Web Engineering — Prüfungs-Zusammenfassung

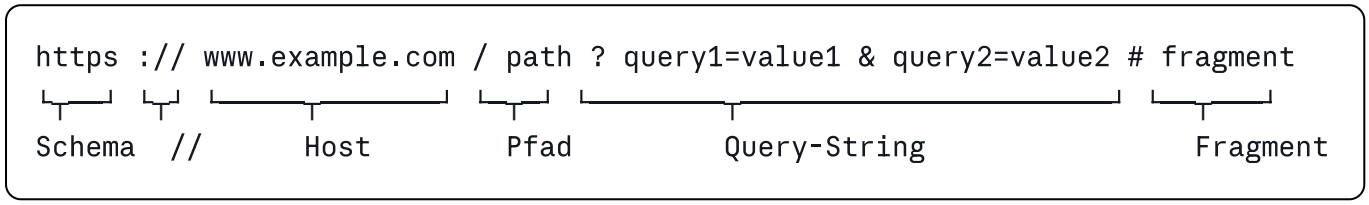
Diese Zusammenfassung deckt alle Themen der hochgeladenen Altprüfung ab, gegliedert nach Wissensbereich. Bei jedem Thema steht in Klammern, welche Prüfungsfrage es betraf. Stellen, an denen erfahrungsgemäss Punkte verloren gehen, sind als **Merke** markiert.

---

1. Web-Grundlagen

URL-Aufbau (Frage 5)

Eine URL ist von links nach rechts streng aufgebaut. Die Reihenfolge musst du im Schlaf können:



Im Detail, Baustein für Baustein: `https` (Schema/Protokoll) → `:` → `//` → `www.` (Subdomain) → `example.` (Domain) → `com` (Top-Level-Domain) → `/` → `path` (Pfad) → `?` (leitet Query ein) → `query1=value1` → `&` (trennt Parameter) → `query2=value2` → `#` (leitet Fragment ein) → `fragment` (Anker innerhalb der Seite).

Wichtig: Das `?` leitet **einen** Query-String ein, weitere Parameter werden mit `&` angehängt. Das `#`-Fragment wird **nicht** an den Server gesendet, es wird nur clientseitig (vom Browser) ausgewertet.

HTTP-Statuscodes (Frage 14)

Statuscodes sind in Klassen organisiert. Die genaue Bedeutung von 200/201/204 wird gerne verwechselt:

Code	Klasse	Bedeutung
200 OK	2xx Erfolg	Anfrage erfolgreich, angeforderte Daten **sind** im Body
201 Created	2xx Erfolg	Neue Ressource erstellt (z. B. „neues Buch hinzugefügt"), Details im Body
204 No Content	2xx Erfolg	Erfolgreich, aber **keine** Daten im Body
301 / 302	3xx Umleitung	Permanente / temporäre Weiterleitung
400 Bad Request	4xx Client-Fehler	Anfrage wegen Client-Fehler nicht verarbeitbar
401 / 403	4xx Client-Fehler	Nicht authentifiziert / nicht berechtigt
404 Not Found	4xx Client-Fehler	Ressource nicht gefunden
500	5xx Server-Fehler	Interner Serverfehler

Merke (häufiger Punktverlust): 2**0**0 = OK mit Daten · 2**0**1 = Created (etwas Neues) · 2**0**4 = No Content (kein Body). Eselsbrücke: 20**1** liefert die **1** neu erstellte Ressource, 20**4** liefert **0** Inhalt.

Grobe Klassenlogik: **2xx** = hat geklappt, **3xx** = woanders suchen, **4xx** = du (Client) hast Mist gebaut, **5xx** = der Server hat Mist gebaut.

HTML-Formularelemente (Frage 1)

