

```
color: #333;
background-color: #f0f0f0;
font-size: 16px;
font-weight: bold;
text-align: center;
border: 1px solid #ccc;
```

```
nav {
    display: flex;            /* the links side by side */
    gap: 16px;                /* space between them */
    align-items: center;      /* centred across the row */
}
```

JavaScript

JavaScript is the programming language of the browser: every browser has a JavaScript engine that runs the scripts a page loads. The scripts read and change the DOM, react to clicks and key presses, and send requests.

Node.js is the same engine outside a browser, as a program on your computer or a server. The tools that build a Vue project run on Node.js; the finished frontend runs in the browser.

```
document.querySelector('#save').addEventListener('click', () => {
    document.querySelector('.result').textContent = 'Saved'
})
```

`querySelector` finds an element by a CSS selector; `addEventListener('click', …)` runs a function on every click.

Values and variables



`const`	the variable cannot be given a new value; the object in it can still change
`let`	the variable can be given a new value
`employee.name`	a property of an object
`ids.length`, `ids[0]`	the length; the first item

```
`Hello, ${employee.name}!`
```

a **template literal**: back quotes, `${…}` puts a value into the text

Functions

```
function greet(name) {
    return 'Hello ' + name
}

const add = (a, b) => a + b          // an arrow function

const names = employees.map((e) => e.name)
const inSales = employees.filter((e) => e.departmentName === 'Sales')
```

- an **arrow function** `(a, b) => …` is a shorter way to write a function; with one expression after `=>`, that expression is returned
- `map` makes a new array from each item; `filter` keeps the items for which the function returns `true`
- `===` compares without converting types: `1 === '1'` is `false`

Waiting for the network

A request to the backend takes time, and the browser does not stop while it waits. A function that starts such work returns a **Promise**: an object that later holds the result, or an error.

```
async function loadEmployees() {
    const response = await fetch('/api/employees')
    const employees = await response.json()
    return employees
}
```

- `async` marks a function that uses `await`; it returns a Promise itself
- `await` waits for a Promise and gives its result; the rest of the page keeps working meanwhile
- `fetch(url)` sends an HTTP request and gives a Promise of the response

An error in an awaited Promise is thrown at the `await`, and caught with `try { … } catch (err) { … }`.

Modules

A **module** is one JavaScript file with its own names. What it offers to other files it **exports**; other files **import** it:

```
// src/services/api.js
export default api              // the module's one main export
export function helper() {}     // a named export
```

```
// another file
import api from '@/services/api'           // the default export
import { helper } from '@/services/api'    // a named export
import { ref, computed } from 'vue'         // named exports of an installed package
```

A path starting with `@/` means the project's `src/` folder, as set in the project's configuration; a name without a path, such as `'vue'`, is a package installed in the project.

Changing the page by hand

Showing a list of employees with plain JavaScript means building the DOM yourself:

```
const employees = await (await fetch('/api/employees')).json()
const table = document.querySelector('#employees')
for (const e of employees) {
    const row = document.createElement('tr')
    row.innerHTML = `<td>${e.name}</td><td>${e.email}</td>`
    table.appendChild(row)
}
```

After every change of the data, a delete, a new employee, a search, the code has to find the right nodes again and change them. The page and the data drift apart as soon as one place is forgotten.

Vue

Vue is a JavaScript framework for building user interfaces from **components**. A component describes, in a **template**, what the page looks like for a given state, and holds that state in **reactive** variables.

When a reactive variable changes, Vue works out which parts of the page depend on it and updates those DOM nodes. The code changes data; Vue changes the page.

```
<tr v-for="e in employees" :key="e.id">
  <td>{{ e.name }}</td>
  <td>{{ e.email }}</td>
</tr>
```

`v-for` repeats the row once per employee; `{{ e.name }}` shows a value. Removing an employee from the `employees` array removes its row. React and Angular work on the same principle, with different syntax; Milestone 1 allows any of the three.

The tools

Tool	Is	Does
Node.js	JavaScript outside the browser	runs the tools below; a `create-vue` project needs 22 from 22.18, or 24 from 24.12
npm	Node's package manager	downloads packages (libraries) and runs the project's scripts
`package.json`	the project's description	its name, scripts, and the packages it needs, with version ranges
`node_modules/`	the downloaded packages	created by `npm install`; never in git
`package-lock.json`	the exact version of every package installed	in git, so every machine installs the same
Vite	a build tool	a development server that reloads on every save; a production build
create-vue	the official project generator	creates a new Vue project that uses Vite

npm is to a Vue project what Maven or Gradle is to the Spring project.

