Lokale Modelle, RAG und WebMCP mit JavaScript

KI zieht in den Browser

von Murat Sari, Ines Chargui & Gil Fink

KI zieht in den Browser – und macht aus dem Frontend eine neue AI-Runtime. Dieses Whitepaper zeigt, wie lokale Modelle, RAG und WebMCP moderne JavaScript-Anwendungen verändern. Kompakt, praxisnah und mit konkreten Ansätzen für die Umsetzung.

☞ **Wissen im Tab: RAG im Browser mit Gemma 4**
von Murat Sari

☞ **AI im Browser**
von Ines Chargui

☞ **WebMCP: die Brücke zwischen AI Agents und dem Web?**
von Gil Fink

Retrieval-augmented Generation komplett clientseitig: mit Angular, Transformers.js, WebGPU und Googles neuem On-Device-Modell Gemma 4 E2B

Wissen im Tab: RAG im Browser mit Gemma 4

von Murat Sari

Cloudbasierte Sprachmodelle sind leistungsfähig, verursachen aber laufende Kosten, verarbeiten Daten außerhalb der eigenen Infrastruktur und haben keinen Zugriff auf interne Wissensbestände. Es geht aber anders: Mit Gemma 4 läuft ein Instruction-tuned On-Device-Modell direkt im Browser – ohne Server, ohne API-Schlüssel und ohne dass Fragen oder Dokumentinhalte an einen Modellserver gesendet werden müssen. In Kombination mit Retrieval-augmented Generation beantwortet es Fragen zu den eigenen Dokumenten und liefert auch Belege.

Die meisten KI-Anwendungen funktionieren nach demselben Muster: Eine App schickt eine Anfrage an einen Cloud-Dienst. Dort rechnet ein großes Modell, und die Antwort kommt zurück. Das ist zwar bequem, hat aber mehrere Nachteile. Erstens kostet jede Anfrage Geld. Zweitens verlassen womöglich sensible Daten das eigene Haus. Und drittens kennt das Modell weder firmeninterne Inhalte noch irgendetwas, was nach seinem Trainingsstichtag passiert ist.

Seit einiger Zeit gibt es ernstzunehmende Alternativen. Moderne, kompakte Modelle laufen direkt auf dem Endgerät: im Browser, über die Grafikkarte (WebGPU). Der Download geschieht einmalig (cachefähig), danach läuft die Inferenz lokal im Browser. Es fallen keine laufenden Kosten an, maximale Datensparsamkeit ist gewährleistet. Was bislang fehlte, war ein Modell, das klein genug für den Browser und gleichzeitig gut genug für echte Aufgaben ist. Gemma 4 schließt diese Lücke.

Das zweite Puzzlestück heißt Retrieval-augmented Generation (RAG) [1]. Statt das Modell raten zu lassen, werden zur Laufzeit passende Textstellen aus einer eigenen Wissensbasis herausgesucht und dem Modell als Kontext vorgelegt. Das Modell antwortet dann auf Basis dieser Belege: Die Antworten sind aktuell, nachvollziehbar und deutlich seltener frei erfunden. Beides zusammen, lokales Modell und lokales Wissen, ergibt einen Assistenten, der vollständig im Browser läuft.

Gemma 4: ein Sprachmodell fürs Endgerät

Gemma 4 ist eine am 2. April 2026 veröffentlichte, offene Modellfamilie von Google DeepMind [2], die unter der Apache-2.0-Lizenz steht (Kasten: „Die Gemma-4-Familie auf einen Blick“). Sie umfasst fünf Größen, vom winzigen E2B- bis zum 31-Milliarden-Modell. Für den Browser ist die kleinste Variante interessant: Gemma 4 E2B mit rund 2,3 Milliarden effektiven Parametern, einem Kontextfenster von 128 000 Tokens und der Fähigkeit, neben Text auch Bilder und Audio zu verarbeiten. Das „E“ steht für „effective“ bzw. „edge“: Dank einer Technik namens Per-Layer-Embeddings (PLE) ist die Zahl der zur Laufzeit aktiven Parameter kleiner als die Gesamtzahl, wodurch Speicher gespart wird [3]. Wie alle modernen Sprachmodelle basiert Gemma auf der Transformer-Architektur [4]. E2B läuft je nach Quantisierung, Backend und Kontextbudget auf Geräten mit mehreren GB verfügbarem Speicher.

Für RAG ist gerade das kleine Modell gut geeignet, denn hier zählt nicht enzyklopädisches Weltwissen, sondern das Verstehen des mitgelieferten Kontexts. Zwei Neuerungen von Gemma 4 helfen dabei konkret. Zum einen unterstützt die Familie erstmals nativ die Systemrolle: Die Anweisung

„Antworte ausschließlich auf Basis des Kontexts“ lässt sich sauber vom eigentlichen Nutzertext trennen. Zum anderen gibt es einen abschaltbaren Thinking-Modus. Für faktentreue, gut belegte Antworten lassen wir ihn deaktiviert.

Für RAG ist genau das entscheidend: Bei präzisen Details wie einer OpenSSL-Versionsnummer, einer CVE-Kennung oder einem Datum neigen selbst gute Modelle zum Halluzinieren, weil sie sich nur ungefähr erinnern. Legt man dem Modell die Textstelle mit dem Beleg vor, wird aus dem vagen Erinnern ein nachprüfbares Zitat. Genau hier zeigt RAG seinen Wert: Das Wissen stammt aus unserem Korpus und nicht aus dem Gedächtnis des Modells.

Die Gemma-4-Familie auf einen Blick

- *E2B:* ~2,3 Mrd. effektive Parameter, 128K Kontext, Text/Bild/Audio; für Smartphones, Laptops und den Browser
- *E4B:* ~4,5 Mrd. effektive Parameter, 128K Kontext, Text/Bild/Audio; für Laptops ab ~8 GB RAM
- 12B (Unified): ~12 Mrd. Parameter, 256K Kontext, Text/Bild/Audio (encoder-frei); für Laptops und Server.
- *26B A4B:* Mixture-of-Experts, 25,2 Mrd. gesamt / 3,8 Mrd. aktiv, 256K Kontext; schnell wie ein 4-Mrd.-Modell
- *31B:* dichtes Modell, 30,7 Mrd. Parameter, 256K Kontext; für Workstations und Server
- *Alle Modelle:* Apache-2.0-Lizenz, über 140 Sprachen, natives Function Calling und konfigurierbarer Thinking-Modus; alle Größen auf Hugging Face [5].

Warum es Gemma 4 gibt und wer dafür bezahlt

Ein Modell dieser Qualität zu trainieren, kostet wahrscheinlich mehrere Millionen – Google DeepMind veröffentlicht dazu keine konkreten Zahlen. Warum verschenkt Google es dann? Gemma basiert auf derselben Forschung und Technologie wie die geschlossenen Gemini-Modelle [2] und Google verfolgt damit eine bewusste Doppelstrategie: Die Gemini-Modelle bleiben proprietär und werden über API und Produkte vermarktet, während die offene Gemma-Familie Entwicklerinnen und Entwickler an das eigene Ökosystem bindet. Dieses Kalkül ähnelt dem von Red Hat mit Linux: Den Kern verschenken und an Support, Hosting und Integration verdienen. Wer mit Gemma prototypisiert, nutzt für den produktiven Betrieb häufig Google Clouds Vertex AI, wodurch das Geld zurückfließt.

Die Entwicklung wird also von Google DeepMind bezahlt und nicht über Lizenzgebühren refinanziert (die Modelle stehen erstmals unter der freien Apache-2.0-Lizenz [2]). Die Refinanzierung erfolgt über Cloud-Umsätze, den Verkauf von Rechenzeit (TPUs) und die Strahlkraft auf das übrige Portfolio. Die Gemma-Familie verzeichnet über 400 Millionen Downloads und dieselbe kompakte Technik, die E2B im Browser laufen lässt, kommt auch bei On-Device-Funktionen in Android zum Einsatz. Für uns als Entwickler:innen zählt vor allem das Ergebnis: ein einsatzfähiges Modell ohne Nutzungsgebühren unter der kommerziell freizügigen Apache-2.0-Lizenz.

Unter der Haube: Mixture-of-Experts und spekulatives Decoding

Zwei Architekturideen der Familie lohnen einen genaueren Blick, auch wenn im Browser das kleine E2B arbeitet. Die erste ist Mixture-of-Experts (MoE), sichtbar am Modell 26B A4B. Statt für jedes Token das gesamte Netz zu berechnen, enthält jede Schicht hier viele spezialisierte „Experten“, und ein kleines Gating-Netz wählt pro Token nur eine Handvoll davon aus. Bei Gemma 4 sind es acht von 128 Experten, plus ein stets aktiver gemeinsamer Experte. So sind pro Rechenschritt nur rund 3,8 von 25,2 Milliarden Parametern aktiv [2]. Das Ergebnis ist ein Modell, das fast so schnell rechnet wie ein 4-Milliarden-Modell, aber wie ein viel größeres Modell antwortet.

Die zweite Idee ist das spekulative Decoding, das Gemma 4 als Multi-Token Prediction (MTP) umsetzt [6]. Normalerweise erzeugt ein Sprachmodell ein Token nach dem anderen. Beim spekulativen Decoding schlägt ein kleines, schnelles „Draft-Modell“ hingegen mehrere Token auf einmal vor. Das große Zielmodell prüft diese dann in einem einzigen Durchgang. Passt ein Vorschlag nicht, liefert das Zielmodell selbst das korrekte Token und das Draft-Modell macht von dort weiter. Gemma 4 bringt zu jedem Modell ein eigenes Draft-Modell mit, das sich die Embedding-Tabelle und die letzten Aktivierungen mit dem Zielmodell teilt. Dadurch ist die Antwort bit-genau dieselbe wie ohne Beschleunigung, nur deutlich schneller [6]. Gerade für latenzkritische und On-Device-Szenarien ist das sehr wertvoll. Ein Detail am Rande: Bei MoE-Modellen wie 26B A4B kann das Prüfen der Vorschläge zusätzliche Expertengewichte aus dem Speicher nachladen, sodass der Gewinn bei Batchgröße 1 schrumpft [6]. In unserem Browserbeispiel bleiben wir bei der Standardgenerierung ohne MoE/MTP.

Die Architektur des Beispiels

Das Beispielprojekt ist in zwei Hälften unterteilt, die zu unterschiedlichen Zeitpunkten laufen (**Abb. 1**). Die Offline-Pipeline wird einmalig über die Kommandozeile mit Node.js ausgeführt. Dabei liest sie einen Korpus aus Markdown-Dateien ein, zerlegt ihn in Abschnitte, berechnet für jeden Abschnitt einen Vektor und schreibt alles in eine kompakte JSON-Datei. Die Onlinehälfte ist eine Angular-Anwendung. Diese lädt die Datei einmal und erledigt danach alles Weitere im Browser: Sie bettet die Frage ein, sucht die ähnlichsten Abschnitte, baut daraus einen Kontext und lässt Gemma 4 eine Antwort formulieren.

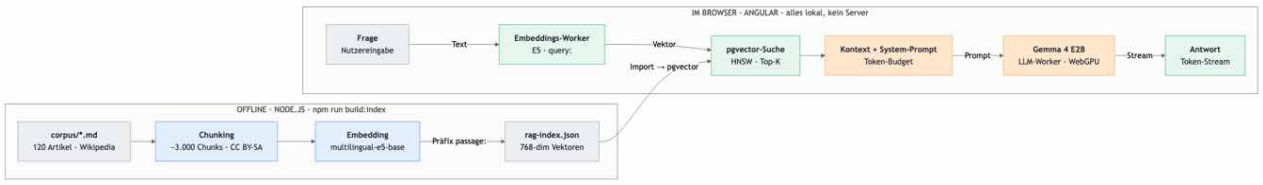


ABB. 1 Die zwei Hälften der Anwendung: Offline entsteht aus Markdown ein Vektorindex, online laufen Einbettung, Suche und Generierung vollständig im Browser

Bewusst haben wir auf alles verzichtet, was ein größeres System an zusätzlichen Funktionalitäten hätte, wie hybride Suche (BM25) und Reranking (Crossencoder). Eingebaut haben wir dagegen ein Query-Rewriting, das Folgefragen auflöst, sowie ein Gedächtnis für den Gesprächsverlauf. Beim Retrieval setzen wir bewusst auf dieselbe Technik wie im Backend: pgLite als Vektordatenbank (PostgreSQL mit der Erweiterung pgvector), nur dass die Datenbank dank WebAssembly im Browser läuft. So bleibt der Kern von RAG sichtbar und der Schritt in die Produktion ist klein.

Damit das Userinterface flüssig bleibt, werden die Vektordatenbank und beide Modelle (Embedder und Sprachmodell) in jeweils eigenen Web Workern ausgeführt, die vom UI-Thread getrennt sind. Der Rest ist eine einzige Angular-SPA mit mehreren Views: ein Chat-Fenster mit aufklappbarem Gedankengang, ein Tokenzähler und ein Kontextinspektor.