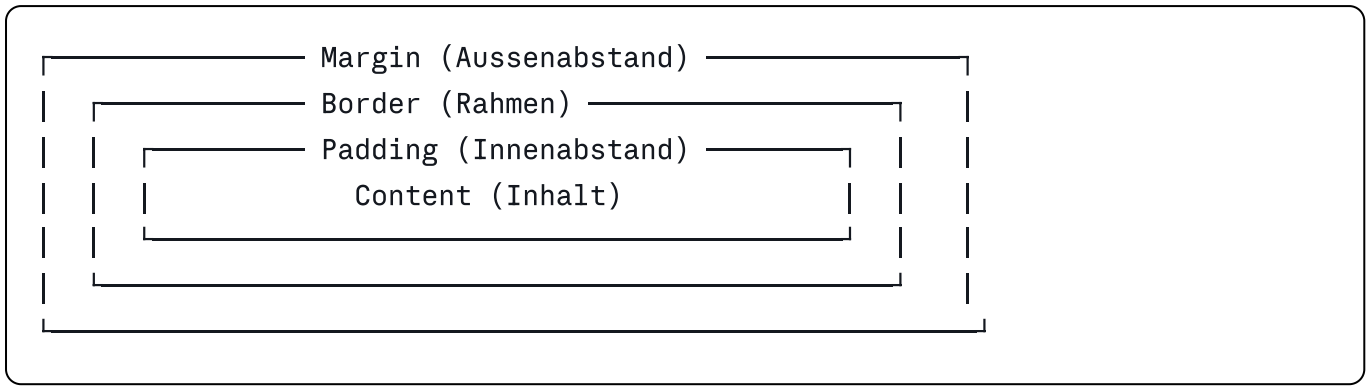
- `<form>` — **Container** für Formularelemente; bündelt Eingaben und definiert, wohin/wie gesendet wird (`action`, `method`).
- `<input type="text">` — **Eingabefeld** für Text. Der `type` bestimmt die Art (text, email, password, checkbox, radio …).
- `<button>` — **löst eine Aktion aus** (z. B. Absenden mit `type="submit"`).
- `<label>` — **beschriftet/beschreibt** eine Formularkontrolle. Über `for="id"` mit dem Feld verknüpft → wichtig für Barrierefreiheit (Klick aufs Label fokussiert das Feld).

---

2. CSS

Box Model (Frage 4)

Jedes HTML-Element ist eine Box aus vier Bereichen, von innen nach aussen:



Reihenfolge auswendig: **Content → Padding → Border → Margin**. `box-sizing: border-box` rechnet Padding und Border in die angegebene Breite/Höhe ein (sonst werden sie addiert).

CSS-Spezifität (Frage 6)

Spezifität entscheidet, welche Regel bei Konflikten gewinnt. Aufsteigend (niedrig → hoch):

1. **Universal-Selektor** `*`
2. **Typ-/Element-Selektor** `p`, `div`, `button`
3. **Klassen-Selektor** `.klasse` (auch Attribut- und Pseudoklassen `:hover`)
4. **ID-Selektor** `#id`

Darüber stehen noch Inline-Styles (`style="…"`) und ganz oben `!important`.
Gewichtungslogik: ID (100) > Klasse (10) > Element (1) > Universal (0).

---

3. JavaScript

Funktionen, `this`, Pfeilfunktionen (Frage 2)

- **Traditionelle Funktionen** (`function() {}`): haben ein **eigenes** `this`, das vom **Aufruf** abhängt — bei Methodenaufruf das Objekt, im `strict mode` `undefined`, sonst das globale Objekt. Sie **können** als Konstruktoren (mit `new`) verwendet werden.
- **Pfeilfunktionen** (`() => {}`): haben **kein eigenes** `this`, sondern erben es **lexikalisch** aus dem umgebenden Scope. Sie können **nicht** als Konstruktoren verwendet werden und eignen sich schlecht als Objekt-/Klassenmethoden, wenn man auf das Objekt-`this` zugreifen will.

Merke (Achtung Prüfungsfehler): Die Aussage „Das `this` in traditionellen Funktionen ist immer global" ist **falsch** — es hängt vom Aufrufkontext ab. Und die offizielle Lösung wertet „Pfeilfunktionen haben ihr eigenes this-Kontext" als *Wahr*, obwohl das fachlich **falsch** ist (sie haben gerade **kein** eigenes `this`). Kenne beide Versionen: inhaltlich korrekt ist „kein eigenes this", in der Prüfung war die Bewertung aber umgekehrt.

Events, Closures, Zähler (Frage 3)

```html
<button id="myButton">Klicke mich</button>
<script>
  let count = 0;
  document.getElementById('myButton').addEventListener('click', function() {
    count++;
    alert('Anzahl der Klicks: ' + count);
  });
</script>
```

`count` wird **einmal** mit 0 initialisiert. Die Event-Handler-Funktion bildet eine **Closure** über `count` — die Variable bleibt zwischen Klicks erhalten. Beim **ersten** Klick wird `count` zuerst auf 1 erhöht, **dann** angezeigt → „Anzahl der Klicks: **1**", danach 2, 3, … Klassische Falle: Es wird nicht „0" angezeigt, weil `count++` **vor** dem `alert` läuft.

