

Lernzettel

Web-Technologien und Frontend-Integration:
REST vs GraphQL, HTTP/2, WebSockets,
Server-Sent Events

Universität: Technische Universität Berlin
Kurs/Modul: Architektur von Anwendungssystemen
Erstellungsdatum: September 19, 2025



Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

<https://study.AllWeCanLearn.com>

Architektur von Anwendungssystemen

Lernzettel: Web-Technologien und Frontend-Integration: REST vs GraphQL, HTTP/2, WebSockets, Server-Sent Events

(1) Überblick und Lernziel.

Dieses Kapitel vermittelt ein Verständnis der wichtigsten Web-Technologien für Frontend-Integration in verteilten Anwendungssystemen. Fokus liegt auf REST, GraphQL, HTTP/2, WebSockets und Server-Sent Events sowie deren Einsatzmöglichkeiten, Vor- und Nachteile und typische Architektur-Muster. Ziel ist es, Architekturentscheidungen bewusst treffen und gemäß Anwendungsfall bewerten zu können.

(2) REST vs GraphQL.

REST und GraphQL sind zwei gängige Architekturstile zur Abfrage und Manipulation von Serverdaten. Wähle je nach Anwendungsfall passende Methode.

REST:

- Ressourcenorientiert: Ressourcen werden über eindeutige URLs adressiert, z. B. GET /users, POST /orders.
- HTTP-Methoden spiegeln Operationen wider (GET, POST, PUT, PATCH, DELETE).
- Vorteile: klare Semantik, einfache Caching-Strategien, gute Tool-Unterstützung.
- Herausforderungen: Over-/Under-fetching, mehrere Endpunkte, Versionierung.
- Typische Muster: hypermedia (HATEOAS), statelessness, RESTful Endpoints.

GraphQL:

- Ein einzelner Endpunkt (typisch POST /graphql) mit einem Typ-System im Server.
- Abfragen spezifizieren exakt die benötigten Felder, was Over-/Under-fetching reduziert.
- Vorteile: flexible Queries, bessere Client-Entkopplung, einfache Aggregationen über verschachtelte Strukturen.
- Herausforderungen: komplexeres Caching, mehr Server-Logik pro Abfrage, Subscriptions für Echtzeit-Features.
- Typensystem, Introspection und klare Verlaufsmuster für Mutationen und Subscriptions.

Kompromisse und Hinweise:

- REST eignet sich gut für klare Ressourcen-APIs, einfache Caching-Szenarien und stabile, gut verstandene Schnittstellen.
- GraphQL eignet sich gut für komplexe UI-Ansichten mit vielen verschachtelten Daten, reduziert Overhead, erfordert aber oft zusätzliche Caching-Strategien und komplexere Client-Logik.

(3) HTTP/2 – Basiswissen für Frontends.

HTTP/2 baut auf dem Transportprotokoll TLS/HTTPS auf und verbessert die Effizienz von Web-Anwendungen wesentlich.

- Multiplexing: Mehrere Streams gleichzeitig über eine einzige TCP-Verbindung, wodurch Head-of-Line-Blockierung reduziert wird.
- HPACK-Header-Kompression: Reduziert Overhead durch komprimierten Header-Verkehr.
- Server Push: Server kann Ressourcen vorschlagen, bevor der Client sie anfragt (mit Vorsicht einsetzen, Caching beachten).
- Priorisierung: Wichtige Ressourcen können Vorrang erhalten.
- Einfluss auf REST/GraphQL: REST- und GraphQL-APIs profitieren von geringeren Ladezeiten und besserer Parallelität.

(4) WebSockets – bidirektionale Echtzeit-Verbindungen.

WebSockets ermöglichen eine persistente, bidirektionale Kommunikation über einen einzigen, offenen Kanal.

- Handshake erfolgt über ein Upgrade auf ws/wss (TLS-gesichert).
- Vorteile: echtes Push-Feedback, geringer Overhead pro Nachricht im Vergleich zu wiederholtem Polling.
- Typische Anwendungsfälle: Chat, Live-Updates, Multiplayer-Interaktionen, kollaborative Editoren.
- Herausforderungen: Skalierbarkeit (Stateful-Server), Infrastruktur (Proxy/Load-Balancer), Firewall-Umgehung, Backpressure-Handling.

(5) Server-Sent Events (SSE) – unidirektionales Server-Streaming.

SSE ermöglicht dem Server, kontinuierlich Ereignisse an den Client zu senden.

- Einfache API: EventSource im Browser, `text/event-stream` als Content-Type.
- Vorteile: einfache Implementierung, gut geeignet für Live-Feeds, Logs, Benachrichtigungen.
- Einschränkungen: nur serverseitig zu Client; kein nativer Bidirektional-Channel, kein komplexes Messaging-Protokoll.
- Robustheit: automatische Wiederverbindung bei Verbindungsabbruch, eventuelle Reconnections-Strategien.

(6) Frontend-Integrationsmuster – Auswahl anhand von Anforderungen.

Der passende Mix aus REST, GraphQL, HTTP/2, WebSockets und SSE hängt von den Anforderungen ab.

- Realzeit-Updates: WebSockets oder GraphQL Subscriptions (über WebSocket-Protokolle) bzw. SSE für einfache Push-Feeds.
- Abfragevielfalt vs. Konsistenz: GraphQL bei verschachtelten UI-Anfragen; REST bei klar definierten Ressourcen.

- Netzwerkauslastung: HTTP/2 verbessert Multiplexing; gezieltes Caching für REST, adäquate Client-Caches (z. B. Apollo-Cache bei GraphQL).
- Offline- bzw. Fehlertoleranz: ggf. Kombination aus lokalen Speicherarten und Synchronisationslogik.
- Sicherheits- und Skalierungsaspekte: Authentifizierung/Autorisierung, TLS, Lastverteilung, Back-End-Streaming-Strategien.

(7) Entscheidungsbausteine und Best Practices.

Hinweise zur Architekturentscheidung:

- Verwendungszweck klären: statische vs. dynamische Daten, Push-Niveau, Bidirektionalität.
- Komplexität vs. Leistung gegen Konsistenz: GraphQL reduziert Overfetch, erfordert jedoch mehr Client-Logik; REST mit HTTP/2 bleibt einfach portierbar.
- Konsistenzmodell beachten: Caching-Strategien für REST, GraphQL (Client-Cache vs. Server-Cache).
- Real-Time-Anforderungen definieren: SSE für einfache Updates, WebSockets/Subscriptions für anspruchsvolle Interaktionen.
- Infrastruktur beachten: Proxy- und Load-Balancer-Unterstützung, Skalierbarkeit von Verbindungen (WebSocket-Handling).