LU12a - Daten laden mit fetch()

In modernen Web-Applikationen werden Inhalte selten direkt im Code gespeichert. Stattdessen kommen sie von einem Server — einem CMS, einer Datenbank oder einer öffentlichen API. Die Browser-Funktion `fetch()` ist das wichtigste Werkzeug, um solche Daten zu laden.

1. Warum async? Das Single-Thread-Problem

JavaScript ist **single-threaded**: Es kann immer nur eine Operation gleichzeitig ausführen. Stellen Sie sich vor, eine Funktion wartet drei Sekunden auf eine Server-Antwort:

```
console.log('Seite lädt...');
const daten = getDatenVomServer(); // ← blockiert für 3 Sekunden
console.log('Fertig');             // ← erscheint erst nach 3 Sekunden
```

Während dieser drei Sekunden würde **die gesamte App einfrieren** — kein Klick, kein Scrollen, keine Eingabe würde reagieren.

Asynchroner Code löst das Problem: Die lange Operation wird gestartet und in den Hintergrund geschickt. Der restliche Code läuft weiter. Wenn die Antwort eintrifft, wird die Funktion fortgesetzt.

```
Zeit:  ──────────────────────────────────────────────────────►

       fetchFaqs() startet
       │
       ├── fetch() abgeschickt → läuft im Hintergrund
       │
       │   Vue rendert das Template (isLoading-Spinner
sichtbar)
       │   Benutzer kann klicken, scrollen, tippen
       │
       │   [Server antwortet]
       │
       └── Daten kommen an → faqItems.value = data → UI
updated
```

2. Promise: Das Versprechen

`fetch()` gibt nicht sofort Daten zurück — das kann es gar nicht. Stattdessen gibt es ein **Promise** zurück: ein Objekt, das ein zukünftiges Ergebnis repräsentiert.

Ein Promise hat drei mögliche Zustände:

Zustand	Bedeutung
`pending`	Anfrage läuft, Antwort noch nicht da
`fulfilled`	Antwort eingetroffen, Daten verfügbar
`rejected`	Etwas ist schiefgegangen (Netzwerkfehler etc.)

> **Analogie:** Ein Promise ist wie eine Bestellbestätigung im Online-Shop. Sie bekommen sie sofort — aber das Paket kommt erst später. Die Bestätigung sagt: „Ich liefere dir etwas. Ich weiss noch nicht genau wann und ob alles klappt.“

3. async / await

Die modernste Art, mit Promises umzugehen. `async` und `await` machen asynchronen Code lesbar — er sieht aus wie normaler, sequenzieller Code:

```
const fetchFaqs = async () => {       // 'async' markiert
die Funktion
  const response = await fetch(url);  // 'await' pausiert
diese Funktion bis die Antwort da ist
  const data = await response.json(); // nochmals 'await' –
auch das Parsen ist async
  faqItems.value = data;
};
```

Wichtige Regeln:

- `await` kann nur innerhalb einer `async`-Funktion verwendet werden
- `await` pausiert **nur die aktuelle Funktion** — nicht die gesamte App
- Jedes `await` wartet auf ein Promise

4. Das Response-Objekt

Nach `await fetch()` erhalten wir kein Array mit Daten — sondern ein **Response-Objekt**. Das ist eine eingebaute Browser-Klasse mit festen Eigenschaften:

```
const response = await fetch('https://...mockapi.io/faqs');
console.log(response);

// Ausgabe in der Konsole:
// Response {
//   type: "cors",
```

```
//   url: "https://...mockapi.io/faqs",
//   status: 200,
//   ok: true,
//   statusText: "",
//   headers: Headers { ... },
//   body: ReadableStream,
//   bodyUsed: false
// }
```

Eigenschaft	Bedeutung
`status`	HTTP-Statuscode (bekannt aus M290): `200` OK, `404` Not Found, `500` Server Error
`ok`	`true` wenn Status zwischen 200–299, sonst `false`
`headers`	Antwort-Headers (Metadaten der HTTP-Antwort)
`body`	Die eigentlichen Daten — als `ReadableStream`
`bodyUsed`	`false` = Stream noch nicht gelesen, `true` = bereits gelesen

Warum ist body ein ReadableStream?