`fetch`, async/await, REST-Aufruf (Frage 15)

```javascript
async function updateUser(userId, data) {
  const response = await fetch(`/users/${userId}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(data)
  });
  if (!response.ok) {
    throw new Error(`HTTP error! status: ${response.status}`);
  }
  return await response.json();
}
```

Das musst du zeilenweise lesen können:

- `method: 'PUT'` → die Funktion **aktualisiert** eine bestehende Ressource (PUT = ersetzen/aktualisieren; POST = neu erstellen; DELETE = löschen; GET = lesen).
- `JSON.stringify(data)` → JS-Objekt wird als JSON-String in den Request-Body geschrieben.

- `response.ok` ist `true` bei Statuscodes 200–299. Bei `!response.ok` wird eine **Ausnahme mit dem HTTP-Statuscode** geworfen.
- `return await response.json()` → gibt das **aktualisierte Benutzerobjekt** (geparstes JSON) zurück.

Korrekte Aussagen waren also: „aktualisiert einen Benutzer und gibt das aktualisierte Objekt zurück" **und** „bei Fehler wird eine Ausnahme mit dem Statuscode geworfen". Eine `async`-Funktion gibt zwar technisch immer ein Promise zurück, aber sie erstellt hier **keinen neuen** Benutzer (das wäre POST) und löscht auch nichts.

---

4. Architektur, Frameworks & APIs

Monolith vs. Microservices (Fragen 7, 11)

Monolith: eine einzige, eng gekoppelte Codebasis. Vorteil: einfach am Anfang. Nachteil: schwer skalierbar, ein Fehler kann das ganze System betreffen.

Microservices: viele kleine, unabhängige Dienste. Vorteile und ihre strategischen Begründungen (genau diese Zuordnung wurde geprüft):

Merkmal	Strategischer Vorteil
Dezentralisierte Datenverwaltung	Individuelle Sicherheits- und Compliance-Massnahmen pro Dienst möglich
Unabhängige Deploybarkeit	Operative Exzellenz; einzelne Dienste reagieren schnell auf Verkehrs-/Systemänderungen
Kleine, fokussierte Aufgaben	Klareres Code-Management, leichteres Onboarding neuer Entwickler
Eigenständige Technologie	Mehr Flexibilität in der Service-Entwicklung, schnellere Anpassung an Geschäftsanforderungen
Kommunikation über REST/Messaging	Reduziert Abhängigkeit von einzelnen Technologieentscheidungen → höhere Agilität

Hauptvorteil als MC-Frage: Fehlerisolierung (ein Dienst fällt aus, der Rest läuft weiter). *Geringe Komplexität* und *einheitliche Datenverwaltung* sind **keine** Microservice-Vorteile — im Gegenteil, verteilte Systeme sind komplexer.

Merke (grösster Punktverlust): Die Zuordnung von *Eigenständige Technologie*, *Unabhängige Deploybarkeit* und *Kommunikation REST/Messaging* nicht verwechseln. Eselsbrücke: **Deploybarkeit → reagieren** (operative Exzellenz), **eigene Technologie → Flexibilität**, **Kommunikation → weniger Abhängigkeit**.

REST-Prinzipien (Frage 13)

REST (Representational State Transfer) ist ein Architekturstil für Web-APIs:

- **Uniform Interface (Einheitliche Schnittstelle)** → fördert die **Skalierbarkeit** der Anwendung.
- **Stateless (Zustandslosigkeit)** → jeder Request enthält alle nötigen Infos; ermöglicht dem Server, die **Last effizient zu verteilen** (kein gespeicherter Sitzungszustand nötig).
- **Cacheable (Cache-Fähigkeit)** → Antworten können zwischengespeichert werden → verbessert die **Netzwerkeffizienz**.

Weitere REST-Prinzipien: Client-Server-Trennung, Layered System, (optional) Code on Demand. Daten meist als **JSON** (siehe `Content-Type: application/json`).