Abbildung 2 zeigt den Datenfluss im Browser. Der *RagService* orchestriert vier Dienste und drei Web-Worker (Vektordatenbank, Embedder und Gemma 4); den vorberechneten Index lädt der DB-Worker. Das UI bleibt schlank und reagiert nur auf Signale.

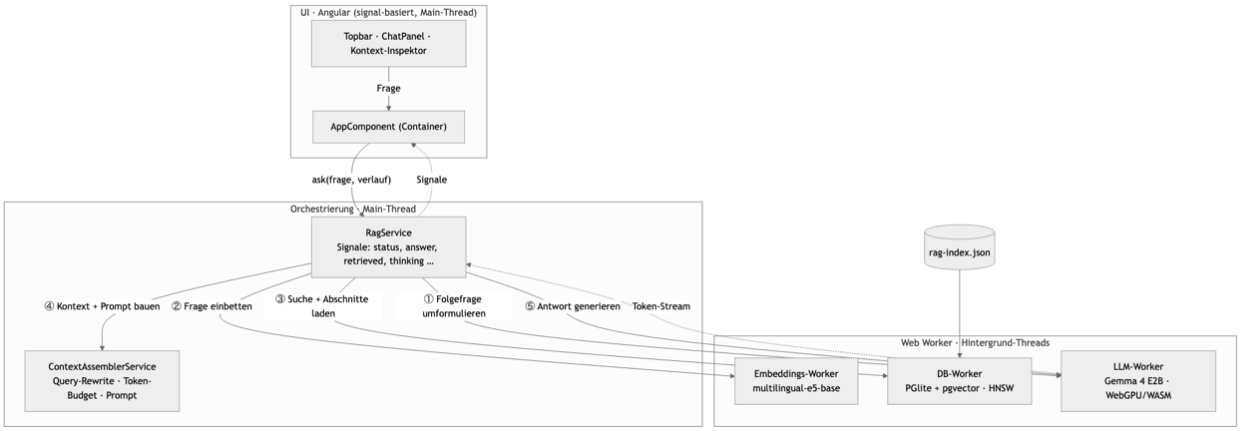


ABB. 2 Datenfluss im Browser

Schritt 1: Aus Dokumenten wird ein Vektorindex

Bevor mit der Suche begonnen werden kann, muss das Wissen aufbereitet werden. Als Demokorpus dient ein deutschsprachiges Informatikwissensarchiv bestehend aus rund 120 Wikipedia-Artikeln (CC BY-SA 4.0) [7] zu berühmten Softwarebugs und Sicherheitsvorfällen (Heartbleed, Ariane 5, Stuxnet …), zu Programmiersprachen, Algorithmen und Datenstrukturen, zu Kryptografie und Netzwerkprotokollen sowie zu Pionierinnen und Pionieren der Informatik. Solche Inhalte sind gehaltvoll und ein guter RAG-Prüfstein, da sie voller präziser Fakten wie Versionsnummern, CVE-Kennungen, Komplexitäten und Jahreszahlen stecken. Bei diesen Fakten liegt ein Sprachmodell aus dem Gedächtnis gern daneben.

Der erste Schritt ist das Chunking: Lange Artikel werden in kleinere Abschnitte zerlegt, damit später nur die relevanten Teile in den Prompt gelangen. Wir nutzen überschriftenbasiertes Chunking: Jeder Abschnitt unter einer Überschrift wird zu einem Chunk, und der Überschriftenpfad (etwa „SQL-Injection > Gegenmaßnahme: Prepared Statements“) wandert als Herkunftsangabe mit. Zu lange Abschnitte werden satzweise auf ein Zeichenbudget aufgeteilt. So entstehen aus den 120 Artikeln rund 3 000 Chunks.

Anschließend wird jeder Chunk in einen Vektor, ein sogenanntes Embedding, übersetzt. Das Modell ist so trainiert, dass Texte mit ähnlicher Bedeutung auch dann nahe beieinander im Vektorraum liegen, wenn sie andere Wörter verwenden. Genau das macht die spätere semantische Suche möglich. Die Idee, Fragen und Passagen über dichte Vektoren statt über Stichwörter zu matchen, geht auf „Dense Passage Retrieval“ zurück [8]. Als Einbettungsmodell wählen wir *multilingual-e5-base*, ein nach dem E5-

Rezept [9] trainiertes, mehrsprachiges Modell [10]: Deutsche Fragen finden deutsche Passagen. Mit 768 Dimensionen ist dieses Modell etwas größer als die *small*-Variante, dafür aber treffsicherer und klein genug, um auch im Browser nahezu verzögerungsfrei zu laufen.

Zwei Details entscheiden über die Qualität. Erstens erwarten die E5-Modelle ein Präfix: Passagen werden mit *passage:* eingeleitet, Fragen später mit *query:*. Zweitens normalisieren wir die Vektoren auf die Länge *1* – das ist der Standard für die Kosinusähnlichkeit, mit der pgvector später arbeitet. Listing 1 zeigt den Kern der Pipeline. Die Offline-Pipeline bettet alle Chunks in Batches ein und schreibt einen kompakten Index (hier rund 3 000 Chunks, rund 24 MB).

Listing 1: Offline-Pipeline

```
// pipeline/embedding.ts: Embedder auf Basis von multilingual E5
export async function createEmbedder() {
  const extract = await pipeline('feature-extraction',
                                              'Xenova/multilingual-e5-base');
  return async (texts, prefix) => {
    // mean-Pooling + Normalisierung => einheitslange Vektoren
    const output = await extract(texts.map((t) => prefix + t),
                                         { pooling: 'mean', normalize: true });
    return output.tolist();
  };
}

// pipeline/build-index.ts: Korpus chunken, einbetten und als JSON schreiben
const chunks = loadChunks(CORPUS_DIR);
// Markdown -> Chunks (Überschriften)
const embed  = await createEmbedder();
const vectors = [];
for (let i = 0; i < chunks.length; i += 64) {
  // schonend in Batches
  const vecs = await embed(chunks.slice(i, i + 64).map((c) =>
                                                 c.content), 'passage: ');
  vectors.push(...vecs);
}
writeFileSync(OUTPUT, JSON.stringify({
  embeddingModel: 'Xenova/multilingual-e5-base', dim: 768,
  chunks: chunks.map((c, i) => ({ ...c, embedding: vectors[i] })),
}));
```

Das Ergebnis ist eine einzige Datei, *rag-index.json*. Sie enthält für jeden Chunk den Text, den Überschriftenpfad, den Link zum entsprechenden Wikipedia-Artikel, eine grobe Tokenschätzung und den 768-dimensionalen Vektor. Der Link sorgt für echtes Grounding: Jede Aussage lässt sich bis zur Quelle zurückverfolgen. Diese Datei dient als Importquelle für unsere Vektordatenbank im Browser (Schritt 2).

Schritt 2: Vektorsuche im Browser

Beim Start importiert die App den Index einmalig in eine echte Vektordatenbank im Browser: PostgreSQL mit der Erweiterung pgvector [11], ausgeführt über PGlite [12], nach WebAssembly kompiliertes PostgreSQL. Die Vektoren liegen in einer Tabellenspalte vom Typ *vector(768)*. Ein HNSW-Index [13] (Hierarchical Navigable Small World) beschleunigt die Suche, und die Daten werden in *IndexedDB* persistiert, sodass der Import nur beim ersten Besuch nötig ist.

Wenn eine Frage gestellt wird, wird sie mit demselben E5-Modell eingebettet (Präfix *query:*, im Web-Worker). Die eigentliche Suche ist dann ein normales SQL-Statement mit pgvectors Kosinusoperator <=>, also genau das, was man auch auf einem Server schreiben würde, nur dass die Datenbank lokal im Tab läuft. pgvector findet also mit Hilfe des <=>-Operators und eines HNSW-Index die ähnlichsten Chunks (Listing 2).

JavaScript Track

JavaScript & Angular Days

JavaScript 2026: Moderner entwickeln, sauberer strukturieren!

Starte mit einem JavaScript-Crashkurs und entdecke Trend-Themen für Einsteiger:innen, Umsteiger:innen und Fortgeschrittene. Nutze neue Syntax, Toolchains & Browser-APIs für saubere, performante Projekte. Teste automatisiert mit klaren Patterns. Strukturiere smart mit JSON-Modulen & nativen Tools. Jetzt Wissen tanken, das echten Fortschritt bringt.

Lerne von führenden Expert:innen:

- **JavaScript 2026:** Neue Syntax, Toolchains & Browser-APIs einsetzen
- **Automatisiertes Testing:** Komponenten mit klaren Patterns absichern
- **Projektstruktur:** JSON-Module & native Tools sauber strukturieren
- **Komponentenarchitektur:** Provider & Compound Patterns gezielt anwenden
- **Async & Datenflüsse:** Streams, Iterators & immutable Arrays nutzen

Listing 2: Retrieval als SQL-Abfrage

```
// src/app/workers/db.worker.ts: Nächste-Nachbarn-Suche per pgvector
async function search(queryVec: number[], topK: number):Promise<Retrieved[]> {
  const emb = ,[${queryVec.join(',')}]‘;
  const { rows } = await db.query(
    ,SELECT id, content, section_path, url, token_count, 1 -
                                    (embedding <=> $1::vector) AS similarity
      FROM chunks
      ORDER BY embedding <=> $1::vector   -- Kosinus-Distanz, HNSW-Index
      LIMIT $2‘,
    [emb, topK],
  );
  return rows.map((r) => ({ chunk: { /* … */ }, score: Number(r.similarity) }));
}
```

Der Charme dabei ist, dass es sich um dieselbe Technik wie im Backend (PostgreSQL + pgvector) handelt – nur ohne Server. Wer den Index vom Prototyp in die Produktion überführen möchte, kann denselben SQL-Code gegen eine echte Postgres-Instanz laufen lassen. Für unsere ~3 000 Chunks liefert die Suche Ergebnisse in wenigen Millisekunden.

Schritt 3: Das Kontextfenster im Blick

Die Toptreffer wandern nun in den Prompt, allerdings nicht alle. Da das Kontextfenster eines Modells begrenzt ist, ist es selbst bei Gemma 4 E2B mit 128 000 Tokens klug, pro Anfrage nur ein knappes Budget zu vergeben. Das spart Rechenzeit und hält das Modell fokussiert. Zu viel Kontext kann sogar schaden. Modelle nutzen Informationen in der Mitte langer Eingaben nachweislich schlechter („Lost in the Middle“-Phänomen [14]). Aus den besten Treffern laden wir jeweils den gesamten zugehörigen Abschnitt – nicht nur den einzelnen Chunk – und füllen damit ein festes Budget, bis es erschöpft ist. System-Prompt und Frage zählen ebenfalls mit.

Genau das macht der Kontextinspektor der Anwendung sichtbar. Ein gestapelter Balken zeigt, wie sich das Budget auf System-Prompt, Gesprächsverlauf, Frage, geladene Abschnitte und freien Platz verteilt; ein Tokenzähler nennt die belegte Zahl. Daneben listet der Inspektor die gefundenen Chunks samt Ähnlichkeitswert, Herkunftspfad und einem direkten Link zum entsprechenden Wikipedia-Artikel auf. Man sieht also nicht nur die Antwort, sondern kann jede Aussage bis zur Quelle nachschlagen. Dieses „Aufklappen“ der Blackbox ist didaktisch wertvoll: Es macht greifbar, dass meist nur eine Handvoll Chunks ins Budget passt und dass die Auswahl maßgeblich über die Antwortqualität entscheidet.

Für die Schätzung der Tokenanzahl genügt folgende Faustregel: Im Deutschen entspricht ein Token ungefähr vier Zeichen. Das ist zwar nicht exakt, aber schnell und für eine Budgetanzeige völlig ausreichend.

Abbildung 3 zeigt die Anwendung im Einsatz (hier mit der Frage „Wie schützt man sich vor SQL-Injection?“). Links ist die geerdete Antwort mit Quellenangabe zu sehen, rechts der Kontextinspektor mit Tokenbudget und den abgerufenen Chunks samt Ähnlichkeitswert und direktem Link zum Wikipedia-Artikel.