Wenn `await fetch()` zurückkommt, sind die Header vollständig angekommen — aber der Body (die eigentlichen Daten) wird als **Datenstrom** geliefert. Der Browser hat die Bytes empfangen, aber noch nicht interpretiert.

Das ist vergleichbar mit einem Brief im Briefkasten:

- `await fetch()` → Der Brief liegt im Kasten. Er ist da.
- `await response.json()` → Sie öffnen den Umschlag, lesen und verstehen den Inhalt.

Den Body auslesen

Um die Daten aus dem Stream zu lesen und in ein JavaScript-Objekt umzuwandeln:

```
const data = await response.json();
```

Auch das braucht `await`: Das Lesen und Parsen des Streams ist bei grossen Antworten messbar zeitaufwändig.

5. Fehlerbehandlung: try / catch / finally

Bei einer externen Anfrage kann vieles schiefgehen: Netzwerkausfall, Server überlastet, URL falsch. Ohne Fehlerbehandlung würde die App abstürzen.

```
const fetchFaqs = async () => {
  try {
    // Code, der einen Fehler werfen könnte
    const response = await fetch(url);
    const data = await response.json();
    faqItems.value = data;

  } catch (err) {
    // Wird ausgeführt wenn in 'try' ein Fehler geworfen
wird
    console.error('Fehler:', err.message);

  } finally {
    // Wird IMMER ausgeführt – egal ob Erfolg oder Fehler
    isLoading.value = false;
  }
};
```

Block	Wann wird er ausgeführt?
`try`	Immer — enthält den „normalen“ Code
`catch`	Nur wenn in `try` ein Fehler geworfen wird
`finally`	

|

Die Besonderheit von fetch(): response.ok

`fetch()` wirft bei HTTP-Fehlercodes **keinen** JavaScript-Fehler. Ein `404 Not Found` oder `500 Server Error` landet **nicht** automatisch im `catch`-Block — aus Sicht von `fetch()` war die Kommunikation erfolgreich, es kam ja eine Antwort.

Deshalb prüfen wir `response.ok` manuell und werfen mit `throw` selbst einen Fehler:

```
const response = await fetch(url);

if (!response.ok) {
  throw new Error(`HTTP-Fehler! Status:
${response.status}`);
  // throw leitet den Fehler weiter in den catch-Block
}
```

`throw` unterbricht die Funktion sofort und springt in den `catch`-Block — genau wie ein natürlicher

JavaScript-Fehler.

6. Das Drei-Zustands-Muster

Beim Laden von Daten gibt es immer drei mögliche Zustände. Gutes UI-Design zeigt dem Benutzer jeden davon:

```
┌──────────────────────────────────────────────────────┐
│  Zustand 1: LADEN      isLoading = true              │
│  → Spinner / "Daten werden geladen..."               │
├──────────────────────────────────────────────────────┤
│  Zustand 2: FEHLER     error = "Fehlermeldung"       │
│  → Fehlermeldung anzeigen                            │
├──────────────────────────────────────────────────────┤
│  Zustand 3: DATEN DA   faqItems = [...]              │
│  → Inhalte rendern                                   │
└──────────────────────────────────────────────────────┘
```

In Vue werden diese drei Zustände als reaktive Variablen modelliert:

```
const faqItems  = ref([]);    // die eigentlichen Daten
const isLoading = ref(false); // lädt gerade?
const error     = ref(null);  // Fehlermeldung (null = kein
Fehler)
```