Rendering-Strategien (Kontext zu SPA/PWA)

- **SSR** (Server-Side Rendering): HTML wird auf dem Server erzeugt, schnelle erste Anzeige, gut für SEO.
- **SPA** (Single-Page Application): eine Seite, Inhalte werden per JS clientseitig nachgeladen.
- **SSG** (Static Site Generation): Seiten werden beim Build vorgerendert.

Data Binding (Frage 9)

Verknüpfung zwischen Datenmodell und View:

One-Way Binding (z. B. React): Daten fliessen in **eine** Richtung (Model → View).

- Bessere Performance durch weniger unnötige DOM-Manipulationen, besonders bei grossen Apps. ✔ wahr
- Stellt sicher, dass Änderungen keine unerwarteten Nebeneffekte an anderen Stellen verursachen (vorhersehbarer Datenfluss). ✔ wahr
- Erleichtert das Verständnis für Entwickler von **traditionellen** Frameworks? ✘ falsch (für die ist der einseitige Fluss eher ungewohnt).
- Führt zu erhöhter Komplexität durch manuelle View-Updates? ✘ falsch (genau die Vorhersehbarkeit reduziert Komplexität).

Two-Way Binding (z. B. Angular `[(ngModel)]`): Daten fliessen in **beide** Richtungen (Model ↔ View).

- Reduziert Codezeilen, da Model-Änderungen automatisch in der View erscheinen. ✔ wahr
- Erhöht aber die Wahrscheinlichkeit von Fehlerquellen durch direkte Modellaktualisierungen aus der View. ✔ wahr
- Kann bei zu vielen Bindungen Performance-Probleme verursachen. ✔ wahr

- Ist „immer intuitiver, unabhängig von Vorerfahrung"? ✘ falsch (Absolutaussage).

Merke: „immer", „ausschliesslich", „nie" in Aussagen sind fast immer ein Warnsignal für *falsch*.

Technologie-Stacks (Frage 10)

Stack	Bestandteile	Charakter
MEAN	MongoDB, Express.js, **Angular**, Node.js	Durchgängig JavaScript; für kleine Projekte evtl. überdimensioniert
MERN	MongoDB, Express.js, **React**, Node.js	Modular → Flexibilität & Skalierbarkeit für SPAs / mobile Apps
LAMP	Linux, Apache, MySQL, PHP/Perl/Python	Bewährt, grosse Community
Django	Python-Framework + Postgres	ORM, schnelle Entwicklung, eingebaute Sicherheit

Wahre Aussagen aus der Prüfung:

- MEAN kann für kleine Projekte überdimensioniert sein. ✔
- Django+Postgres bietet ORM für schnelle Entwicklung datenintensiver Apps. ✔
- MERN bietet durch Modularität Flexibilität/Skalierbarkeit für SPA & mobil. ✔
- Django+Postgres kann für **Echtzeit**anwendungen weniger leistungsfähig sein. ✔

Falsche Aussagen:

- MEAN „besonders für Anfänger geeignet" wegen durchgängigem JS. ✘
- LAMP „besonders für Echtzeitanwendungen geeignet". ✘ (LAMP ist nicht speziell echtzeitstark)
- LAMP „weniger kosteneffizient wegen alter Technologie". ✘
- MERN „weniger geeignet für Projekte mit vielen Programmiersprachen". ✘

Merke (Punktverlust): Die einheitliche JS-Sprache von MERN ist ein **Vorteil**, kein Nachteil — die Aussage, MERN sei deshalb „weniger geeignet", ist **falsch**. Und für **Echtzeit** (z. B. Live-Chat, Streaming) ist ein Node.js-Stack mit WebSockets stärker als das klassische request-basierte Django.

Progressive Web Apps — PWA (Fragen 8, 12)

Web-Apps, die sich wie native Apps verhalten. **Kernkomponente: Service Worker** (nicht jQuery/React/SPA — das sind keine PWA-spezifischen Komponenten).

Wahre Eigenschaften:

- **Service Worker** ermöglichen **Offline-Fähigkeit** (Caching) → funktionieren ohne aktive Internetverbindung. ✔
- **HTTPS ist Pflicht** für eine PWA (Sicherheit, Voraussetzung für Service Worker). ✔

Falsche Aussagen:

- PWAs unterstützen **doch** moderne Gerätefunktionen (Push, teils Sensoren) — „keine OS-Funktionen" ist **falsch**. ✘
- PWAs **können** Push-Benachrichtigungen und Installation auf dem Homescreen (über das Web App Manifest) — „bieten keine Push, nicht installierbar" ist **falsch**. ✘

Bausteine einer PWA: **Service Worker** + **Web App Manifest** + **HTTPS**.