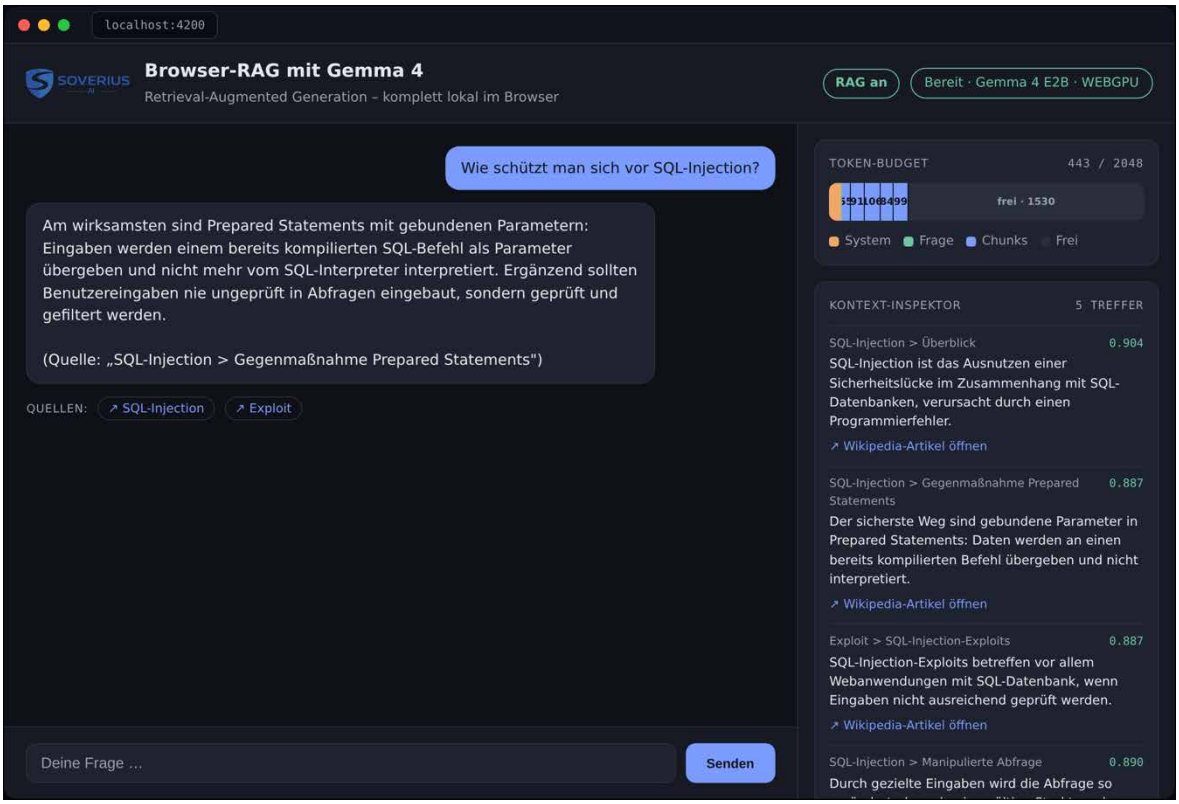


ABB. 3 Die Anwendung im Einsatz

Schritt 4: Gemma 4 antwortet

Jetzt kommt das Sprachmodell ins Spiel. Geladen wird Gemma 4 E2B über Transformers.js [15], die JavaScript-Bibliothek von Hugging Face, die ONNX-Modelle über die ONNX Runtime im Browser ausführt. Die Gewichte stammen aus dem Repository [16]. Wir laden sie bevorzugt im Format *q4f16* (4-Bit-Gewichte, fp16-Aktivierungen) auf WebGPU [17] und greifen auf das langsamere, aber überall verfügbare WASM-Backend zurück, falls keine Grafikkarte angesprochen werden kann (Listing 3). Dieses Muster macht die App auf jedem Gerät lauffähig.

Listing 3: Erst WebGPU, dann WASM als Auffangnetz

```
// src/app/workers/llm.worker.ts: Modell laden mit WebGPU und WASM-Fallback
try {
  model = await Gemma4ForConditionalGeneration.from_pretrained(MODEL_ID,
                    { dtype: 'q4f16', device: 'webgpu', progress_callback });
} catch {
  model = await Gemma4ForConditionalGeneration.from_pretrained(MODEL_ID,
                         { dtype: 'q4', device: 'wasm', progress_callback });
}
```

Für die eigentliche Antwort erstellen wir Nachrichten mit den Rollen *system* und *user*. Der System-Prompt weist Gemma 4 an, ausschließlich den bereitgestellten Kontext zu verwenden, die relevanten Abschnitte zu nennen und stets die beste daraus ableitbare Antwort zu geben. Das *apply_chat_template* des Tokenizers bringt alles ins richtige Format, *enable_thinking: false* schaltet die Gedankenkette ab und ein *TextStreamer* liefert die Antwort Token für Token zurück, sodass sie live in der Oberfläche wächst (Listing 4). Mit *do_sample: false* dekodieren wir deterministisch: Dieselbe Frage liefert bei gleichem Kontext dieselbe Antwort, was bei einem faktentreuen Assistenten erwünscht ist.

Listing 4: Native Systemrolle, abgeschalteter Thinking-Modus und Token-Streaming

```
// src/app/workers/llm.worker.ts: geerdete Antwort streamen
const inputs = tokenizer.apply_chat_template(
  [{ role: 'system', content: system }, ...history,
                                        { role: 'user', content: prompt }],
  { add_generation_prompt: true, return_dict: true, enable_thinking: false });

const streamer = new TextStreamer(tokenizer, {
  skip_prompt: true, skip_special_tokens: true,
  callback_function: (token) => postMessage({ type: 'token', token }),
});

await model.generate({ ...inputs, max_new_tokens: 2048, do_sample:
                                                          false, streamer });
```

Das Ergebnis: Man gibt eine Frage ein, erkennt im Inspektor sofort, welche Chunks gezogen wurden, und sieht die Antwort Wort für Wort entstehen. Das Ganze läuft lokal ab, nachdem App, Index und Modell geladen sind – ohne Aufruf an einen Modellserver.

Qualität messen, bevor das Modell rät

Ein RAG-System ist nur so gut wie sein Retrieval. Wenn die Suche die falschen Stellen findet, hilft auch das beste Modell nicht. Deshalb lohnt es sich, das Retrieval getrennt vom Sprachmodell zu prüfen: Das geht schnell, ist günstig und das große Modell muss dafür nicht einmal geladen werden. Das Maß dafür heißt *Recall@k*. Für jede Testfrage wird definiert, welche Chunks relevant sind, und gemessen, ob ein relevanter Treffer unter den ersten *k* Ergebnissen auftaucht.

Genau das tut unser Skript *verify-recall.ts* für 22 deutsche Testfragen, die sich über den gesamten Korpus verteilen. Über alle ~3 000 Chunks erreicht die reine Vektorsuche einen *Recall@1* von 100 Prozent (22 von 22 Testfragen): Für jede Frage steht ein relevanter Chunk an erster Stelle. Bemerkenswert ist, wie gut auch umformulierte Fragen greifen: Die Frage „Warum heißt JavaScript so, obwohl es nichts mit Java zu tun hat?“ landet zielsicher beim richtigen Abschnitt, obwohl die Stichwörter nicht exakt übereinstimmen. Das ist der eigentliche Mehrwert der semantischen Suche gegenüber der reinen Stichwortsuche.

Dass hier ein so glattes Ergebnis herauskommt, liegt auch am Korpus: Die 120 Artikel decken klar abgegrenzte Themen ab, sodass die richtige Quelle meist leicht zu finden ist. Bei eng verwandten Inhalten oder wenn es auf eine exakte Zeichenfolge wie eine CVE-Nummer oder einen Produktnamen ankommt, sinkt die Trefferquote. Dann werden Techniken wie hybride Suche (Vektor + Stichwort BM25) oder das Reranking wichtig. Genau deshalb ist diese modellfreie Messung so nützlich: Sie deckt solche Schwächen auf, lange bevor das Sprachmodell ins Spiel kommt.

Der Gegentest: dieselbe Frage ohne RAG

Wie viel trägt das Retrieval wirklich bei? Die App hat einen Schalter dafür: Stellt man RAG aus, überspringt sie die Suche komplett und schickt nur die nackte Frage an Gemma 4, ohne Kontext und ohne Quellen. Bei einer allgemeinen Frage wie „Wie schützt man sich vor SQL-Injection?“ liefert das Modell dann zwar eine plausibel klingende Antwort aus seinem Trainingswissen, aber ohne jeden Beleg. Der Kontextinspektor bleibt dann leer.

Drastischer wird der Unterschied, sobald es um Spezifisches geht, beispielsweise um eine konkrete CVE-Nummer, eine genaue Versionsangabe („Welche OpenSSL-Versionen betraf Heartbleed?“) oder ein Detail aus einem eigenen, womöglich erst nach dem Trainingsstichtag entstandenen Bestand. Ohne RAG rät das Modell oder gibt eine auf den ersten Blick plausible Antwort; mit RAG wird die Textstelle als Beleg samt Link daneben angezeigt. Genau diesen Mehrwert macht der Negativtest in **Abbildung 4** sichtbar.

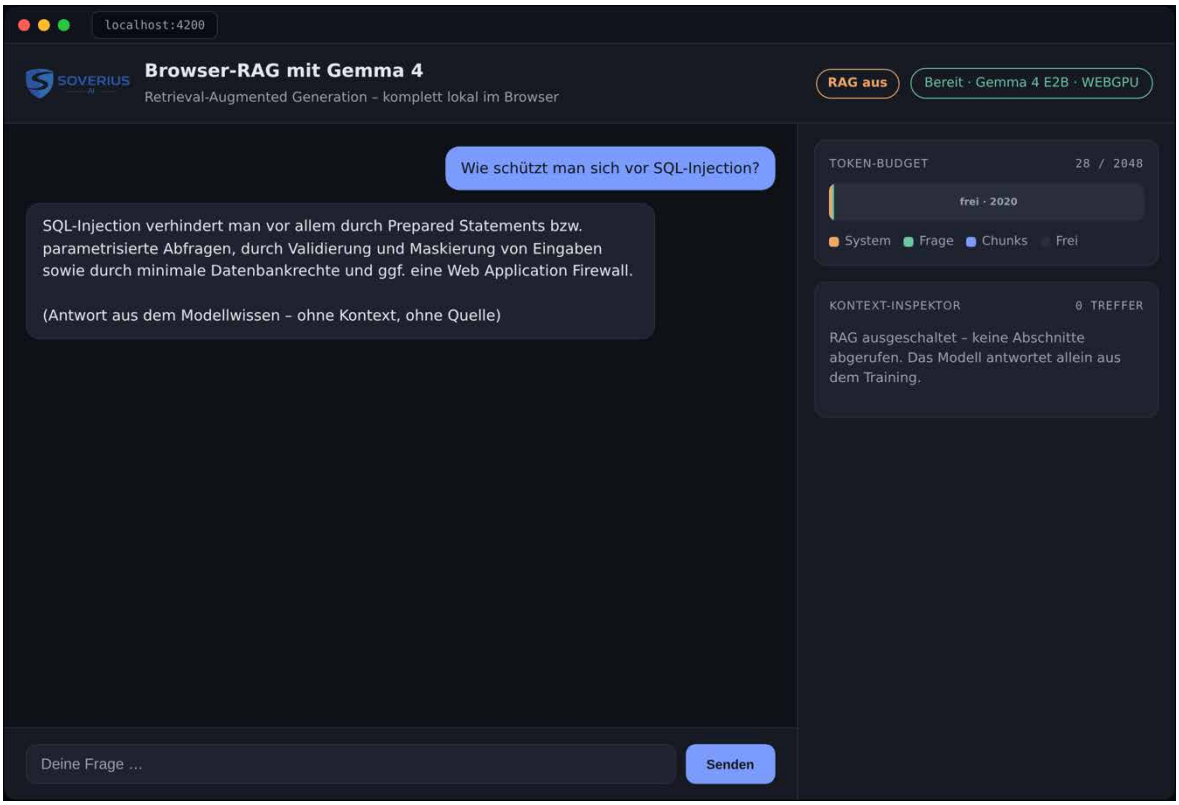


ABB. 4 Negativtest mit ausgeschaltetem RAG

Grenzen und Ausblick

Das Beispiel ist bewusst minimal gehalten. An seinen Lücken sieht man gut, wie es zu einem produktionsreifen System weiterentwickelt werden kann. Die naheliegendste Verbesserung ist die hybride Suche: Durch die Kombination der semantischen Vektorsuche mit dem klassischen BM25-Algorithmus fängt man auch exakte Zeichenfolgen wie Fehlercodes oder API-Namen zuverlässig ein. Ein zweiter Ansatz ist Reranking: Ein kleines Modell bewertet die Top-Kandidaten noch einmal genauer und sorgt so dafür, dass auch Fälle abgedeckt werden, in denen der beste Chunk nicht ganz oben landet. Beide Verfahren sind bekannt, gut dokumentiert und ließen sich an unsere Pipeline andocken.

Auch betrieblich gibt es Stellschrauben. WebGPU ist inzwischen in den Browsern Chrome, Edge, Firefox und Safari angekommen, auf älteren Geräten und unter Linux oder Android jedoch nicht garantiert. Der WASM-Fallback ist daher Pflicht. Die größte Hürde bleibt der einmalige Modelldownload mit einer Größe von rund drei Gigabyte. Klare Fortschrittsanzeigen und ein durchdachtes Caching entscheiden über den ersten Eindruck. **Abbildung 5** veranschaulicht das: Ein klarer Statushinweis zeigt den einmaligen Download von Gemma 4 E2B an, während die Oberfläche und der Inspektor bereits sichtbar sind.