Creating a project

```
$ npm create vue@latest example-vue -- --router

> npx
> create-vue example-vue --router

┌  Vue.js - The Progressive JavaScript Framework

Scaffolding project in …/example-vue...
│
└  Done. Now run:

   cd example-vue
   npm install
   npm run dev
```

Part	Means
`npm create vue@latest`	download the latest `create-vue` and run it
`example-vue`	the folder and project name
`--`	the options after it go to `create-vue`, not to npm
`--router`	include Vue Router, for several pages; without options, `create-vue` asks about each feature

The optional features

`create-vue` offers, as questions or as options:

Option	Adds	For Milestone 1
`--router`	Vue Router: several pages, each with its own address	yes
`--ts`	TypeScript: JavaScript with types	optional
`--pinia`	Pinia: state shared by many components	optional
`--vitest`, `--playwright`, `--cypress`	test tools	optional
`--eslint`, `--prettier`	checks and formatting of the code	optional

The examples on these slides use the router only, plain JavaScript. Everything else can be added to a project later with `npm install`.

Installing and starting

```
$ cd example-vue
$ npm install

added 144 packages, and audited 145 packages in 13s

found 0 vulnerabilities

$ npm run dev

  VITE v8.3.1  ready in 863 ms

  ➜  Local:   http://localhost:5173/
```

- `npm install` reads `package.json` and downloads what it lists into `node_modules/`: here 144 packages, 74 MB
- `npm run dev` runs the script named `dev` in `package.json`: Vite's development server on port 5173
- the development server rebuilds what changed on every save and updates the open page without a reload; it runs until Ctrl-C

The project's files

```
example-vue/
├── index.html
├── package.json
├── package-lock.json
├── vite.config.js
├── public/
│   └── favicon.ico
└── src/
    ├── main.js
    ├── App.vue
    ├── assets/
    ├── components/
    ├── router/
    │   └── index.js
    └── views/
```

Path	Holds
`index.html`	the one HTML page; everything else is added by JavaScript
`vite.config.js`	Vite's settings
`public/`	files served as they are
`src/main.js`	the code that starts the application
`src/App.vue`	the root component
`src/views/`	components that are whole pages
`src/components/`	components used inside pages
`src/router/index.js`	which address shows which page

The generated example components (`HelloWorld` , `TheWelcome` , `WelcomeItem` , the icons) can be deleted, together with their use in `HomeView.vue` and `App.vue` .

`package.json`

```
{
  "name": "example-vue",
  "version": "0.0.0",
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "vue": "^3.5.42",
    "vue-router": "^5.3.1"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^6.0.8",
    "vite": "^8.2.2",
    "vite-plugin-vue-devtools": "^8.2.1"
  },
  "engines": {
    "node": "^22.18.0 || >=24.12.0"
  }
}
```

- `scripts`: what `npm run <name>` runs
- `dependencies`: packages the application needs when it runs; `devDependencies`: only for building it
- `^3.5.42`: any `3.x.y` from 3.5.42 up, never 4; `package-lock.json` records the exact one installed
- `"type": "module"`: the project's `.js` files are modules, with `import` and `export`
- `engines`: the Node.js versions the project needs
- `"private": true`: npm refuses to publish it as a package

How the page starts

`index.html`:

```
<body>
  <div id="app"></div>
  <script type="module" src="/src/main.js"></script>
</body>
```

`src/main.js`:

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'

const app = createApp(App)
app.use(router)
app.mount('#app')
```

1. the browser loads `index.html`: an empty `<div id="app">`
2. it runs `main.js`
3. `createApp(App)` creates the application with `App.vue` as its root component
4. `app.use(router)` adds the router
5. `app.mount('#app')` draws `App` into the empty div

Everything the user sees from then on is drawn by components.

A component: one `.vue` file

A **single-file component** keeps a component's three parts in one file:

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

<template>
  <h1>{{ title }}</h1>
</template>

<style scoped>
h1 { color: #2b2557; }
</style>
```

Part	Holds
`<script setup>`	the component's state and functions; everything declared here is usable in the template
`<template>`	the HTML the component draws, with Vue's additions
`<style scoped>`	CSS; `scoped` limits it to this component's elements

