REST, GRAPHQL, GRPC UND CO.

Moderne API-Stile im Vergleich

Florian Bader

MIT WEM HABT IHR ES ZU TUN?

Florian Bader

Solution Architect | CTO

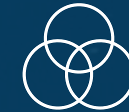
Florian.Bader@lunaris.digital

https://lunaris.digital
https://github.com/florianbader

LUNARIS
Digital Solutions

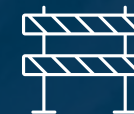
KEY TAKE AWAYS

Welche API-Stile gibt es überhaupt?



Welche Probleme versuchen diese zu lösen?

Welche Herausforderungen gibt es?



Wie treffe ich eine gute Entscheidung?

EINE PLATFORM ENTSTEHT

Was klein startet, wird schnell groß

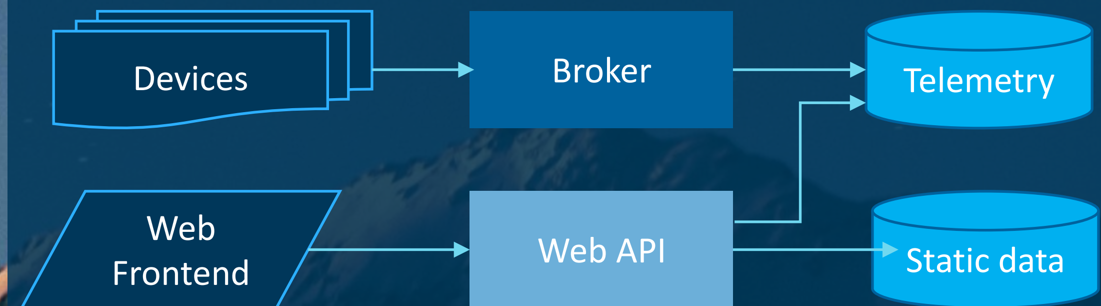
Eine Plattform soll industrielle Geräte verwalten

- Geräte senden Telemetriedaten
- Webanwendung für die Konfiguration
- Webanwendung für die Darstellung der Daten

Spätere Anforderungen

- KPIs sollen in Dashboards dargestellt werden
- Plattform soll Alarme generieren und über E-Mail benachrichtigen
- Daten sollen exportierbar sein, für weitere Auswertungen

Architektur



REST

```
GET        /devices
GET        /devices/{id}
POST       /devices
PUT        /devices/{id}
DELETE     /devices/{id}

GET        /devices/{id}/telemetry
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddOpenApi();

var app = builder.Build();
app.MapOpenApi();

app.MapGet("/devices", async (IDeviceRepository repo) =>
{
    var devices = await repo.GetDevicesAsync();
    return Results.Ok(devices);
})
.WithName("GetDevices");
```


REST FÜR RESSOURCEN

Geräte und Konfiguration klar modellieren

REST kommt aus der Web Architektur

- Von Roy Fielding beschrieben
- Oft genug falsch verstanden

Der zentrale Gedanke: Ressourcen

- Ressourcen werden über stabile URLs Pfade
- Clients arbeiten mit Repräsentationen dieser Ressourcen

HTTP liefert die gemeinsame Semantik

- HTTP-Methoden für CRUD (GET, POST, PUT, DELETE, PATCH)
- Response Status Codes (Ok, Bad Request, Internal Server Error)

OpenAPI liefert das Schema

- Welche Routen gibt es
- Welche Request und Response Types gibt es

REST & HATEOAS

```
GET        /devices/children
POST       /devices/prepareUpdate

Querying
GET        /devices
   ?page=1
   &pageSize=20
   &status=online
   &sort=name
```


```json
{
  "id": "dev-001",
  "name": "Press Line 1",
  "status": "online",
  "_links": {
    "self": "/devices/dev-001",
    "configuration": "/devices/dev-001/configuration",
    "telemetry": "/devices/dev-001/telemetry/latest"
  }
}
```


REST IN DER PRAXIS

Wenn CRUD allein nicht mehr reicht

REST APIs wachsen schnell über einfaches CRUD hinaus

- Listen, Details, Beziehungen, Status, Aktionen
- Unterschiedliche Clients brauchen unterschiedliche Sichten

OpenAPI schafft einen stabilen Vertrag

- Dokumentation, Client Generierung, Testbarkeit, Governance

Abfragen brauchen konsistente Regeln

- Pagination, Filtering, Sorting, Searching

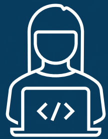
HATEOAS kann APIs navigierbar machen

Herausforderungen:

- Overfetching, Unterfetching
- Spezialendpunkte (/dashboard/mobile)
- Uneinheitliche Query Parameter