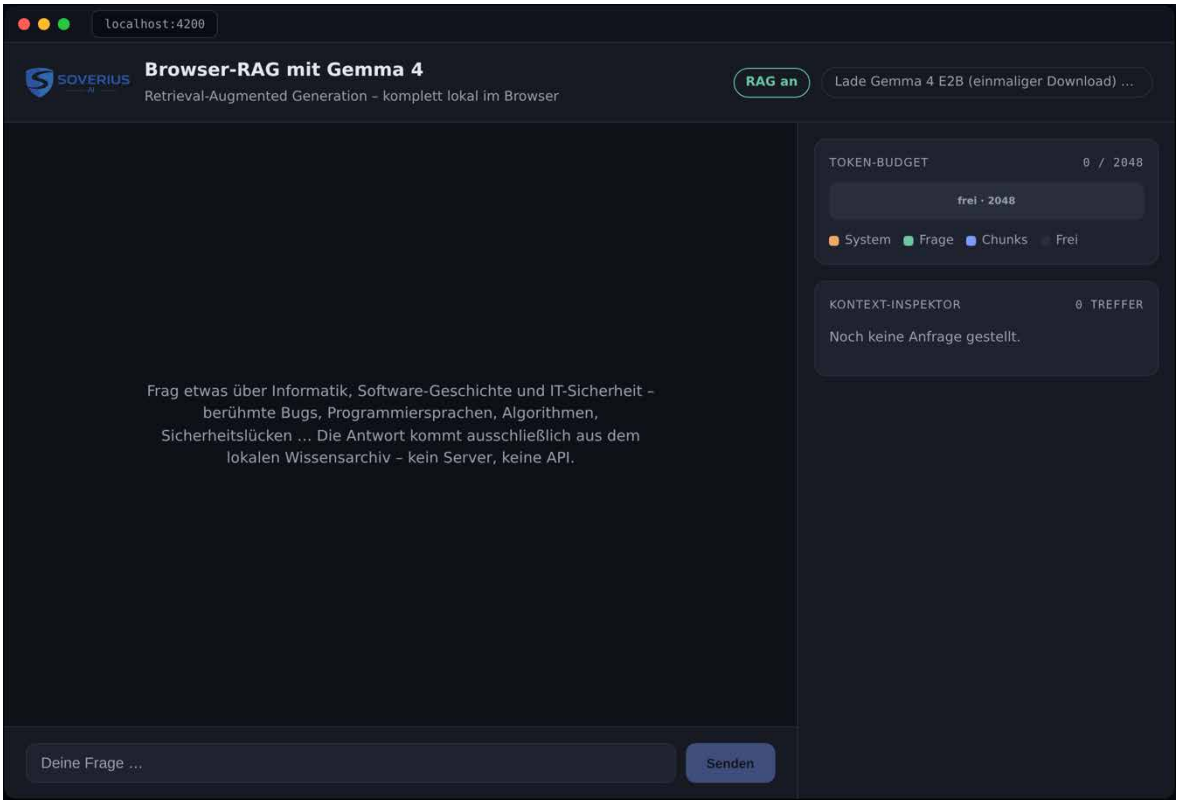


ABB. 5 Schon der erste Start zählt für die UX

Und schließlich ist E2B ein kleines Modell. Für besonders schwierige Fragen kann ein hybrider Ansatz sinnvoll sein, bei dem einfache Anfragen lokal beantwortet werden und nur die schwierigsten an ein größeres Modell weitergereicht werden.

Fazit

Ein Sprachmodell im Browser, das die eigenen Dokumente kennt, lässt sich mit Gemma 4 E2B, Transformers.js und WebGPU in einer überschaubaren Angular-App umsetzen. RAG sorgt dafür, dass die Antworten geerdet,

aktuell und nachvollziehbar bleiben. Die lokale Ausführung gewährleistet, dass keine Frage- und Dokumentendaten an einen Modellserver übermittelt werden und keine Nutzungskosten entstehen. Wer das Prinzip anhand dieses kleinen Beispiels einmal verstanden hat, kann es in jede Richtung ausbauen. Der beste Einstieg ist, den Korpus durch die eigenen Dokumente zu ersetzen, *npm run build:index* aufzurufen und das Modell im eigenen Tab antworten zu lassen. Die fertige Demo läuft unter [18], der Quellcode findet sich unter [19].



Murat Sari ist Softwareentwickler mit Schwerpunkt Angular und KI.

Links & Literatur

[1] https://arxiv.org/abs/2005.11401
[2] https://blog.google/innovation-and-ai/technology/developers-tools/gemma-4/ sowie https://ai.google.dev/gemma/docs/core
[3] https://arxiv.org/abs/2503.19786
[4] https://arxiv.org/abs/1706.03762
[5] https://huggingface.co/google
[6] https://ai.google.dev/gemma/docs/mtp/overview
[7] https://de.wikipedia.org
[8] https://arxiv.org/abs/2004.04906
[9] https://arxiv.org/abs/2212.03533
[10] https://arxiv.org/abs/2402.05672
[11] https://github.com/pgvector/pgvector
[12] https://pglite.dev
[13] https://arxiv.org/abs/1603.09320
[14] https://arxiv.org/abs/2307.03172
[15] https://huggingface.co/docs/transformers.js
[16] https://huggingface.co/onnx-community/gemma-4-E2B-it-ONNX
[17] https://www.w3.org/TR/webgpu/
[18] https://soverius-ai-rag-basic.web.app
[19] https://github.com/Soverius-AI/ai-rag-basic

Wie lokale Modelle die Webarchitektur verändern

AI im Browser

von Ines Chargui

AI verändert nicht nur die Art, wie Software entwickelt wird, sondern zunehmend auch die Architektur moderner Webanwendungen. Welche Aufgaben künftig auf dem Endgerät, im Frontend oder auf dem Server ausgeführt werden, wird neu verhandelt. Christian Liebel und Sebastian Springer erläutern, welche technologischen Entwicklungen hinter diesem Wandel stehen – und welche Folgen er für JavaScript, Fullstack-Entwicklung und moderne Webarchitekturen hat.

AI verändert die Webentwicklung derzeit in hohem Tempo. Doch welche Entwicklungen werden langfristig relevant sein und welche Konsequenzen ergeben sich daraus für Entwickler:innen? Darüber haben wir mit Christian Liebel und Sebastian Springer gesprochen. Beide sind bei den JavaScript & Angular Days als Trainer und fachliche Advisors aktiv und befassen sich seit Jahren mit modernen Webarchitekturen, AI und Fullstack-JavaScript.

Wie AI die Arbeit von Entwickler:innen verändert

Technisch betrachtet hat sich die Disziplin nicht grundlegend verändert: Es geht weiterhin um HTML, JavaScript und CSS. Verändert hat sich jedoch die Art und Weise, wie Software entsteht.

Wer bislang mit Frameworks wie Angular oder React arbeiten wollte, benötigte meist tiefgehendes Fachwissen. AI hat diese Hürde deutlich gesenkt, da Wissen über Frameworks inzwischen Teil des Wissens vieler Modelle ist. Grundlegendes Software-Engineering-Know-how in Kombination mit AI-Unterstützung ermöglicht es Entwickler:innen, mit Sprachen und Frameworks zu arbeiten, in denen sie keine ausgewiesenen Expert:innen sind.

Paradoxerweise steigert AI gleichzeitig den Wert von Expertenwissen. AI kann zwar oft genau das tun, was man ihr vorgibt, doch ohne fundiertes Fachwissen wird man vollständig von ihr abhängig. Dadurch können gravierende Fehler unbemerkt bleiben.

Fullstack-JavaScript: Wie AI die Grenzen zwischen Frontend und Backend verändert

Fullstack als Disziplin verschwindet nicht; es kommt vielmehr eine zusätzliche Dimension hinzu. Frontend, Backend und AI haben jeweils ihre eigenen Schwerpunkte, doch die Grenzen zwischen den Fachgebieten werden zunehmend unschärfer. Frontend-Entwickler:innen können Anwendungen heute einfacher deployen, Backend-APIs implementieren und Bereiche erschließen, die zuvor außerhalb ihres Tätigkeitsfelds lagen.

Wenn die einzige Serverabhängigkeit einer Anwendung die AI-Inferenz ist, kann diese inzwischen ins Frontend verlagert werden. In bestimmten Fällen kann dadurch der Server vollständig entfallen. Kleine, spezialisierte Anwendungen, die bisher lediglich einen Backend-Service benötigten, um API-Aufrufe weiterzuleiten, können heute vollständig auf der Client-Seite laufen.

Das ist eine echte architektonische Veränderung und nicht lediglich ein Verwischen von Rollenbildern. Das Backend als Architekturmodell bleibt jedoch bestehen. Es kann beispielsweise AI-Inferenz vermitteln oder die Anbindung an Cloud-Anbieter übernehmen.

Die Frage lautet daher nicht mehr: „Bist du Frontend- oder Backend-Entwickler:in?“, sondern: „Wo gehört die jeweilige Logik tatsächlich hin und an welchem Ort wird sie am sinnvollsten ausgeführt?“ AI liefert auf diese Frage eine dritte Antwortmöglichkeit, die es zuvor nicht gab.

Führt AI eine neue Ebene architektonischer Komplexität in JavaScript ein?

AI verändert JavaScript selbst nicht, da die grundlegenden Prinzipien ja unverändert bleiben. Sie verändert jedoch die Art und Weise, wie Systeme erweitert und orchestriert werden. Für Christian Liebel gilt: „KI ist eine Ergänzung zu JavaScript, keine Transformation davon.“

In der monolithischen Ära wurden Codeblöcke ergänzt. In der Ära von Microservices und Micro Frontends wurden zusätzliche Services integriert. Heute kommen Werkzeuge und Agenten hinzu. Die Grundidee bleibt ähnlich, aber die Mechanismen für Konnektivität, Routing und Dispatching unterscheiden sich deutlich.

„Normalerweise programmierst du die Entscheidung direkt in deinem Code: Ist etwas wahr oder falsch, ist es A, B oder C? Jetzt hast du einen intelligenten Knotenpunkt in deiner Anwendung, der diese Entscheidung trifft. Dadurch wird deine Software wesentlich flexibler und deutlich dynamischer”, erklärt Sebastian Springer.

AI Inferenz in Node.js und serverseitigem JavaScript

Grundsätzlich ist der Aufruf einer KI lediglich ein weiterer HTTP-Aufruf an einen externen Dienst. Sie stellt ein weiteres Werkzeug und eine zusätzliche Abstraktionsschicht dar. Der Unterschied besteht darin, dass heute statt strukturierter Daten natürlich formulierte Prompts ausgetauscht werden. Die Ergebnisse müssen anschließend wieder in eine strukturierte Form überführt werden, damit sie maschinell verarbeitet werden können. Die Herausforderung besteht darin, aus inhärent probabilistischen Systemen zuverlässig strukturierte Ausgaben zu erhalten.

Streaming-APIs spielen dabei eine zentrale Rolle, erläutert Liebel. Viele Entwickler:innen fühlen sich jedoch noch nicht wohl dabei, Backends auf dieser Grundlage zu entwickeln. Node.js unterstützt inzwischen das Web Streaming API. Dadurch kann Streaming-Code zwischen Client und Server geteilt werden, was einen erheblichen praktischen Vorteil darstellt.

AI im Browser: Lokale Modelle und AI Inference auf dem Endgerät

KI im Browser beschränkt sich nicht auf den aktuellen Hype rund um generative KI. Sie umfasst auch klassische Machine-Learning-Anwendungen. Die Hintergrundunschärfe in Google Meet ist beispielsweise seit Jahren ein etabliertes Beispiel für ein lokal ausgeführtes KI-Modell. Die zentrale Frage lautet dabei: „Wo wird das KI-Modell ausgeführt?“

Es gibt zwei Ansätze (Tabelle 1):

Ansatz 1: Eigene KI-Modelle direkt im Browser

Die Website bringt ein vortrainiertes Modell selbst mit – beispielsweise ein Open-Weights-Modell von Hugging Face – und führt dieses innerhalb der bereits vorhandenen JavaScript-Laufzeitumgebung des Browsers aus. Relevante APIs sind unter anderem WebGPU und WebNN. Auf Framework-Ebene kommen beispielsweise Apache TVM, Transformers.js oder ONNX Runtime Web zum Einsatz.

Angular Track

JavaScript & Angular Days

Entwickle moderne Angular-Apps: Skalierbar, performant, zukunftssicher!

Nutze RxJS, Signals & NgRx für reaktive Anwendungen. Setze Playwright & Jest für zuverlässige Tests ein. Integriere GenAI, SSR & Micro Frontends für smarte Architektur. Lerne in Angular Schulungen, wie du barrierefreie, robuste Apps entwickelst, die im Einsatz überzeugen.

