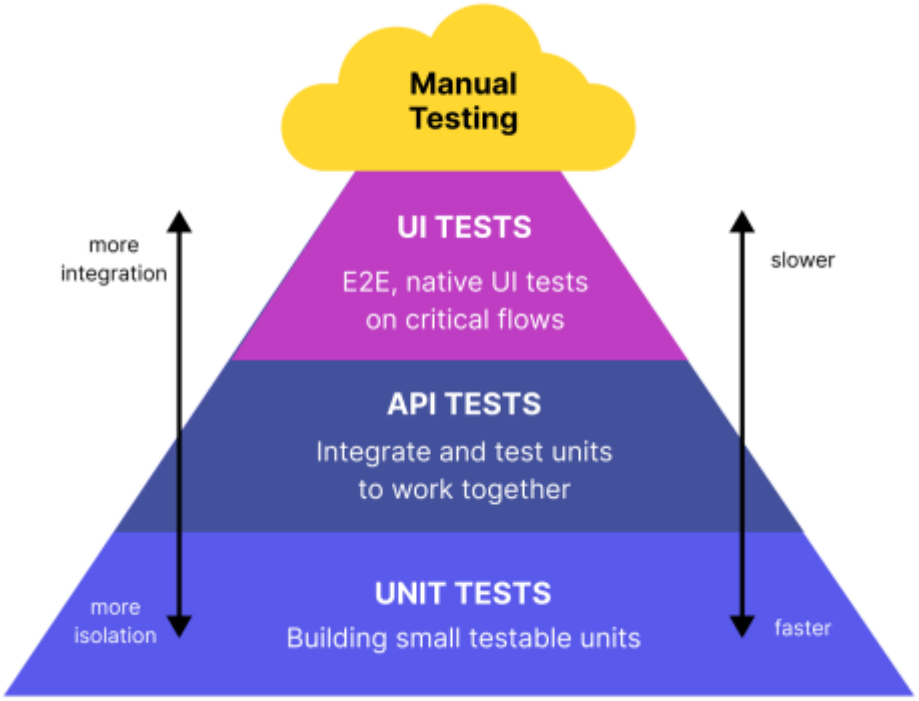
LU16b - Testing

Testing stellt sicher, dass eine Web-Applikation so funktioniert, wie erwartet – bevor sie in Production gelangt. Es gibt vier Testarten, die sich in Umfang, Geschwindigkeit und Aufwand unterscheiden.