Showing values: `{{ }}`

Double braces put a value into the page as text:

```
<h1>{{ employee.name }}</h1>
<p>{{ employee.email }} · {{ employee.departmentName }}</p>
<p>{{ employees.length }} employees</p>
<p>{{ employees.length === 0 ? 'Nobody yet' : '' }}</p>
```

- inside the braces is one JavaScript **expression**: a variable, a property, a calculation, a condition with `? :`
- the value is inserted as text: `<b>` in a name appears as the characters `<b>`, not as bold text

Reactive state: `ref`

`ref(value)` creates a reactive variable. Vue tracks where it is used and redraws those places when it changes.

```
import { ref } from 'vue'

const employees = ref([])        // an empty array to start with
const loading = ref(true)

employees.value = response.data // in <script>: through .value
loading.value = false
```

```
<p>{{ employees.length }} employees</p>      <!-- in the template: without .value -->
```

A plain variable, `let loading = true`, is not reactive: changing it changes nothing on the page. `reactive({...})` makes a whole object reactive, without `.value`; a `ref` holds any value, which is why the examples use `ref` throughout.

Setting attributes: `:`

`:attribute="expression"`, short for `v-bind:attribute`, sets an attribute from a value instead of fixed text:

```
<a :href="`/employees/${e.id}`">{{ e.name }}</a>
<img :src="photoUrl" :alt="employee.name" />
<option :value="d.id">{{ d.name }}</option>
<button :disabled="saving">Save</button>
<tr :class="{ inactive: !e.active }">
```

- without `:` the attribute is the text as written: `href="/employees"`
- with `:` it is the expression's value: `:href="`/employees/${e.id}`"` is `/employees/1` for the first employee
- `:class="{ inactive: … }"`: the class is there while the condition is true

Handling events: `@`

`@event="…"`, short for `v-on:event`, runs code when the event happens:

```
<button @click="remove(e.id)">Delete</button>
<form @submit.prevent="save"> … </form>
<input @keyup.enter="search()" />
```

Part	Means
`@click="remove(e.id)"`	on a click, call `remove` with this row's id
`@submit.prevent`	on submit, and **prevent** the browser's default: reloading the page with the form's data
`@keyup.enter`	only when the key is Enter

Showing parts conditionally

```
<p v-if="loading">Loading…</p>
<p v-else-if="error" class="error">Could not load the employees: {{ error }}</p>
<table v-else> … </table>
```

- `v-if` : the element exists only while the condition is true; `v-else-if` , `v-else` directly after it
- exactly one of the three is on the page at a time

```
<p v-show="expanded">Details</p>
```

- `v-show` : the element always exists and is hidden with CSS; switching it is cheaper than `v-if` when it changes often

`<template v-if="…">` groups several elements under one condition without adding an element to the page.

Lists: `v-for`

```
<tr v-for="e in employees" :key="e.id">
  <td>{{ e.name }}</td>
  <td>{{ e.email }}</td>
  <td><button @click="remove(e.id)">Delete</button></td>
</tr>
```

- one `<tr>` per item of `employees`; inside, `e` is the current item
- `:key` gives each row an identity. When the list changes, Vue keeps, moves or removes the rows by key, instead of rewriting them by position. The key is unique and stable: the database id
- the position in the list, `v-for="(e, i) in employees" :key="i"`, is not stable: after a delete, every following row gets another row's key

Form fields: `v-model`

`v-model` connects a form field and a reactive variable both ways: typing changes the variable, changing the variable changes the field.

```
<input v-model="name" placeholder="Name" required />
<input v-model="email" type="email" placeholder="E-mail" required />
<select v-model="departmentId" required>
  <option v-for="d in departments" :key="d.id" :value="d.id">{{ d.name }}</option>
</select>
<input v-model.number="salary" type="number" />
```

- a `<select>` gives the chosen option's `:value` : here the department's id, a number
- a text field always gives a string; `.number` converts it to a number
- `required` , `type="email"` : the browser's own checks, before `@submit` runs

Derived state: `computed`

A **computed** value is calculated from other reactive values, and recalculated when they change:

```
const search = ref('')

const shown = computed(() =>
  employees.value.filter((e) => e.name.toLowerCase().includes(search.value.toLowerCase())),
)
```

```
<input v-model="search" placeholder="Search by name" />
<p>{{ shown.length }} of {{ employees.length }}</p>
<tr v-for="e in shown" :key="e.id"> … </tr>
```

Typing in the search field changes `search`; `shown` is calculated again; the table shows the matching rows. Nothing else in the code refers to the table.