Lerne von führenden Expert:innen:

- **UI:** Barrierefreiheit (A11y), Design Tokens & Storybook für skalierbare Angular-Apps
- **GenAI in Angular:** Copilot & WebLLM für KI-gestützte Entwicklung trainieren
- **Testing:** E2E- und Unit-Tests mit Playwright & Jest praxisnah umsetzen
- **Reaktive Modelle & State RxJS, Signals & NgRx** sinnvoll kombinieren
- **Architektur:** Monorepos, Micro Frontends & Standalone Components nutzen
- **Performance:** SSR, Incremental Hydration, Deferrable Views mit Angular 20 optimieren

Ansatz 2: Integrierte KI-Modelle im Browser

Der Browser selbst enthält das Modell und führt es aus. Entwickler:innen wählen lediglich den Anwendungsfall, nicht jedoch das Modell. Dieser Ansatz ist derzeit in Chrome und Edge implementiert. Chrome verwendet Gemini Nano, das in den Browser heruntergeladen und anschließend lokal auf dem Gerät ausgeführt wird. Das Prompt API dient dabei als Schnittstelle zum Sprachmodell.

Eigenes KI-Modell mitbringen	Integrierte KI verwenden
• Präzise Auswahl des verwendeten Modells • Problematisch hinsichtlich des Speicherbedarfs • Aufgrund der Same-Origin-Policy können Modelle nicht zwischen Websites geteilt werden • Wenn jede Website ein Modell mit beispielsweise 5 GB Größe mitbringt, wird lokaler Speicher schnell knapp	• Nutzung der vollen nativen Hardwareleistung möglich • Ausführung direkt auf der Hardware statt über zusätzliche Abstraktionsschichten • WebGPU und WebNN sind im Vergleich typischerweise etwa 10–15 % langsamer • Entwickler:innen wissen nicht genau, welches Modell sie erhalten • Die Qualität kann dadurch unvorhersehbar variieren

TABELLE 1 Lokale KI-Modelle vs. integrierte Browser-KI: Vor- und Nachteile

„Die lokale Ausführung von KI ist eine Form von Edge AI – also KI direkt auf dem Gerät der Nutzer:innen. Niemand sonst kann die Daten sehen. Sie funktioniert offline und bietet erhebliche Vorteile für den Datenschutz", sagt Christian Liebel.

Wann ist KI im Browser sinnvoller als Cloud-Inferenz?

Es gibt zahlreiche Anwendungsfälle, in denen lokale KI beziehungsweise KI im Browser sinnvoller ist als Cloud-basierte Inferenz. Liebel nennt insbesondere vier Aspekte:

Kosten: Es entstehen keine Kosten für Cloud-Inferenz. Das ist sowohl für Unternehmen als auch für Hobbyentwickler:innen relevant, die KI-Funktionen einsetzen möchten, aber keine nutzerabhängigen Inferenzkosten tragen können.

Latenz: Ein Beispiel ist die Hintergrundunschärfe bei Videokonferenzen. Das Übertragen von Videobildern an einen Server und zurück wäre zu langsam. Die lokale Verarbeitung erfolgt nahezu verzögerungsfrei.

Datenschutz: Prompts und Eingabedaten verlassen das Gerät niemals.

Offline-Fähigkeit: Die Funktionen stehen unabhängig von der Netzwerkverbindung zur Verfügung.

Sicherheit und Datenschutz in KI-gestützten Webanwendungen

Es ist wichtig, zwischen zwei Sicherheitsdimensionen zu unterscheiden:

1. KI-unterstützte Angriffe, bei denen Angreifer:innen KI einsetzen, um Schwachstellen zu finden.
2. Durch KI eingeführte Schwachstellen, die durch die Integration von KI in Anwendungen entstehen.

KI-Modelle sind sehr gut darin, Muster zu erkennen und Schwachstellen deutlich effizienter als Menschen aufzuspüren. Die Zahl der Fehler steigt dadurch nicht zwangsläufig – sie werden lediglich leichter entdeckt.

Ein eindrucksvolles Beispiel ist das Mythos-Projekt von Anthropic. Mit der Technologie wurde ein Fehler im weit verbreiteten Betriebssystem BSD gefunden, der über etwa 30 Jahre hinweg von Menschen unentdeckt geblieben war. Dies verdeutlicht, wie fragil die IT-Welt tatsächlich ist und wie viel KI erkennen kann, was Menschen übersehen.

Eine praktische Gegenmaßnahme gegen KI-gestützte Angriffe besteht laut Springer darin, KI-basierte Sicherheitsanalysen als festen Bestandteil des Softwareentwicklungsprozesses zu etablieren. „Nutze dieselben Werkzeuge, die potenzielle Angreifer:innen verwenden würden, bevor diese gegen dich eingesetzt werden können. KI-gestützte Schwachstellenanalysen sollten Teil jeder Entwicklungs-Pipeline sein“, so Springer weiter.

Diese Sicherheitsfragen entstehen auch deshalb, weil KI zunehmend selbst Teil der Anwendung wird. Damit stellt sich eine weitere Frage: **Wenn KI zu einem nativen Baustein von Webanwendungen wird, verändert sie dann auch das Erscheinungsbild einer Anwendung?**

Von Websites zu AI-nativen Anwendungen

Anwendungen sind traditionell rund um Seiten organisiert. Nutzer:innen navigieren durch Menüs, Suchergebnisse, Formulare und Workflows, um Informationen oder Funktionen zu erreichen. KI-gestützte Interfaces stellen dieses Modell infrage.

Nir Kaufman, Fullstack-Entwickler, AI-Trainer und Keynote-Speaker der iJS, beschreibt eine Zukunft, in der Nutzer:innen nicht mehr durch vollständige Anwendungen navigieren. Stattdessen interagieren sie mit einem KI-Assistenten, der sie durch eine Konversation führt und dabei die benötigten Teile der Website dynamisch zusammenstellt.

Eine Karte erscheint, wenn ein Ort relevant wird. Eine Buchungskomponente wird eingeblendet, wenn eine Reservierung erforderlich ist. Und sobald eine Transaktion abgeschlossen werden soll, erscheint ein Zahlungs-Widget.

Das ist mehr als ein neues Frontend-Pattern. Es verändert die Rolle der Anwendung selbst. Anstatt Nutzer:innen durch Seiten und Navigationsstrukturen zu führen, stellen Anwendungen Fähigkeiten bereit, die innerhalb einer laufenden Konversation aufgerufen werden können. Die Website wird weniger zu einem Ziel, das man besucht, und stärker zu einer Sammlung von Services, Daten und Interaktionskomponenten, auf die ein KI-Assistent bei Bedarf zugreifen kann.

Für Entwickler:innen könnte dies eine andere Art der Anwendungskonstruktion erforderlich machen. Anstatt Software um Seiten, Routen und Navigationshierarchien herum aufzubauen, könnten Anwendungen zunehmend aus wiederverwendbaren Komponenten bestehen, die dynamisch ausgewählt und zusammengesetzt werden. Entwickler:innen entwerfen in diesem Modell **Fähigkeiten und Interaktionen statt vorgegebener User Journeys**.

Neue Ansätze wie Googles **Agent-to-UI-Protokoll (A2UI)** untersuchen diese Idee. Agenten beschreiben UI-Elemente als strukturierte Objekte, während Clients bei Bedarf die passende Benutzeroberfläche rendern. Anstatt komplette Screens zu übertragen, können Systeme gezielt einzelne Interaktionskomponenten in eine laufende Konversation streamen.

Aus dieser Perspektive betrachtet, ist KI nicht einfach eine weitere Funktion bestehender Systeme. Sie wird zu einer **neuen Zugriffsschicht zwischen Nutzer:innen und Anwendungen**. Die architektonische Herausforderung verschiebt sich damit vom Design von Seiten und User Flows hin zum Design von Fähigkeiten, die dynamisch entdeckt, kombiniert und orchestriert werden können.

AI-native User Experiences gestalten

Wenn KI zu einer neuen Zugriffsschicht auf Anwendungen wird, ist die nächste Herausforderung nicht mehr primär technischer Natur. Es geht darum, Interaktionen so zu gestalten, dass Menschen sie verstehen, ihnen vertrauen und sie kontrollieren können.

Der UX-Forscher und Berater Christian Kuhn nähert sich dieser Frage mit sechs Designprinzipien. Statt KI lediglich als Feature zu betrachten, konzentriert er sich darauf, wie Menschen mit intelligenten Systemen zusammenarbeiten.

Die Prinzipien stellen menschliche Handlungsfähigkeit, Empathie, Transparenz, Kontrollierbarkeit und die Möglichkeit zur Fehlerkorrektur in den Mittelpunkt:

- **Human First:** KI sollte menschliche Fähigkeiten erweitern und menschliches Urteilsvermögen nicht ersetzen.
- **Empathy First:** Systeme sollten Kontext und Bedürfnisse der Nutzer:innen verstehen, statt Menschen zu zwingen, sich an die Maschine anzupassen.
- **Automation vs. Augmentation:** Nicht jede Aufgabe sollte vollständig automatisiert werden. Häufig profitieren Nutzer:innen davon, aktiv am Prozess beteiligt zu bleiben.
- **Transparency and Confidence:** Nutzer:innen müssen nachvollziehen können, woher Informationen stammen und wie zuverlässig sie sind.
- **Control and Editability:** KI-generierte Ergebnisse sollten editierbar, reversibel und unter menschlicher Kontrolle bleiben.
- **Mental Models and Graceful Failure:** Systeme müssen Menschen dabei helfen zu verstehen, wie sie funktionieren, und bei Fehlern klare Wege zur Wiederherstellung anbieten.

Mehrere dieser Prinzipien gewinnen an Bedeutung, sobald KI-Systeme das klassische Chatinterface verlassen. Kuhn formuliert es zugespitzt: **„Texteingabe ist de facto die anstrengendste und schlechteste Eingabemethode."**

Mit der Weiterentwicklung von Sprache, Bildverarbeitung und Kontextverständnis werden konversationsbasierte Experiences wahrscheinlich weit über die Texteingabe in ein Chatfeld hinausgehen. Für UX-Designer:innen entsteht dadurch eine neue Designdisziplin. Es geht nicht mehr ausschließlich darum, Screens, Navigationspfade oder Workflows zu optimieren. Zunehmend geht es darum zu gestalten, **wie Menschen und KI-Systeme zusammenarbeiten, miteinander kommunizieren und mit Fehlern umgehen.**

Wohin entwickelt sich die Webentwicklung?

Die Architektur von Webanwendungen verändert sich – und mit ihr die Art und Weise, wie Nutzer:innen mit diesen Anwendungen interagieren. Wenn KI Teil des Web-Stacks wird, überschneiden sich Fragen, die früher hauptsächlich Entwickler:innen betrafen, zunehmend mit Themen aus Architektur, Produktentwicklung und UX. Entscheidungen über Inferenz, Interfaces, Workflows und Nutzerführung können nicht länger isoliert voneinander betrachtet werden.

Dadurch wird Zusammenarbeit wichtiger, nicht unwichtiger. Die Entwicklung KI-gestützter Webanwendungen erfordert, dass technische, architektonische und UX-Perspektiven deutlich früher zusammengebracht werden, als dies in vielen Teams heute üblich ist.

Gleichzeitig macht KI die Grundlagen des Software Engineerings nicht überflüssig. Im Gegenteil: Sie macht sie noch wichtiger. Saubere Architekturen, klare Dokumentation, zuverlässige Automatisierung, umfassende Tests, gute Observability und von Anfang an berücksichtigte Sicherheit bleiben essenziell. Je dynamischer und komplexer Systeme werden, desto wichtiger werden diese Disziplinen.

Für Webentwickler:innen, Architekt:innen und UX-Designer:innen besteht die Herausforderung deshalb aus zwei Teilen: **offen für neue Architektur- und Interaktionsmodelle zu bleiben und gleichzeitig weiterhin jene Engineering-Prinzipien anzuwenden, die schon immer notwendig waren, um zuverlässige Software zu entwickeln.**



Ines Chargui ist Online-Redakteurin bei Software & Support Media. Ihr Schwerpunkt liegt auf allen Themen rund um JavaScript.

Wie WebMCP unzuverlässiges UI-Scraping durch strukturierte, browsernative Agentenschnittstellen ersetzt

WebMCP: die Brücke zwischen AI Agents und dem Web

von Gil Fink

KI-Agenten stoßen bei klassischen Weboberflächen schnell an Grenzen, weil sie auf fehleranfälliges DOM-Parsing und simulierte Nutzerinteraktionen angewiesen sind. WebMCP führt einen browsernativen Ansatz ein, über den Anwendungen strukturierte Tools direkt für Agenten bereitstellen können. Dadurch wird Automatisierung zuverlässiger, sicherer und deterministischer. Der Artikel erklärt, wie WebMCP funktioniert, vergleicht den Ansatz mit bestehenden Methoden und zeigt deklarative sowie imperative Implementierungsmuster.

In letzter Zeit gab es viel Wirbel um KI-Agenten. Die Branche verspricht viel: autonome oder halbautonome Systeme, die komplexe Workflows für uns übernehmen. Wenn Sie jedoch Frontend-Architekt oder Senior Developer sind und tatsächlich versucht haben, einen KI-Agenten durch eine moderne Webanwendung navigieren zu lassen, sind Sie vermutlich längst an Grenzen gestoßen. Denn jenseits der Marketingrhetorik zeigt sich: Die agentengesteuerte Bedienung ist äußerst fragil.