REST

Abschlussbewertung



Dev-Produktivität

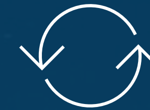
- Einfacher Einstieg, schwer zu meistern
- OpenAPI gibt guten Rahmen vor

Client Flexibilität

- Gut für bekannte Szenarien mit stabilen DTOs
- Wenig flexibel, wenn Clients unterschiedliche Daten benötigen

Performance & Skalierung

- Solide, besonders mit HTTP Caching Headers, Kompression und sauberen Endpunkten
- JSON ist gut debuggbar, aber nicht effizient



Evolution & Breaking Changes

- Versionierung etabliert über URL, Header oder Media Type
- Lässt sich über Schema Diffs, Contract Tests und Sunset Policies abdecken

Betriebsfähigkeit

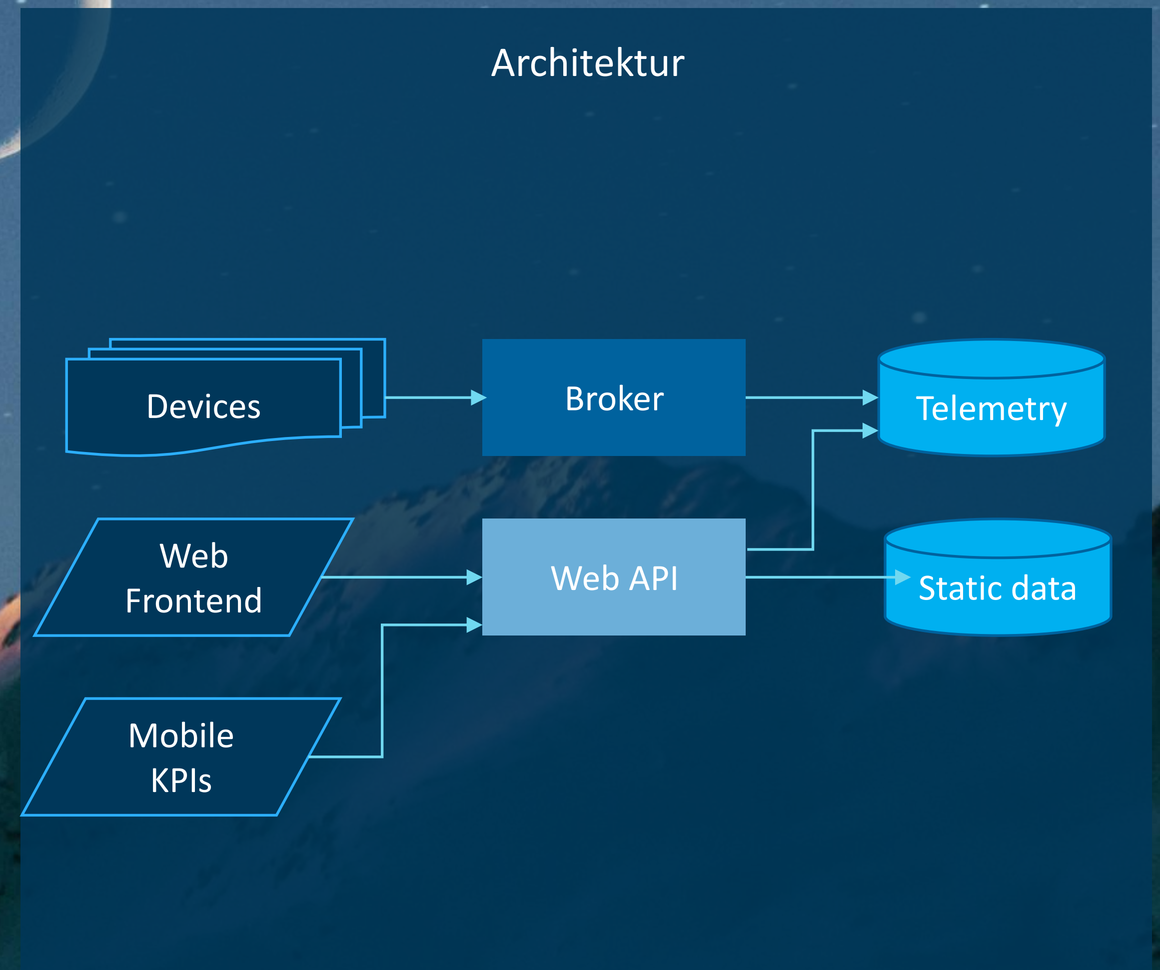
- Requests, Response, Status Codes, Latenz, Fehler sind gut in Logs sichtbar
- Debugging einfach weil Request und Response meist direkt lesbar