Calling the backend: Axios

`fetch` is built into the browser. **Axios** is a package that wraps it with a few conveniences: JSON is converted both ways, and an answer of 4xx or 5xx throws an error instead of counting as success.

```
$ npm install axios

added 26 packages, and audited 171 packages in 2s
```

```
// src/services/api.js
import axios from 'axios'

const api = axios.create({
  baseURL: '/api',
})

export default api
```

`api.get('/employees')` requests `/api/employees`; the parsed JSON is in `response.data`. `api.post(url, object)`, `api.put`, `api.delete` work the same way.

When a component appears: `onMounted`

A component goes through stages: created, drawn into the page (**mounted**), updated, removed (**unmounted**). A function passed to `onMounted` runs once the component is on the page:

```
import { onMounted } from 'vue'

onMounted(load)            // load the data when the page opens
```

```
async function load() {
  try {
    const [e, d] = await Promise.all([api.get('/employees'), api.get('/departments')])
    employees.value = e.data
    departments.value = d.data
  } catch (err) {
    error.value = err.message
  } finally {
    loading.value = false
  }
}
```

`Promise.all` sends both requests at once and waits for both; `finally` runs after either the `try` or the `catch`.

Two ways to write a component

Vue has two ways of declaring a component's state and logic, and both appear in tutorials and in generated code:

Options API

```
<script>
export default {
  data() {
    return { count: 0 }
  },
  methods: {
    increment() { this.count++ }
  },
  mounted() { console.log('mounted') },
}
</script>
```

Composition API, with `<script setup>`

```
<script setup>
import { ref, onMounted } from 'vue'

const count = ref(0)
const increment = () => count.value++

onMounted(() => console.log('mounted'))
</script>
```

This course uses the **Composition API with `<script setup>`**, the form `create-vue` generates. Code from an old tutorial or from an LLM may switch to the Options API: `data()`, `methods:` and `this.` are the signs.

Components inside components

A page is built from smaller components. A parent passes data down to a child as **props**; a child tells its parent that something happened by emitting an **event**:

```
EmployeesView                          (has the list of departments)
  │  :departments="departments"    ↓ props: data down
  │  @created="added"              ↑ events: news up
  └─ EmployeeForm                      (posts a new employee)
```

- a child never changes its props; it emits an event, and the parent changes its own state
- in the child: `defineProps` declares the props it accepts, `defineEmits` the events it emits

The child: `EmployeeForm.vue`, script

```
<script setup>
import { ref } from 'vue'
import api from '@/services/api'

const props = defineProps({ departments: Array })
const emit = defineEmits(['created'])

const name = ref('')
const email = ref('')
const departmentId = ref(null)

async function save() {
  const response = await api.post('/employees', {
    name: name.value, email: email.value, departmentId: departmentId.value,
  })
  emit('created', response.data)
  name.value = ''
  email.value = ''
}
</script>
```

The child: `EmployeeForm.vue`, template

```
<template>
  <form @submit.prevent="save">
    <input v-model="name" placeholder="Name" required />
    <input v-model="email" type="email" placeholder="E-mail" required />
    <select v-model="departmentId" required>
      <option v-for="d in props.departments" :key="d.id" :value="d.id">{{ d.name }}</option>
    </select>
    <button type="submit">Add</button>
  </form>
</template>
```

The parent uses it

```
<script setup>
import EmployeeForm from '@/components/EmployeeForm.vue'

function added(employee) {
  employees.value.push(employee)
}
</script>

<template>
  <EmployeeForm :departments="departments" @created="added" />
</template>
```

- an imported component is used in the template as an element with its file's name
- `:departments="departments"` : the parent's reactive list becomes the child's prop, and stays up to date
- `@created="added"` : when the child emits `created` , `added` runs with the event's value: the new employee as the backend returned it, with its id

Pages: Vue Router

The application is one HTML page. **Vue Router** shows a different component for each address, and changes the address in the browser's address bar without loading a new page: a **single-page application**.

```
// src/router/index.js
import { createRouter, createWebHistory } from 'vue-router'
import HomeView from '../views/HomeView.vue'
import EmployeesView from '../views/EmployeesView.vue'
import EmployeeView from '../views/EmployeeView.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    { path: '/', name: 'home', component: HomeView },
    { path: '/employees', name: 'employees', component: EmployeesView },
    { path: '/employees/:id', name: 'employee', component: EmployeeView },
  ],
})

export default router
```

`createWebHistory` : addresses look like normal paths, `/employees/1` .