Wenn ein KI-Agent auf herkömmliche Web-Browsing-Modi setzt, ist er im Grunde nur ein etwas beeindruckenderer Scraper. Er erstellt Screenshots, analysiert unübersichtliche, tief verschachtelte DOM-Bäume und versucht anhand von Heuristiken, ARIA-Labels, IDs oder CSS-Klassen zu erraten, was ein Button tut. Sobald Ihre App einen clientseitigen Layout-Shift auslöst, eine Komponente in Ihrem Designsystem aktualisiert oder eine asynchrone Zustandsänderung verarbeitet, gerät der Agent aus dem Tritt. Er bleibt in Schleifen hängen oder, schlimmer noch, halluziniert Daten und übermittelt halb ausgefüllte Formulare.

Was wäre, wenn unsere Webanwendungen nicht darauf angewiesen wären, dass ein externer KI-Agent sich mühsam durch eine ausschließlich für menschliche Nutzer gestaltete Oberfläche arbeitet? Was wäre, wenn wir Schnittstellen gestalten könnten, die KI-Agenten ausdrücklich, sauber und sicher mitteilen, was sie tun können, wie mit ihnen interagiert werden soll und welche strukturierten Daten sie erwarten? Genau dieses Problem will WebMCP (Web Model Context Protocol) lösen.

WebMCP wurde kürzlich als neuer Webstandard vorgeschlagen und ist als Early Preview in Google Chrome verfügbar. Für das Frontend-Engineering im KI-Zeitalter kann das ein entscheidender Schritt sein. Es ermöglicht Frontend-Entwicklerinnen und -Entwicklern, die Lücke zwischen Webanwendungen und dem schnell wachsenden Ökosystem der KI-Agenten mit einem sauberen Architekturpattern zu schließen: Progressive Enhancement für KI.

In diesem Artikel untersuchen wir die zugrunde liegende Mechanik von WebMCP, ordnen ein, warum es ein wichtiges Werkzeug für den modernen Frontend-Stack ist, analysieren seine Kernarchitektur und gehen Schritt für Schritt durch die Implementierung der imperativen und der deklarativen Variante.

Die Unzuverlässigkeit von Agentic Actuation

Bevor wir WebMCP richtig einordnen können, müssen wir zunächst die architektonischen Grenzen verstehen, die prägen, wie KI-Agenten heute mit dem Web interagieren.

Wenn ein LLM-gestützter Agent in heutigen Webumgebungen eine Aktion auf einer Website ausführen soll – etwa einen Flug buchen, ein komplexes Spesenformular ausfüllen oder eine Anwendungsdiagnose starten –, stützt er sich auf einen Prozess namens Actuation. Damit ist gemeint, dass der Agent manuelle Nutzerinteraktionen simuliert, beispielsweise Mausbewegungen, Klicks, Scrollen und Tastatureingaben.

Dazu erhält der Agent üblicherweise eine serialisierte Darstellung der aktiven Seite. Das kann ein roher HTML-String sein, ein vereinfachter Accessibility Tree oder eine Sequenz von Viewport-Screenshots. Der Agent verarbeitet diese Daten, sagt die Koordinaten oder DOM-Selektoren des Elements voraus, mit dem er interagieren muss, und führt anschließend die simulierte Eingabe aus.

Dieser Ansatz weist mehrere gravierende architektonische Schwächen auf:

1. *Semantisches Rauschen:* Webseiten enthalten viele Elemente, die für das visuelle Verständnis und die Markenwirkung wichtig sind, für ein LLM aber lediglich Rauschen darstellen. Navigationsmenüs, Werbebanner, Sidebars, Tracking-Skripte und komplexe CSS-Layouts sind nicht dafür gedacht, von einem Agenten genutzt zu werden. Der Agent verschwendet wertvollen Platz im Kontextfenster des Modells darauf, Layoutrauschen herauszufiltern, nur um ein einzelnes Formularfeld zu finden. Das macht den Prozess teuer.
2. *Fehlinterpretation des Kontexts:* Für Menschen gestaltete Benutzeroberflächen stützen sich stark auf visuelle Hinweise und Mikrointeraktionen. Eine Date-Picker-Komponente kann für einen rohen HTML-Parser wie ein einfaches Eingabefeld wirken. Tatsächlich erfordert die Interaktion aber, auf ein kleines Kalender-Icon zu klicken, über Paginierungspfeile durch Monate zu navigieren und eine bestimmte Tabellenzelle im Picker-Panel auszuwählen. Ein KI-Agent, der versucht, diese Komponente direkt über simulierte Klicks zu bedienen, kann den internen Zustand der Komponente beschädigen.
3. *Kein synchroner Zustand:* Single Page Applications (SPAs), die mit modernen Frameworks wie React, Next.js oder Vue gebaut sind, verwalten ihren Zustand asynchron. Wenn ein Agent auf einen Button klickt, der einen clientseitigen Routenwechsel oder eine verzögerte Zustandsänderung auslöst, versucht der Agent seine nächste Aktion möglicherweise, bevor das DOM vollständig neu gerendert wurde. Das führt zu Fehlern bei der Zielauswahl und zu unterbrochenen Abläufen.

4. *Mangel an Determinismus:* LLMs sind von Natur aus probabilistisch. Wenn Sie einen Agenten bitten, ein Registrierungsformular zehnmal auszufüllen, besteht eine statistisch relevante Wahrscheinlichkeit, dass er mindestens einmal ein Feldlabel falsch interpretiert oder ein Datenelement falsch formatiert. In Enterprise-, Finanz- oder Healthcare-Anwendungen ist eine Erfolgsquote von 90 Prozent ein völliges Scheitern. Beim Ausführen von Transaktionen brauchen wir vollständigen Determinismus.

WebMCP setzt genau an diesen Problemen an. Statt den Agenten zu zwingen, Absichten aus der Darstellungsschicht abzuleiten, ermöglicht WebMCP der Darstellungsschicht, ihre Funktionen direkt gegenüber der Browser-Runtime zu deklarieren.

Was ist WebMCP?

WebMCP ist eine Adaption und Erweiterung des umfassenderen Model Context Protocol (MCP), das ursprünglich von Anthropic und Open-Source-Mitwirkenden entwickelt wurde. Während Standard-MCP darauf ausgerichtet ist, KI-Modelle über Transportschichten wie SSE (Server-sent Events) oder stdio mit lokalen Entwicklungswerkzeugen, Datenbanken und sicheren Enterprise-Umgebungen zu verbinden, bringt WebMCP dieses Protokoll nativ in den Browser.

WebMCP stellt eine strukturierte, besonders sichere Brücke zwischen der Webseite, die in einem Tab ausgeführt wird, und dem KI-Assistenten bereit, der auf Browserebene arbeitet (**Abb. 1**). Es verwandelt Ihre Webanwendung von einem statischen Dokument, das visuell gescrapt werden muss, in eine dynamische, abfragbare und ausführbare Tool-Engine.

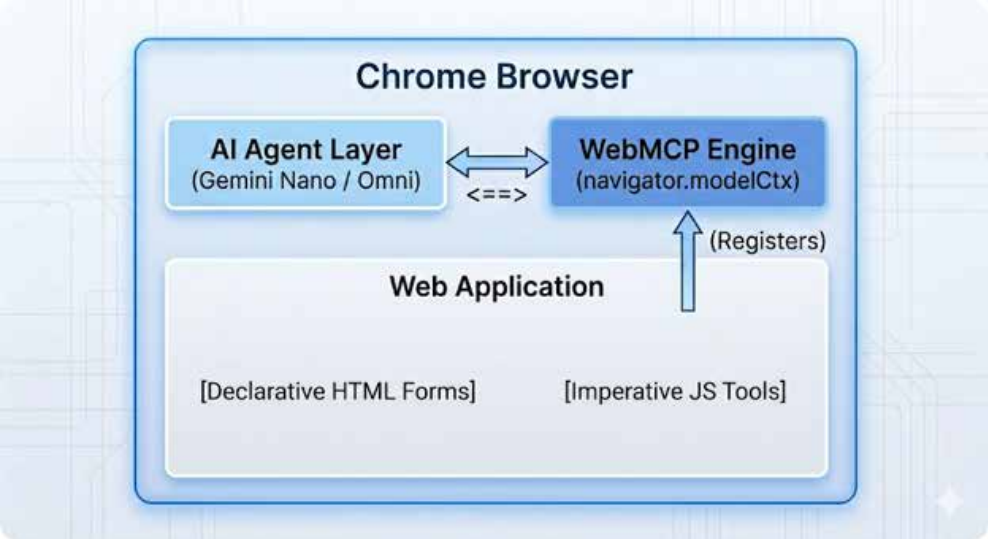


ABB. 1 AI Agent und WebMCP verbinden Webanwendungen im Browser

Wenn eine Webseite WebMCP nutzt, registriert sie bestimmte Funktionen im Model Context des Browsers. Der Agent muss nicht mehr erraten, wie eine Aufgabe auszuführen ist. Stattdessen kann er die Liste der registrierten Tools abrufen, ihre streng typisierten Definitionen auslesen und sie mit deterministischen JSON-Argumenten aufrufen.

Die drei Säulen von WebMCP

Um eine robuste Alternative zur fehleranfälligen UI-Bedienung durch Agenten bereitzustellen, stützt sich WebMCP auf drei zentrale Säulen.

1. *Standardisierte Discovery:* Wenn ein Nutzer eine Website aufruft, benötigt ein Agent, der innerhalb des Browsers oder parallel dazu arbeitet, einen einheitlichen Weg, um abzufragen, welche Operationen auf dieser konkreten Seite verfügbar sind. WebMCP stellt dafür eine native Browser-API (*navigator.modelContext*) bereit, die als Registry fungiert. Sobald die Seite initialisiert wird, registriert Ihre Anwendung ihre Tools. Der Browser kann diese Funktionen sofort an das zugrunde liegende KI-Modell übermitteln, ohne dass das Modell auch nur eine einzige Zeile des visuellen HTML-Layouts Ihrer Anwendung parsen muss.
2. *Strikte JSON-Schemata:* Eine der Hauptursachen für Fehler bei Agenten sind Abweichungen im Datenformat. Ein Beispiel: Ein Agent gibt einen vollständigen Namen in ein Feld ein, das Vor- und Nachnamen strikt getrennt erwartet, oder sendet einen ISO-Zeitstempel an ein benutzerdefiniertes Eingabefeld, das das Format MM/DD/YYYY voraussetzt. WebMCP verlangt, dass jedes registrierte Tool ein *inputSchema* definiert, das der JSON-Schema-Spezifikation entspricht. Da die vom Agenten vorgesehenen Argumente gegen dieses JSON-Schema validiert werden, bevor Code ausgeführt wird, kann WebMCP strukturbedingte Halluzinationen verhindern. Das LLM muss seine Parameter exakt so formatieren, wie es Ihre Anwendungslogik vorgibt. Dieses Schema beschreibt:
    - die zulässigen Properties des Tools
    - die jeweils erwarteten primitiven Datentypen
    - Enums, die Werte auf gültige Zustände einschränken
    - Formats wie date, email oder URI
    - erforderliche oder optionale Parameter

3. *Gemeinsamer Ausführungskontext und visueller State*: Anders als Server-zu-Server-API-Aufrufe werden WebMCP-Tools direkt im aktiven Browsing-Kontext des aktuellen Nutzer-Tabs ausgeführt. Das ist ein entscheidender Unterschied: Wenn ein Agent ein WebMCP-Tool aufruft, wird der Handler des Tools im lokalen clientseitigen JavaScript ausgeführt. Das Tool kann also direkt den In-Memory-Zustand Ihrer Anwendung lesen, das DOM manipulieren, Formular-Komponenten aktualisieren oder clientseitige Router anstoßen. Da diese Ausführung sichtbar auf dem Bildschirm erfolgt, behält der Nutzer den Vorgang jederzeit im Blick. Er kann beobachten, wie sich die Anwendung in Echtzeit aktualisiert. Dadurch bleibt das Human-in-the-Loop-Modell erhalten, ohne dass Sie die sorgfältig gestaltete Benutzeroberfläche Ihrer Anwendung aufgeben müssen.

WebMCP vs. Standard-MCP vs. Actuation

Um besser einordnen zu können, wo WebMCP in Ihren Tech-Stack passt, vergleichen wir es anhand zentraler Kriterien mit traditioneller UI-Bedienung durch Agenten und mit Standard-MCP im Backend.