---

5. Security & Authentifizierung

HTTPS / TLS

HTTPS = HTTP über TLS-Verschlüsselung. Schützt Vertraulichkeit und Integrität der Übertragung. Voraussetzung für PWAs und Pflicht bei Übertragung von Anmeldedaten/Tokens.

HTTP Basic Authentication (Frage 17)

```
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Der String nach `Basic` ist **Base64** von `benutzername:passwort`. Wahre Aussagen:

- Die Anmeldedaten sind faktisch **im Klartext** sichtbar und **leicht entschlüsselbar** (Base64 ist nur eine Codierung). ✔
- Darf deshalb **nur über HTTPS** verwendet werden. ✔

Falsche Aussagen:

- „Base64 ist eine sichere Verschlüsselung, nicht rückgängig machbar". ✘
- „Header verwendet sichere Base64-Verschlüsselung". ✘

Merke (Kernkonzept): Base64 ist Codierung, keine Verschlüsselung. Jeder kann es in einer Zeile zurückdecodieren. Das gilt auch für JWTs.

OWASP Top 10 (2021) → Präventionsstrategien (Frage 16)

Risiko	Prävention
A01 — Broken Access Control	Zugriffsrechte rollenbasiert definieren und durchsetzen

Risiko	Prävention
A02 — Cryptographic Failures	Starke Verschlüsselung einsetzen (SSL/TLS)
A03 — Injection	Prepared Statements & parametrisierte Abfragen
A04 — Insecure Design	Sicherheit früh in den Designprozess integrieren
A05 — Security Misconfiguration	Konfigurationen regelmässig prüfen und aktualisieren

Eselsbrücke: **Access** → Rollen, **Crypto** → TLS, **Injection** → Prepared Statements, **Design** → früh mitdenken, **Misconfiguration** → regelmässig prüfen.

Cross-Site Scripting — XSS (Frage 18)

Stored XSS: Angreifer reicht einen Kommentar mit `<script>…</script>` ein. Die Anwendung speichert ihn ohne **Validierung/Sanitization** und fügt ihn ungefiltert ins HTML ein. Beim Aufruf der Seite wird das Skript im Browser **jedes Opfers** ausgeführt und sendet dessen Cookies an den Angreifer-Server.

Wirksame Massnahmen:

- **HTTPOnly-Cookies** → JavaScript kann nicht auf das Cookie zugreifen (`document.cookie` sieht es nicht) → Cookie-Diebstahl verhindert. ✔
- **Content Security Policy (CSP)** → erlaubt Skripte nur aus vertrauenswürdigen Quellen → blockiert injizierte Skripte. ✔

Falsche Aussagen:

- „Clientseitige Validierung reicht aus." ✘ — clientseitige Prüfung lässt sich umgehen; entscheidend ist **serverseitige** Validierung + Output-Encoding/Sanitization.
- „Opfer muss aktiv auf den Kommentar klicken." ✘ — bei Stored XSS reicht das **Laden** der Seite, kein Klick nötig.

Grundregel gegen XSS: **Eingaben validieren/sanitizen, Ausgaben encoden**, plus HTTPOnly-Cookies und CSP als zusätzliche Schichten.

JSON Web Token — JWT (Frage 20)

JWT für **stateless** Authentifizierung. Aufbau: **Header.Payload.Signature** (drei Base64-codierte Teile, durch Punkte getrennt).

Wahre Aussagen:

- JWTs sind **selbstenthaltend** und speichern Benutzerinformationen (im Payload/Claims). ✔
- JWTs authentifizieren den Benutzer **über mehrere Anfragen hinweg** (der Token wird bei jeder Anfrage mitgeschickt, kein Server-Sitzungsspeicher nötig → stateless). ✔

Falsche Aussagen:

- „JWTs erfordern keine Validierung, da nicht fälschbar." ✘ — der Server **muss** die Signatur prüfen.
- „JWTs sollten über HTTP ohne SSL/TLS verwendet werden, da selbst verschlüsselt." ✘ — JWTs sind **nicht verschlüsselt**, nur **codiert und signiert**. Wer den Token hat, kann den Payload lesen → **immer über HTTPS** übertragen.

Merke (war die einzige 0-Punkte-Frage): „selbstenthaltend, speichert alle Infos" (a) **und** „authentifiziert über mehrere Anfragen" (d) sind richtig. Der zentrale Denkfehler: **codiert ≠ verschlüsselt**, deshalb HTTPS-Pflicht.

---

6. CMS, Web Vitals & SEO

Headless CMS vs. traditionelles CMS (Frage 21)

Traditionelles CMS (z. B. klassisches WordPress): enge **Kopplung** von Inhalt und Darstellung; Ausgabe primär als Webseite.

Headless CMS: charakterisiert durch die **Trennung von Inhalt und Darstellung**. Inhalte werden über eine API (oft REST/JSON) bereitgestellt und können in **beliebige Kanäle** ausgegeben werden (Web, App, Smart Devices). Das „Frontend" (Head) fehlt — daher *headless*.

Falsche Charakterisierungen: „Beschränkung auf Webseiten als Ausgabekanal", „enge Kopplung", „serverseitige Skripte für die Darstellung".

Core Web Vitals (Frage 19)

Web Vitals sind Googles Metriken für die **Nutzererfahrung — nicht nur** Ladezeit. Die drei Kern-Metriken:

- **LCP** (Largest Contentful Paint) — **Ladeleistung** (wann ist das grösste Element sichtbar).
- **FID/INP** (First Input Delay bzw. Interaction to Next Paint) — **Interaktivität** (Reaktion auf Eingaben).
- **CLS** (Cumulative Layout Shift) — **visuelle Stabilität** (springt das Layout?).

Wahre Aussagen:

- Bessere Web Vitals → **höheres SEO-Ranking** bei Google. ✔
- Schnellere und stabilere Seiten → **niedrigere Absprungraten**. ✔
- Web Vitals messen u. a. die **visuelle Stabilität**. ✔

Falsche Aussage:

Merke (Punktverlust): „Schnellere, stabilere Seiten → niedrigere Absprungrate" ist **wahr** — bessere Performance hält Nutzer auf der Seite.

---

7. Testing

Test-Pyramide (Frage 22)

Drei Schichten, von unten nach oben: **Unit → Service → UI.**