Links and the place for the page

```
<!-- src/App.vue -->
<script setup>
import { RouterLink, RouterView } from 'vue-router'
</script>

<template>
  <nav>
    <RouterLink to="/">Home</RouterLink>
    <RouterLink to="/employees">Employees</RouterLink>
  </nav>
  <RouterView />
</template>
```

- `<RouterLink to="…">`: a link that changes the page without reloading it
- `<RouterView />`: where the current address's component is drawn; the `<nav>` stays on every page
- a page opened directly or reloaded at `/employees/1` is requested from the server by that path: the server must answer every such path with `index.html`. Vite's development server does

An address with a parameter

`path: '/employees/:id'` : `:id` matches any value in that place, and the component reads it:

```
<script setup>
import { ref, onMounted } from 'vue'
import { useRoute } from 'vue-router'
import api from '@/services/api'

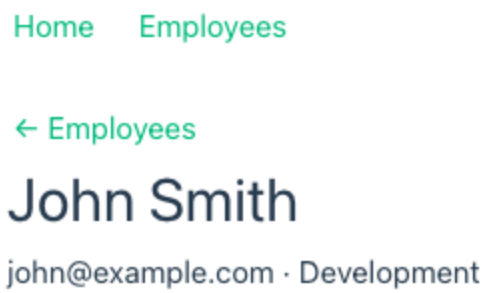
const route = useRoute()
const employee = ref(null)

onMounted(async () => {
  const response = await api.get(`/employees/${route.params.id}`)
  employee.value = response.data
})
</script>
```

An address with a parameter: the template

```
<template>
  <RouterLink to="/employees">← Employees</RouterLink>
  <div v-if="employee">
    <h1>{{ employee.name }}</h1>
    <p>{{ employee.email }} · {{ employee.departmentName }}</p>
  </div>
</template>
```

`/employees/1`



`route.params.id` is `'1'`, a string. `useRouter().push('/employees')` changes the page from code, for example after a save.

Origins

An **origin** is the scheme, host and port of an address:

Address	Origin
`http://localhost:5173/employees`	`http://localhost:5173`
`http://localhost:8080/api/employees`	`http://localhost:8080` : another port, another origin
`https://staff.example.com/api`	`https://staff.example.com`

During development two programs run: Vite's server on port 5173 hands out the frontend, the Spring backend on port 8080 answers `/api`. For the browser they are two origins.

The **same-origin policy**: a page's JavaScript may send a request to another origin, but may read the answer only if that origin allows it.

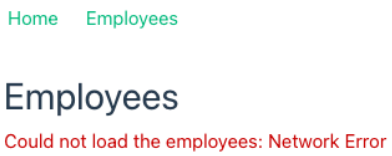
CORS

CORS (Cross-Origin Resource Sharing) is how a server allows it: an `Access-Control-Allow-Origin` header in its answer. The backend sends none:

```
$ curl -s -i localhost:8080/api/employees -H "Origin: http://localhost:5173"
HTTP/1.1 200
Content-Type: application/json
Content-Length: 167
```

With `baseURL: 'http://localhost:8080/api'`, the browser's console says:

```
Access to XMLHttpRequest at 'http://localhost:8080/api/employees' from origin 'http://localhost:5173'
has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
```



One origin instead: a proxy

Two ways out:

1. the backend sends CORS headers for the frontend's origin
2. the browser talks to **one** origin only, and that origin passes `/api` requests on to the backend: a **reverse proxy**

This course uses the second. In development Vite's server is the proxy; on the server, the web server that hands out the frontend does the same. The frontend's code always requests the relative path `/api/…` and never names a host or a port.