Im Template werden die drei Zustände mit `v-if / v-else-if / v-else` gesteuert:

```
<div v-if="isLoading">Daten werden geladen...</div>

<div v-else-if="error" class="error-message">
  {{ error }}
</div>

<div v-else>
  <AccordionItem v-for="item in faqItems" :key="item.id"
:faq="item" />
</div>
```

7. Die vollständige fetchFaqs-Funktion

Alle Konzepte zusammen — das ist das vollständige Muster für eine fetch-Funktion in Vue:

```
import { ref, onMounted } from 'vue';

const faqItems  = ref([]);
const isLoading = ref(false);
const error     = ref(null);

const fetchFaqs = async () => {
  isLoading.value = true;  // Laden beginnt
  error.value = null;      // allfälligen früheren Fehler zurücksetzen

  try {
    const response = await fetch(
      'https://...mockapi.io/faqs' // ← Ihren Endpoint eintragen
    );

    if (!response.ok) {
      throw new Error(`HTTP-Fehler! Status: ${response.status}`);
    }

    const data = await response.json();
    faqItems.value = data; // ← Vue updated das Template automatisch

  } catch (err) {
    error.value = 'Daten konnten nicht geladen werden: ' + err.message;

  } finally {
    isLoading.value = false; // Laden beendet – egal ob Erfolg oder Fehler
  }
};

onMounted(fetchFaqs); // Funktion aufrufen sobald Komponente bereit ist
```

8. Warum onMounted?

onMounted ist ein Vue-Lifecycle-Hook: Er wird ausgeführt, sobald die Komponente das erste Mal ins DOM eingehängt wurde.

```
setup()      →  Komponente wird initialisiert
             →  Template wird gerendert (DOM ist bereit)
onMounted()  →  wird jetzt aufgerufen  ← fetch startet hier
```

onMounted läuft garantiert nur im Browser — nicht auf dem Server (relevant bei Server-Side Rendering mit z. B. Nuxt.js). Es ist ausserdem die konventionelle Stelle für alles, das nach dem ersten Render passieren soll.

9. API-Strukturen vergleichen

Verschiedene APIs strukturieren ihre JSON-Antworten unterschiedlich. Immer zuerst `console.log(data)` aufrufen und die Struktur in der Konsole inspizieren:

API	Antwort-Struktur	Zugriff auf die Daten
MockAPI.io	Daten direkt als Array	`faqItems.value = data`
Sanity.io	Daten unter `result`	`faqItems.value = data.result`
Andere	Evtl. `data.items`, `data.faqs` etc.	Konsole prüfen

```
// MockAPI – Daten direkt:
// [ { id: "1", question: "...", answer: "..." }, ... ]
faqItems.value = data;

// Sanity.io – Daten verschachtelt:
// { result: [ { _id: "faq-1", question: "...", answer: "..." }, ... ] }
faqItems.value = data.result;
```

10. Testen im Browser

Die Browser DevTools sind das wichtigste Werkzeug beim Entwickeln mit `fetch()`:

Network-Reiter (F12 → Network):

- Zeigt alle HTTP-Requests der Seite
- Status-Code des Requests prüfen (`200`?)
- Unter **Response** das rohe JSON inspizieren
- Unter **Timing** die Ladezeit sehen

Fehlerfall simulieren:

- *URL absichtlich falsch schreiben* → Server antwortet mit `404`, `response.ok` ist `false`
- *Network → Offline stellen* → `fetch()` kann Server nicht erreichen, `catch` greift
- *Network → Slow 3G* → Ladezustand (Spinner) wird sichtbar

Weiterführende Ressourcen

- Net Ninja – Async JavaScript (Playlist, 11 Videos, Englisch) — Empfehlung: Video 1, 7 und 9
- MDN – Fetch API
- MDN – Response Objekt
- Vue.js – Lifecycle Hooks

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/de/modul/m291/learningunits/lu12/theorie/a_fetch2?rev=1779057859

Last update: **2026/05/18 00:44**