EINE PLATFORM WÄCHST

Neues Frontend entsteht

Neue Anforderung

KPIs sollen in einer separaten Mobile App in einem Dashboard dargestellt werden



GraphQL

```
POST        /graphql

query {
  devices {
    id
    name
    status
    site {
      name
    }
    latestTelemetry {
      temperature
      vibration
    }
}
```

```
mutation {
  registerDevice(
    input: {
      name: "Sensor A"
      location: "Werk 1"
    }
  ) {
    id
    name
    location
  }
}
```

```
builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>();

app.MapGraphQL();

public class Query
{
    public Task<IReadOnlyList<DeviceDto>> GetDevicesAsync(
        IDeviceRepository repo)
    {
        return repo.GetDevicesAsync();
    }
}
```

GRAPHQL QUERIES

Clients fragen gezielt Daten ab

GraphQL entstand aus Mobile Anforderungen

- Entwickelt von Facebook ab 2012
- Open Source seit 2015

Der zentrale Gedanke: Client beschreibt die Form der Daten

- Schema im Backend beschreibt die verfügbaren Daten
- Client bestimmt die gewünschte Datenform
- Response folgt der Struktur der Query
- Resolver laden die Daten im Backend
- Mutations beschreibt Änderung der Daten (Create, Update, Delete)

Wichtig:

- GraphQL ist kein Datenbank-Browser
- Business Logik bleibt im Backend
- Daten werden als DTO abstrahiert

GraphQL Query Complexity

```
query {
  devices {
    id
    name
    site {
      name
      devices {
        id
        telemetry(last: 1) {
          temperature
          vibration
          timestamp
        }
      }
    }
}
```

GRAPHQL GRENZEN

Flexibilität braucht klare Leitplanken

Client können sehr mächtige Queries bauen

- Tiefe Objektgraphen
- Verschachtelte Listen
- Teure Felder und Aggregationen

Performance muss aktiv geschützt werden

- Query Depth, Complexity Limits, Data Loader gegen N+1

Security wird feingranularer

- Berechtigung auf Typen und Feldern

Caching funktioniert anders

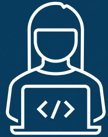
- HTTP-Caching ist schwieriger, Query Plan cachen

Operations braucht mehr Kontext

- Operation Name, Query Hash, Resolver Zeiten

GRAPHQL

Abschlussbewertung



Dev-Produktivität

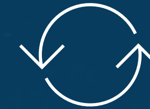
- Höhere Lernkurve
- Frontend Aufwand sinkt, da ein Query mehrere Calls ersetzt
- Backend Aufwand steigt durch Resolver Design, Data Loader, Query Limits

Client Flexibilität

- Sehr hoch, Clients wählen Felder, Datenform innerhalb des Schemas
- Gut für unterschiedliche UI-Varianten

Performance & Skalierung

- Performanter da im Zweifel weniger Daten
- N+1 Probleme und starke Flexibilität kann aber auch Performance kosten