Without a proxy, Vite answers `/api` itself, with the application's page:

```
$ curl -s -i localhost:5173/api/employees | head -3
HTTP/1.1 200 OK
Vary: Origin
Content-Type: text/html
```

Vite's proxy

```
// vite.config.js
export default defineConfig({
  plugins: [vue(), vueDevTools()],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url)),
    },
  },
  server: {
    proxy: {
      '/api': 'http://localhost:8080',
    },
  },
})
```

- `plugins`: Vue's plugin, which reads `.vue` files, and the Vue DevTools plugin
- `server.proxy`: settings of the development server's proxy
- `'/api': 'http://localhost:8080'`: every request whose path starts with `/api` is sent on to the backend, with the same path
- `resolve.alias`: the `@` in `@/services/api` is `src/`
- Vite reads this file when it starts: after a change, stop `npm run dev` and start it again

The request, through the proxy

```
1. EmployeesView:   api.get('/employees')
2. Axios requests:  http://localhost:5173/api/employees      ← the page's own origin
3. Vite:            the path starts with /api
4. Vite requests:   http://localhost:8080/api/employees      ← server to server: no CORS
5. Spring answers:  JSON
6. Vite passes it to the browser, the page gets response.data
```

```
$ curl -s -i localhost:5173/api/employees | head -3
HTTP/1.1 200 OK
Vary: Origin
content-type: application/json
$ curl -s localhost:5173/api/employees
[{"id":1,"name":"John Smith","email":"john@example.com","departmentName":"Development"},
 {"id":2,"name":"Jane Doe","email":"jane@example.com","departmentName":"Sales"}]
```

The same-origin policy is a rule of browsers: requests from one server to another are not limited by it.

The employees page: script

```
<script setup>
import { ref, computed, onMounted } from 'vue'
import api from '@/services/api'
import EmployeeForm from '@/components/EmployeeForm.vue'

const employees = ref([])
const departments = ref([])
const loading = ref(true)
const error = ref(null)
const search = ref('')

const shown = computed(() =>
  employees.value.filter((e) => e.name.toLowerCase().includes(search.value.toLowerCase())),
)

async function load() { … }              // GET both lists with Promise.all, set loading and error

async function remove(id) {
  await api.delete(`/employees/${id}`)
  employees.value = employees.value.filter((e) => e.id !== id)
}

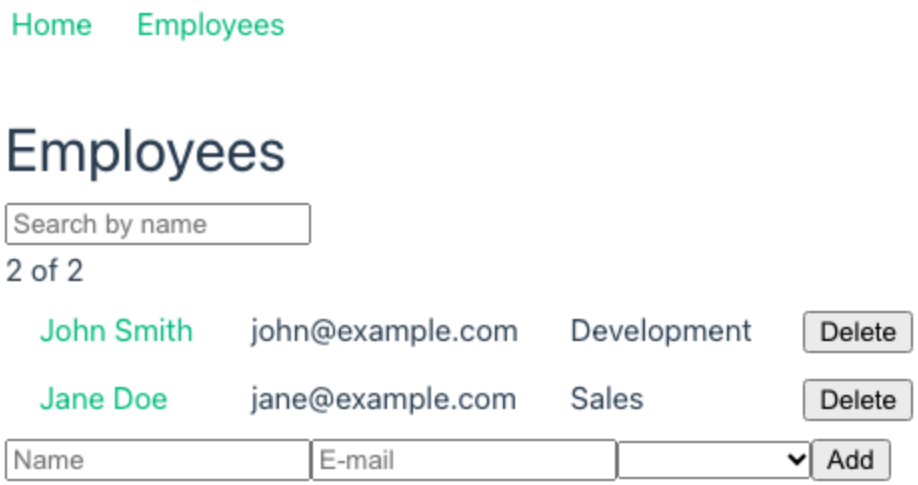
function added(employee) {
  employees.value.push(employee)
}

onMounted(load)
</script>
```

The employees page: template

```
<template>
  <h1>Employees</h1>
  <p v-if="loading">Loading…</p>
  <p v-else-if="error" class="error">Could not load the employees: {{ error }}</p>
  <template v-else>
    <input v-model="search" placeholder="Search by name" />
    <p>{{ shown.length }} of {{ employees.length }}</p>
    <table>
      <tr v-for="e in shown" :key="e.id">
        <td><RouterLink :to="`/employees/${e.id}`">{{ e.name }}</RouterLink></td>
        <td>{{ e.email }}</td>
        <td>{{ e.departmentName }}</td>
        <td><button @click="remove(e.id)">Delete</button></td>
      </tr>
    </table>
    <EmployeeForm :departments="departments" @created="added" />
  </template>
</template>
```