Feature	Traditionelle UI Actuation	Standard-MCP (Backend)	WebMCP (Browser-nativ)
Ausführungsumgebung	externer Agent (Headless/Virtual Display)	serverseitige Node/Go/Python Runtime	clientseitige Browser-Tab-Sandbox
Schnittstellenmedium	DOM Selectors, Coordinates, Screenshots	Secure APIs, DB Connections, File-Systems	JavaScript Event Handlers, HTML Form Fields
Nutzersichtbarkeit	für den Nutzer verborgen, außer bei Bildschirmfreigabe	vollständig unsichtbare Backend-Verarbeitung	sichtbar auf dem Bildschirm innerhalb des aktiven UI ausgeführt
State-Zugriff	rein visuell / Serialized DOM Scraping	Backend-Datenbank oder Session Records	live, in-Memory Client-side Application State
Zuverlässigkeit	niedrig, anfällig für Layout- und Stylingänderungen	hoch, direkte Program-to-Program APIs	hoch, klar definierter Vertrag zwischen Anwendung und Agent
Implementierungsaufwand	keiner in der Web-App; extrem hoch beim Agent	hoch, Backend-Set-up, Authentication, Proxying	niedrig bis moderat, Frontend Progressive Enhancement

Die zwei Implementierungsmodelle: deklarativ vs. imperativ

Die Chrome-Implementierung der WebMCP-Spezifikation trägt dem Umstand Rechnung, dass Webentwicklung von einfachen statischen Dokumenten bis hin zu hochinteraktiven, zustandsgetriebenen Single Page Applications reicht. Um dieses Spektrum abzudecken, stellt sie zwei verschiedene APIs bereit:

1. Das deklarative API ist für eine schnelle Integration mit geringem Overhead konzipiert. Es ermöglicht es, Standard-HTML-*<form>*-Elemente allein durch semantische Attribute in WebMCP-Tools zu verwandeln. Sie müssen weder eigene JavaScript-Registries schreiben noch die Schema-Extraktion manuell umsetzen. Der Browser parst die Eingaben des Formulars, leitet daraus automatisch ein JSON-Schema auf Basis



Angular Bootcamp

Generative AI für Angular
AI-Integration im Frontend und Backend

Lerne in 2 Tagen:
- **Generative AI & RAG:** Konzepte & Praxis-Use Cases
- **AI-Backends aufbauen:** LangChain & moderne APIs
- **UX & Streaming-APIs** für Anwendungen
- **Angular-Integration:** TypeScript & Frontend-Workflows
- **Enterprise-Szenarien:** KI in Business-Anwendungen einsetzen

Jetzt entdecken!



JavaScript Bootcamp

Fullstack JavaScript
von der API bis zum Frontend

Lerne in 2 Tagen:
- **Backend & REST:** APIs entwickeln und sauber strukturieren
- **Validierung & Auth:** Daten prüfen und Authentifizierung verstehen
- **Frontend-Architektur:** React entwickeln, Angular & Vue vergleichen
- **Backend-Integration:** Frontend und Backend verbinden
- **Testing & CI/CD:** Testen, automatisieren und deployen

Jetzt entdecken!

der HTML5-Validierungsattribute ab und stellt es dem Agent-Layer bereit. Wenn der Agent das Tool aufruft, füllt der Browser die Formularfelder automatisch aus und löst das Submit-Event aus.

2. Das imperative API ist die leistungsfähigere Variante für professionelle Webentwicklerinnen und Webentwickler, die mit komplexen Benutzeroberflächen, clientseitigem Routing, State-Management-Bibliotheken und individuellen Designsystem-Komponenten arbeiten. Über *navigator.modelContext.registerTool()* schreiben Sie in Standard-JavaScript explizit Code, der die Absicht des Agenten direkt auf Ihre interne Anwendungslogik abbildet, DOM-Interaktionen bei Bedarf vollständig umgeht und saubere Zustandsänderungen ausführt.

Einrichtung Ihrer WebMCP-Entwicklungsumgebung

Da WebMCP derzeit den Standardisierungsprozess durchläuft und aktiv in experimentellen Origin Trials in Google Chrome getestet wird, müssen Sie Ihre Entwicklungsumgebung ausdrücklich darauf vorbereiten, die APIs zu erkennen und auszuführen.

Schritt 1: Aktivierung des Flags

Zunächst muss das Flag aktiviert werden:

1. Google Chrome starten
2. Rufen Sie in der Adresszeile *chrome://flags/#enable-webmcp-testing* auf.
3. Suchen Sie das Flag *WebMCP Testing* und stellen Sie es auf *Enabled* (**Abb. 2**).
4. Starten Sie den Browser neu.



ABB. 2 WebMCP Testing aktivieren

Schritt 2: Permissions-Policy implementieren

WebMCP verfügt über eine robuste, integrierte Sicherheitsgrenze, die über eine spezielle Permissions Policy namens *tools* gesteuert wird. Standardmäßig wird diese Policy auf *self* gesetzt. Das bedeutet, dass Tool-Discovery und Tool-Ausführung in Top-Level-Browsing-Contexts sowie in

Same-Origin-iFrames erlaubt sind. Gleichzeitig wird strikt verhindert, dass nicht vertrauenswürdige Cross-Origin-iFrames von Drittanbietern Tools registrieren, die eine agentische Sitzung kapern könnten.

Wenn Ihre Architektur ausdrücklich erfordert, eine WebMCP-fähige Anwendung in einem Cross-Origin-iFrame zu hosten – etwa ein eingebettetes Buchungs-Widget oder ein White-Label-Zahlungs-Gateway –, müssen Sie den Kontext explizit über das *allow*-Attribut freigeben:

```
< iframe src="https://trusted-widget.
          sparxys.com" allow="tools"></ iframe>
```

Schritt 3: das Inspector-Tool installieren

Um Ihre Anwendungsschemata zu debuggen und zu verifizieren, ohne ein umfangreiches externes Agent-Orchestration-Framework integrieren zu müssen, sollten Sie die Model Context Tool Inspector Extension aus dem Chrome Developer Ecosystem nutzen.

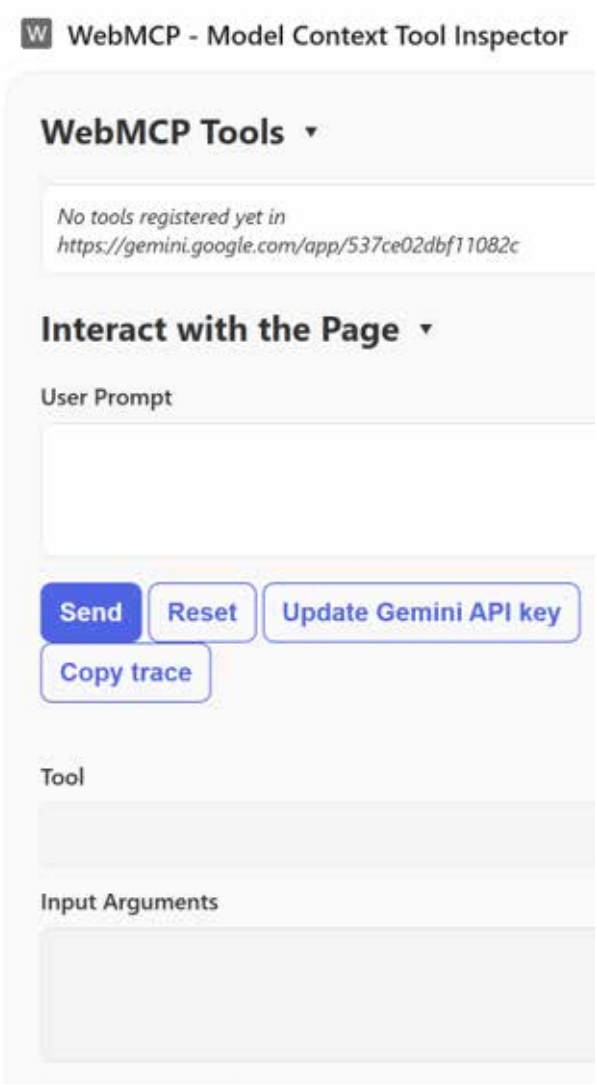


ABB. 3 Der Inspector mit geladenem Gemini-Modell

Dieses Tool stellt ein Inspector-Panel bereit, das auf navigator.modelContext zugreift und sofort alle registrierten imperativen und deklarativen Tools auf der aktiven Seite auflistet. Es enthält außerdem einen Natural-Language-Prompting-Playground, der von lokal oder in der Cloud gehosteten Gemini-Modellen unterstützt wird, etwa Gemini Nano. Dort können Sie natürlichsprachige Eingaben verwenden, um zu prüfen, ob der Agent Argumente sauber extrahiert und Ihren Schemata korrekt zuordnet (**Abb. 3**).

Implementierung des imperativen API

Die eigentliche architektonische Kontrolle bietet das imperative API. Wenn Sie eine SPA mit komplexen Zustandsgrenzen entwickeln, möchten Sie nicht, dass der Agent direkt mit dem DOM interagiert. Stattdessen soll er direkt mit Ihrer Anwendungslogik kommunizieren.

Sehen wir uns daher an, wie sich mit *navigator.modelContext.registerTool()* ein Tool registrieren lässt. Dazu erstellen wir ein kleines Beispiel mit dem imperativen API: ein Buchungsformular mit Reiseziel, Reisemonat und Anzahl der Gäste.

Schritt 1: das HTML-Interface erstellen

Erstellen Sie die *index.html* (Listing 1).

Listing 1

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Hotel Search - Imperative WebMCP</title>
  <link rel="stylesheet" href="booking.css">
</head>
<body>

<div class="booking-form">
  <div class="form-group">
    <label>Destination</label>
    <input type="text" id="dest" value="Orlando">
  </div>
  <div class="form-group">
    <label>Travel Month</label>
    <input type="month" id="dates" value="2026-08">
  </div>
  <div class="form-group">
    <label>Guests</label>
    <input type="number" id="guests" value="5">
  </div>
  <button onclick="executeSearch()">Search</button>
</div>

<div id="results"></div>

<script src="scripts.js"></script>
</body>
</html>
```

Schritt 2: das imperative Registrierungsskript schreiben

Nun erstellen wir *scripts.js*. Achten Sie genau darauf, wie wir unser JSON-Schema konstruieren. Wir möchten, dass der Agent drei streng typisierte Felder übergibt: *destination*, *dates* und *guests* (Listing 2).

Listing 2

```
function updateUI(dest, dates, guests) {
  const output = `Searching for ${guests} guests in ${dest} for
${dates}...`;
  const render = () => document.getElementById('results').innerText =
output;

  if (document.startViewTransition) {
    document.startViewTransition(render);
  } else {
    render();
  }
}

function executeSearch() {
  const dest = document.getElementById('dest').value;
  const dates = document.getElementById('dates').value;
  const guests = document.getElementById('guests').value;
  updateUI(dest, dates, guests);
}

// WebMCP Imperative Registration
if ('modelContext' in navigator) {
  navigator.modelContext.registerTool({
    name: 'searchHotel',
    description: 'Search for hotel room availability by destination, dates,
and guest count.',
    inputSchema: {
      type: 'object',
      properties: {
        destination: { type: 'string' },
        dates: { type: 'string' },
        guests: { type: 'number' }
      },
      required: ['destination', 'dates', 'guests']
    },
    execute: async (params) => {
      updateUI(params.destination, params.dates, params.guests);
```

```
        // Update visual inputs to reflect agent's actions
        document.getElementById('dest').value = params.destination;
        document.getElementById('dates').value = params.dates;
        document.getElementById('guests').value = params.guests;

        return JSON.stringify({ status: 'success', summary:
                                `Search initiated for ${params.destination}` });
      }
    });
}
```

Implementierung des deklarativen API

Das imperative API bietet vollständige Kontrolle über die Programmlogik, erfordert aber auch, dass Sie Ihre Schema-Definitionen manuell mit Ihrer Zustandsarchitektur synchron halten. Für Standardformulare zur Texteingabe, Datenverarbeitungsformulare oder Support-Desk-Szenarien ist das deklarative API schneller und schlanker. Sehen wir uns an, wie sich das vorherige imperative Beispiel in den deklarativen Ansatz übertragen lässt.

Deklaratives Markup annotieren

Durch die Nutzung von Standard-HTML-Attributen und das Anwenden des *webmcp-tool*-Konfigurationslayouts weisen wir das Chrome-Parsing-Layout an, die Registrierung in unserem Auftrag zu übernehmen. Dazu erstellen wir eine eigene Index-Datei, die wir *declarative.html* nennen (Listing 3).

Listing 3

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Hotel Search - Declarative WebMCP</title>
  <link rel="stylesheet" href="booking.css">
</head>
<body>

<form class="booking-form"
      id="searchForm"
      toolname="searchHotel"
      tooldescription="Search for hotel room availability"
      toolautosubmit>

  <div class="form-group">
```

```
    <label for="dest">Destination</label>
    <input type="text" name="destination" id="dest" value="Orlando"
      toolparamdescription="The city to search for hotels in">
  </div>
  <div class="form-group">
    <label for="dates">Travel Month</label>
    <input type="month" name="dates" id="dates" value="2026-08"
      toolparamdescription="The month and year of travel">
  </div>
  <div class="form-group">
    <label for="guests">Guests</label>
    <input type="number" name="guests" id="guests" value="5"
      toolparamdescription="Total number of guests traveling">
  </div>
  <button type="submit">Search</button>
</form>

<div id="results"></div>

<script>
  const form = document.getElementById('searchForm');

  form.addEventListener('submit', (e) => {
    e.preventDefault();
    const data = new FormData(form);
    const outputMsg = `Searching for ${data.get('guests')}
          guests in ${data.get('destination')} for ${data.get('dates')}...`;

    const updateUI = () => document.getElementById('results')
                                                    .innerText = outputMsg;

    if (document.startViewTransition) {
      document.startViewTransition(updateUI);
    } else {
      updateUI();
    }

    // The agentInvoked is true if the browser's agent triggered the form
    if (e.agentInvoked) {
      e.respondWith(Promise.resolve(JSON.stringify({
        status: 'success',
        message: outputMsg
      })));
    }
  });
```

```
  // Listen for agent actuation events for debugging/logging
  window.addEventListener('toolactivated', ({ toolName }) => {
    console.log(`Agent activated WebMCP tool: ${toolName}`);
  });
</script>
</body>
</html>
```

Analyse der deklarativen Engine-Konvertierung

Der elegante Teil daran: Der Browser übernimmt im Hintergrund einen Großteil der Arbeit. Wenn Sie native HTML5-Validierungsattribute und Standard-Markup verwenden, leitet WebMCP automatisch Typen, Einschränkungen und Enums ab und stellt dem Agenten ein sauber strukturiertes JSON-Schema bereit – ganz ohne manuelles JavaScript-Mapping (Listing 4).