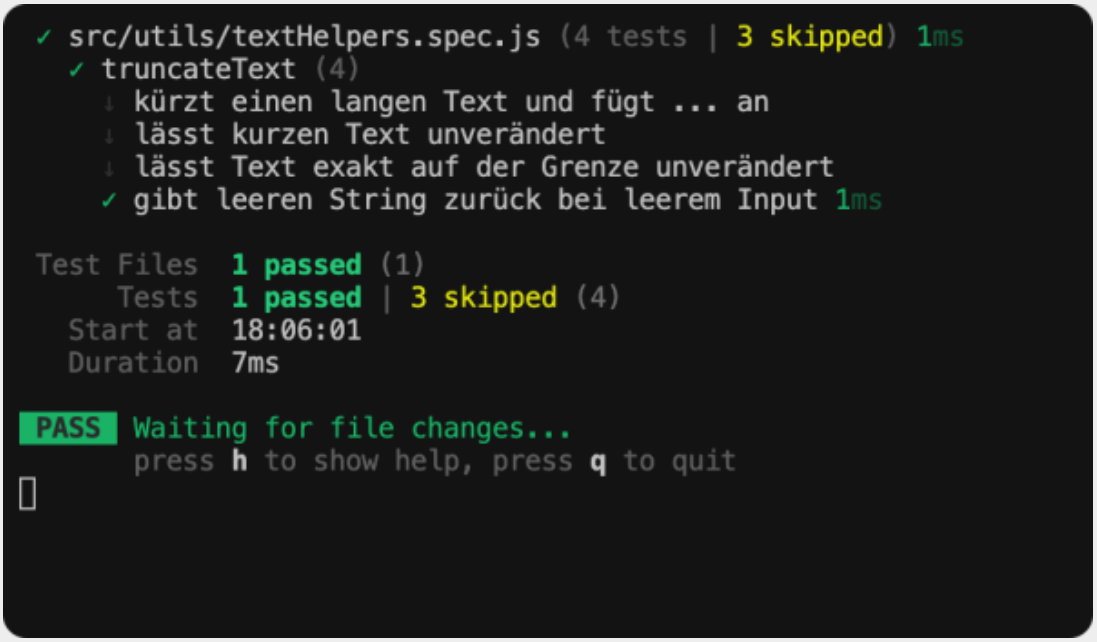
Testing-Pyramide mit vier Ebenen

Die Pyramide zeigt die empfohlene Gewichtung:

- **Unten (breit):** Viele schnelle Unit Tests – testen einzelne isolierte Funktionen.
- **Mitte:** Weniger Integration Tests – testen das Zusammenspiel von Funktionen.
- **Oben (schmal):** Wenige, langsame End-to-End Tests – aufwändig, aber realitätsnah.
- **Wolke:** Manuelles Testen – nicht automatisierbar, aber unverzichtbar.

Unit Test

Ein Unit Test prüft eine **einzelne, isolierte Funktion** – ohne Abhängigkeiten von aussen (kein Browser, keine API, keine Datenbank).



Screenshot Vitest-Output im Terminal mit grünem PASS

Werkzeug: Vitest

Beispiel: Eine Funktion, die langen Text abkürzt.

```
// textHelpers.js
export function truncateText(text, maxLength) {
  if (text.length <= maxLength) return text
  return text.slice(0, maxLength) + '...'
}
```

```
// textHelpers.spec.js
import { truncateText } from './textHelpers.js'
import { test, expect } from 'vitest'

test('kürzt langen Text ab', () => {
  expect(truncateText('Was ist Vue.js?', 6)).toBe('Was
is...')
})

test('lässt kurzen Text unverändert', () => {
  expect(truncateText('Hallo', 10)).toBe('Hallo')
})
```

Eigenschaft	Unit Test
Geschwindigkeit	Sehr schnell (Millisekunden)
Browser nötig?	Nein
Echte API nötig?	Nein
Testet	Eine einzelne Funktion

Integration Test

Ein Integration Test prüft das **Zusammenspiel mehrerer Teile** – zum Beispiel wie eine Funktion einen API-Aufruf durchführt und die Antwort verarbeitet.

[Platzhalter: Screenshot Vitest mit Integration Test und gemocktem fetch]

Werkzeug: Vitest

Beispiel: Die `fetchFaqs()`-Funktion abrufen und prüfen, ob die Daten korrekt ankommen.

```
// fetchFaqs.spec.js
import { test, expect, vi } from 'vitest'
import { fetchFaqs } from './fetchFaqs.js'

vi.stubGlobal('fetch', () => Promise.resolve({
  json: () => Promise.resolve([{ id: '1', question: 'Was ist
Vue?' }])
}))

test('gibt FAQ-Daten zurück', async () => {
  const data = await fetchFaqs('https://mockapi.io/faqs')
  expect(data[0].question).toBe('Was ist Vue?')
})
```

Warum `fetch` mocken? Im Test soll nicht die echte MockAPI.io aufgerufen werden. Mit `vi.stubGlobal` wird `fetch` durch eine Fake-Version ersetzt, die sofort antwortet – ohne Internetverbindung, immer gleich.

Eigenschaft	Integration Test
Geschwindigkeit	Schnell (Millisekunden)

Eigenschaft	Integration Test
Browser nötig?	Nein
Echte API nötig?	Nein (gemockt)
Testet	Fetch + Datenverarbeitung zusammen

End-to-End Test (E2E)

Ein E2E Test öffnet einen **echten Browser** und simuliert, wie eine Nutzerin oder ein Nutzer mit der App interagiert – Klicks, Formular-Eingaben, Navigation.