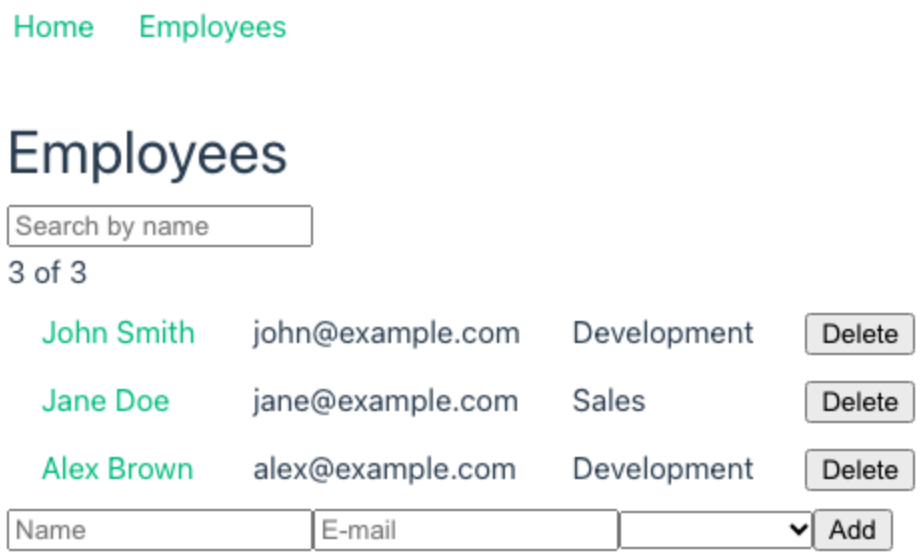
The page running



A new employee, posted the way the form posts it:

```
$ curl -s localhost:5173/api/employees -H 'Content-Type: application/json'
    -d '{"name":"Alex Brown","email":"alex@example.com","departmentId":1}'
{"id":3,"name":"Alex Brown","email":"alex@example.com","departmentName":"Development"}
```

After adding



The form's `created` event added the answer to `employees`; the count, the table and the search results follow from it. Delete removes the row the same way: the backend first, then the array.

Seeing what the page does

The browser's developer tools (F12, or Cmd-Option-I on a Mac):

Tab	Shows
Console	errors, the CORS message, and anything the code prints with `console.log(…)`
Network	every request: its address, method, status, and the request and response bodies
Elements	the current DOM, as Vue left it

A request that does not work: in **Network**, its address shows whether it went to `localhost:5173/api/…`, through the proxy, or somewhere else; its status and response show the backend's answer.

`create-vue` also adds **Vue DevTools**: `http://localhost:5173/__devtools__/` shows the components and their current state.

Building for production

```
$ npm run build

vite v8.3.1 building client environment for production...
✓ 93 modules transformed.
dist/index.html                   0.42 kB │ gzip:  0.28 kB
dist/assets/index-D3qsKolI.css    3.05 kB │ gzip:  1.07 kB
dist/assets/index-BI5xrAm3.js   152.32 kB │ gzip: 57.59 kB
✓ built in 492ms
```

- `vite build` turns `src/` and the packages it uses into three files in `dist/`: all `.vue` files and all JavaScript become one script
- the names carry a hash of the content, `index-BI5xrAm3.js`: a changed file gets a new name, so no browser keeps an old copy
- `dist/` is static files: any web server can hand them out. No Node.js runs on the server, and the `/api` proxy there is the web server's job

The commands

Command	Does
`node --version`	the installed Node.js version: 22 from 22.18, or 24 from 24.12
`npm create vue@latest <name> -- --router`	a new project with Vue Router
`npm install`	install what `package.json` lists; after a `git clone`, and after a teammate added a package
`npm install <package>`	add a package, and record it in `package.json` and `package-lock.json`
`npm run dev`	the development server, `http://localhost:5173`
`npm run build`	the production build, into `dist/`
`npm run preview`	serve `dist/` locally, to try the production build

Vue in one table

Write	Does
`const x = ref(0)`, `x.value`	reactive state; `.value` in the script only
`computed(() => …)`	a value calculated from reactive state
`{{ expression }}`	text
`:attr="expression"`	an attribute from a value
`@event="code"`, `@submit.prevent`	run code on an event; without the browser's default
`v-model="x"`, `v-model.number`	a form field and a variable, both ways
`v-if`, `v-else-if`, `v-else`, `v-show`	exists only when; hidden when
`v-for="e in list" :key="e.id"`	one element per item
`onMounted(fn)`	run when the component is on the page
`defineProps`, `defineEmits`, `emit('name', value)`	data in from the parent; events out to it
`useRoute().params`, `useRouter().push(path)`	the address's parameters; go to another page



Questions