Listing 4

```
{
  "name": "searchHotel",
  "description": "Search for hotel room availability",
  "inputSchema": {
    "type": "object",
    "properties": {
      "destination": {
        "type": "string",
        "description": "The city to search for hotels in"
      },
      "dates": {
        "type": "string",
        "format": "^[0-9]{4}-(0[1-9]|1[0-2])$",
        "description": "The month and year of travel"
      },
      "guests": {
        "type": "number",
        "multipleOf": 1,
        "description": "Total number of guests traveling"
      }
    },
    "required": []
  }
}
```

Wenn der Agent versucht, *searchHotel* auszuführen, injiziert der Browser die Werte sicher in die Inputs, führt browsernative Validierungs-Checks aus, um sicherzustellen, dass die Daten passen, und ruft das *Submit*-Event auf.

Architektur-Deep-Dive: Sicherheit und Grenzen der Datenisolation

Bei Protokollen, die externen Systemen erlauben, Operationen innerhalb einer aktiven Web-Session aufzurufen, muss Sicherheit oberste Priorität haben. Wenn bösartige Akteure Tools einschleusen oder unautorisierte Aktionen auslösen könnten, würde WebMCP zu einer erheblichen Sicherheitslücke. Der Standard adressiert dieses Risiko über mehrere zentrale Sicherheitsgrenzen:

1. *Same-Origin-Boundary-Überwachung:* WebMCP-Tools sind strikt an den Sicherheitskontext der Origin gebunden, die sie registriert hat. Ein Tool, das von *https://secure.sparxys.com* registriert wurde, kann nicht von einem Agenten entdeckt oder aufgerufen werden, der einen Tab mit *https://attacker-compromised-site.io* betrachtet. Die Ausführungsgrenze respektiert dabei die standardmäßige DOM-Isolation.
2. *Human-in-the-Loop-Patterns:* WebMCP gewährt automatisierten Agenten keine pauschale Erlaubnis, beliebige Codepfade in Headless-Hintergrundthreads auszuführen. Da das Browserfenster aktiv und sichtbar bleiben muss – eine zustandsverändernde Ausführung im Headless-Modus ist durch das Design der Spezifikation ausgeschlossen –, kann der Nutzer jede einzelne Änderung beobachten. Für hochsensible Aktionen wie das Bestätigen eines finalen Kaufvorgangs, das Überweisen von Geld, das Ändern von Root-IAM-Konfigurationen oder das Ändern von Passwörtern empfiehlt die Architekturspezifikation, Standarddialoge zur Nutzerbestätigung direkt in den JavaScript-Handler-Code einzubauen (Listing 5). Diese strukturelle Sicherheitsbeschränkung stellt sicher, dass ein Agent Parameter sammeln und Formulare schneller ausfüllen kann, die finale Autorisierung aber strikt unter der Kontrolle des Nutzers bleibt.

Listing 5

```
handler: async (args) => {
  const userConsent = await showCustomConfirmationModal
                    ("Are you sure you want to authorize this transaction?");
  if (!userConsent) {
    return { success: false, error: "USER DENIED CONSENT" };
  }
  // Proceed with operational routine...
}
```

Performanceoptimierung und Speichermanagement

Das direkte Offenlegen strukturierter Tools in clientseitigen Runtimes erfordert sorgfältiges Monitoring, um Memory Leaks, eine träge Rendering-Performance und übermäßige CPU-Kontextwechsel zu vermeiden – insbesondere bei modernen SPA-Stacks wie React oder Next.js.

1. *Vermeiden Sie veraltete Closure-Retainer:* Wenn Sie ein imperatives Tool über *navigator.modelContext.registerTool()* innerhalb eines React-Hooks oder im Lifecycle einer Functional Component registrieren, sollten Sie sicherstellen, dass beim Unmount der Komponente sauber aufgeräumt wird. Wenn Sie Tools bei jedem Rerender kontinuierlich neu registrieren, sammeln sich veraltete Closures an, die umfangreiche Referenzen auf den Component State im Speicher halten. Das führt zu Memory Leaks (Listing 6).
2. *Achten Sie auf nicht blockierende asynchrone Handler:* Ihr Handler-Code blockiert den Orchestrierungspfad des Agenten, bis er abgeschlossen ist. Wenn Ihr Handler aufwendige Berechnungen durchführt, große clientseitige Datenmengen sortiert oder auf langsame Drittanbieter-APIs warten muss, sollten Sie diese Arbeit sauber auslagern. Geben Sie einen Pending-Status an den Agenten zurück oder senken Sie die Ausführungspriorität über *requestIdleCallback*, damit der primäre UI-Rendering-Thread des Browsers nicht blockiert wird.

Listing 6

```
// React useEffect example
useEffect(() => {
  if (!navigator.modelContext || !navigator.modelContext.registerTool) return;

  const abortController = new AbortController();

  navigator.modelContext.registerTool({
    name: "update_dashboard_filter",
    description: "Updates data viewing parameters.",
    inputSchema: { /* ... */ },
    handler: async (args) => {
      // Context execution parameters
    }
  }, { signal: abortController.signal });

  return () => {
    abortController.abort();
  };
}, [reactiveDependencyDependencies]);
```

Zusammenfassung

Das Web entwickelt sich weiter. Wir entfernen uns zunehmend von einer Ära, in der Browser ausschließlich von Menschen genutzt wurden, die mit visuellen Point-and-Click-Oberflächen interagierten. Es entsteht ein hybrides Webökosystem, in dem menschliche Nutzer von hochspezialisierten, kontextbewussten KI-Agenten begleitet werden, die operative Aufgaben übernehmen.

Als Senior-Frontend-Entwicklerinnen und -Entwickler sowie Softwarearchitektinnen und -architekten müssen wir unsere Designphilosophie erweitern. Progressive Enhancement darf nicht länger nur als Mittel verstanden werden, um deaktiviertes JavaScript oder langsame mobile Verbindungen abzufangen. Vielmehr müssen wir agentengerechte Zugänglichkeit als grundlegenden Bestandteil moderner Frontend-Architekturen betrachten.

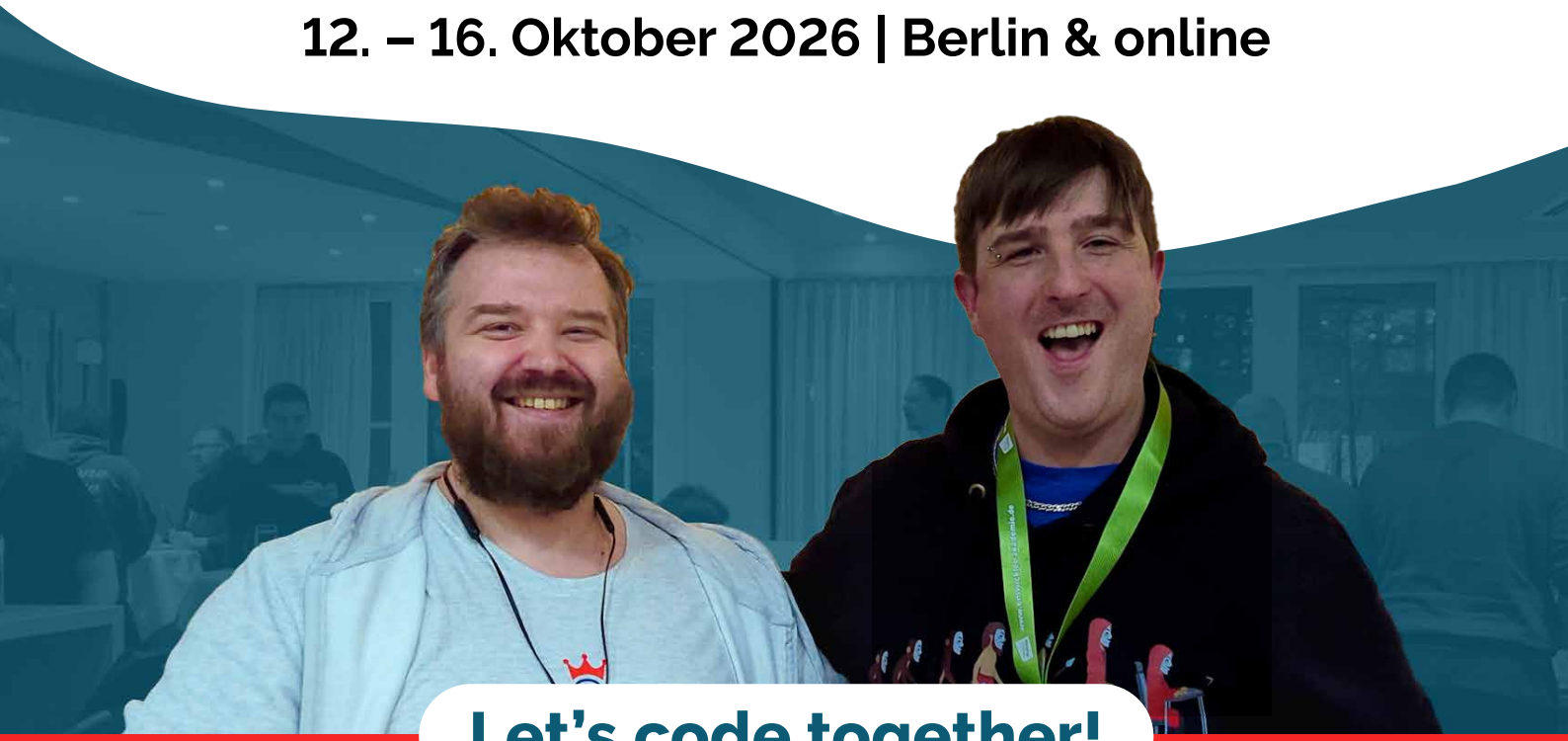
WebMCP liefert dafür ein passendes Modell. Wenn wir Zeit investieren, um saubere, streng typisierte Verträge über imperative Registrierungen und deklarative HTML5-Formularlayouts zu definieren, sorgen wir dafür, dass unsere Anwendungen in einem zunehmend agentischen Webökosystem robust, performant und für Agenten auffindbar bleiben. WebMCP nimmt derzeit Gestalt an. Aktivieren Sie die entsprechenden Flags, testen Sie Ihre Formulare mit der Inspector Extension und beginnen Sie, Ihr UI als API zu denken.



Gil Fink ist Experte für Webentwicklung, GDE für Webtechnologien, MVP für Microsoft-Entwicklungstechnologien und CEO von sparXys. Derzeit ist er als Berater für verschiedene Unternehmen tätig und unterstützt diese bei der Entwicklung webbasierter Lösungen. Außerdem ist er Mitautor mehrerer Microsoft Official Courses (MOCs) und Schulungskits sowie Mitautor des bei Apress erschienen Buchs „Pro Single Page Application Development“.

JavaScript & Angular Days

12. – 16. Oktober 2026 | Berlin & online



Let's code together!

Das große Trainingsevent für moderne Webentwicklung mit JavaScript & Angular

Erlebe neue Trends

Erreiche neue technologische Tiefe und lerne zukunftssichere Fähigkeiten. In den ersten drei Workshop-Tagen stellst du dein eigenes Programm zusammen, anschließend geht es zwei Tage zum Deep Dive ins JS- oder Angular-Bootcamp.

Learning by Doing

80 % des Programms besteht aus Workshops. Hier wird nicht nur studiert, sondern direkt ausprobiert. Egal, ob du live in Berlin oder online zugeschaltet bist.

Triff die Community

Wir verbinden dich mit Menschen, die alle eins im Sinn haben: Sich auf hohem Level auszutauschen und neue Kontakte zu knüpfen. Hier kommen alle zusammen: 20+ Toptrainer:innen der Branche und die Teilnehmenden.

JS TRACK

TRACK

+2 Bootcamps



Präsentiert von: entwickler.de akademie entwickler.de

javascript-days.de/berlin