[Platzhalter: Screenshot Playwright Test Runner mit Browserfenster und Testergebnis]

Werkzeug: Playwright

Beispiel: Prüfen, ob das FAQ-Accordion korrekt öffnet und schliesst.

```
// accordion.spec.js
import { test, expect } from '@playwright/test'

test.beforeEach(async ({ page }) => {
  await page.goto('http://localhost:5173', { waitUntil:
'networkidle' })
})

test('Antwort ist standardmässig nicht sichtbar', async ({
page }) => {
  await expect(
    page.getByText('Your laptop has absorbed your
energy...')
  ).not.toBeVisible()
})

test('Antwort erscheint nach Klick auf die Frage', async ({
page }) => {
  await page
    .getByRole('button', { name: 'Why does my code work on
my machine?' })
    .click()
```

```
  await expect(
    page.getByText('Your laptop has absorbed your
energy...')
  ).toBeVisible()
})
```

Warum `networkidle`? Die FAQ-Daten werden von MockAPI.io geladen. `waitUntil: 'networkidle'` wartet, bis alle Netzwerkanfragen abgeschlossen sind – erst dann starten die Tests.

Eigenschaft	E2E Test
Geschwindigkeit	Langsam (Sekunden pro Test)
Browser nötig?	Ja (Chromium, Firefox, WebKit)
Echte API nötig?	Ja (oder Staging-URL)
Testet	Vollständiger Nutzerfluss im Browser

Manuelles Testen

Manuelles Testen kann nicht vollständig automatisiert werden – es erfordert menschliches Urteilsvermögen. Typische Aufgaben:

- Responsive Design auf verschiedenen Bildschirmgrössen prüfen
- Cross-Browser-Kompatibilität testen (Chrome, Firefox, Edge)
- Visuelle Qualität und Nutzererlebnis beurteilen
- Accessibility mit Tastatur und Screenreader prüfen

[Platzhalter: Screenshot Chrome DevTools mit aktivierter Device Toolbar, verschiedene Geräte-Voreinstellungen sichtbar]

Werkzeug: Chrome DevTools (kostenlos, im Browser integriert)

Öffnen Sie die DevTools mit **F12**, dann klicken Sie auf das **Geräte-Symbol** oben links oder drücken Sie **Ctrl+Shift+M**. Sie können nun die Breite frei einstellen oder ein Gerät aus der Liste wählen.

Empfohlene Testbreiten:

Gerät	Breite
Mobile (z.B. iPhone SE)	375 px

Gerät	Breite
Tablet (z.B. iPad)	768 px
Desktop	1280 px

Testprotokoll

Für das manuelle Testen wird ein **Testprotokoll** geführt. Es dokumentiert, was getestet wurde, was erwartet wurde und was tatsächlich passiert ist.

#	Testfall	Erwartetes Resultat	Effektives Resultat	Status
1	Seite aufrufen	FAQ-Liste wird geladen und angezeigt		☐ ☐ ☐ ☐
2	Auf eine Frage klicken	Antwort wird sichtbar		☐ ☐ ☐ ☐
3	Nochmals auf dieselbe Frage klicken	Antwort schliesst sich		☐ ☐ ☐ ☐
4	Tab-Taste: durch Fragen navigieren	Fokus-Rahmen springt sichtbar von Frage zu Frage		☐ ☐ ☐ ☐
5	Enter auf fokussierter Frage	Antwort öffnet sich		☐ ☐ ☐ ☐
6	DevTools: 375 px	Layout passt sich an, kein horizontales Scrollen		☐ ☐ ☐ ☐
7	Echtes Smartphone	App funktioniert, Klick öffnet die Antwort		☐ ☐ ☐ ☐
8	Firefox oder Edge	App lädt und verhält sich gleich wie in Chrome		☐ ☐ ☐ ☐

Vergleich: Die vier Testarten

	Unit Test	Integration Test	E2E Test	Manueller Test
Werkzeug	Vitest	Vitest	Playwright	Mensch + DevTools
Geschwindigkeit	⚡ ms	⚡ ms	☕ Sekunden	☕☕ Minuten
Browser?	Nein	Nein	Ja	Ja
Echte API?	Nein	Nein (Mock)	Ja	Ja
Testet	1 Funktion	Mehrere Schritte	Nutzerfluss	Visuell, UX, A11y

→ Bundling & Deploy