```
      /‾‾‾‾‾‾\   UI-Tests        (wenige, langsam, teuer)
     / Service \ Service-Tests   (mittlere Menge)
    /   Unit    \ Unit-Tests     (viele, schnell, günstig)
```

Wahre Aussagen:

- **Basis** = **Unit-Tests**, die **am häufigsten** durchgeführt werden sollten. ✔
- **Service-Tests** bilden die **mittlere** Schicht — seltener als Unit-, häufiger als UI-Tests. ✔

Falsche Aussagen:

- „UI-Tests sind weniger kostenintensiv als Unit-Tests." ✘ — UI-Tests sind **teurer** und langsamer.
- „Mit zunehmender Höhe steigt die Geschwindigkeit der Tests." ✘ — nach **oben** werden Tests **langsamer**.

Faustregel: **je weiter unten, desto mehr / schneller / billiger**; **je weiter oben, desto weniger / langsamer / teurer**. Tools: Unit (z. B. Jest), End-to-End/UI (z. B. **Playwright, Cypress**); Einbettung in **CI/CD**.

Schnell-Repetitorium (deine Schwachstellen aus der Prüfung)

1. **HTTP 200 / 201 / 204** — OK-mit-Daten / Created / No-Content. Nicht verwechseln.
2. **Microservice-Zuordnung** — Deploybarkeit=reagieren, eigene Technologie=Flexibilität, Kommunikation=weniger Abhängigkeit.
3. **JWT** — selbstenthaltend + über mehrere Anfragen; **codiert ≠ verschlüsselt** → HTTPS Pflicht.
4. `this` — in traditionellen Funktionen vom Aufruf abhängig (nicht „immer global"); Pfeilfunktionen erben `this` lexikalisch.
5. **Data Binding** — One-Way reduziert Komplexität (nicht erhöht); Two-Way spart Codezeilen.

6. **Web Vitals —** schnellere/stabilere Seiten senken die Absprungrate (wahr).
7. **Absolutaussagen** mit „immer/nie/ausschliesslich" sind fast immer *falsch*.