Evolution & Breaking Changes

- Schema Evolution über additive Felder und Deprecation

Betriebsfähigkeit

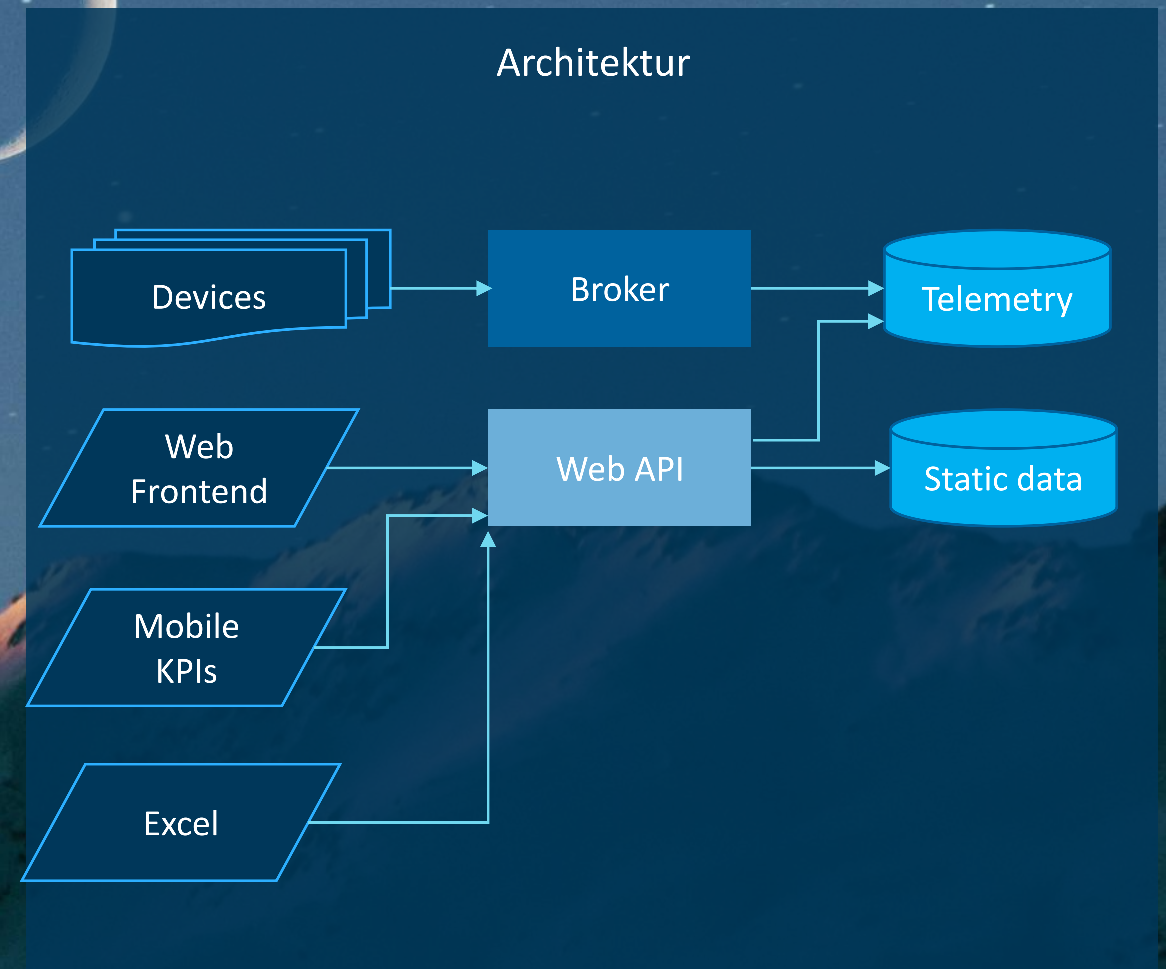
- Monitoring von Operation Name, Query Hash und Resolver Zeiten
- Debugging mit GraphQL Tools im Browser

ENTERPRISE-READY PLATFORM

Wo Daten sind, will auch ausgewertet werden

Neue Anforderung

Die Daten sollen auch in Excel und PowerBI eingebunden werden können.



OData

```
GET /odata/devices?
  $filter=status eq 'Offline'&
  $select=id,name,status,lastTelemetry&
  $orderby=name&
  $top=50
```

```
var modelBuilder = new ODataConventionModelBuilder();
modelBuilder.EntitySet<DeviceDto>("Devices");

builder.Services
    .AddControllers()
    .AddOData();

app.MapControllers();

public class DevicesController(IDeviceRepository repository) :
ODataController
{
    [EnableQuery]
    public IActionResult<IQueryable<DeviceDto>> Get()
    {
        return Ok(_repository.GetDevices());
    }
}
```


ODATA INTEGRATION

Daten standardisiert verfügbar machen

OData ist ein standardisiertes Daten-API-Modell

- OASIS und ISO/IEC Standard
- REST basierter Zugriff auf Datenmodelle

Der zentrale Gedanke:

- Daten über URLs bereitstellen
- Query Optionen standardisieren (filter, select, orderby, top, skip, expand)
- Metadata für Tools verfügbar machen über Entity Data Model (EDM)
- Create, Update, Delete über HTTP-Methods

Wichtig:

- Stabile Read Models definieren, keine internen Entities direkt veröffentlichen
- Query Limits und Berechtigung setzen

Odata und REST

```
GET /devices?status=offline&orderBy=name&top=50
   &fields=id,name,status

GET /odata/devices?$filter=status eq
'Offline'&$orderby=name&$top=50&$select=id,name,status
```


REST WIRD ODATA

Wenn Filter zum Dialekt werden

REST Filtering reicht, wenn

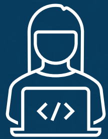
- Wenige fachliche Filter existieren
- Use Cases bekannt sind
- Performance gut kontrollierbar bleibt

OData prüfen, wenn

- Viele kombinierbare Queries entstehen
- Query Optionen ständig nachgebaut werden
- Excel, PowerBI oder ähnliches echte Zielclients sind

ODATA

Abschlussbewertung



Dev-Produktivität

- Effizient für Standard-Datenabfragen
- Wenig produktiv, wenn Team nur wenige fachliche Filter braucht
- Aufwand entsteht durch EDM Modell und Co.

Client Flexibilität

- Hoch für datenorientierte Clients
- Besonders Stark für Tools wie Excel, PowerBI und Power Query

Performance & Skalierung

- Effizient, wenn Queries begrenzt und Read Models sauber sind
- Riskant, wenn komplete Filter ungeprüft erlaubt werden

Evolution & Breaking Changes

- Additive Felder und neue Entity Sets sind meist gut handhabbar
- Breaking Changes eher schwierig und mit Versionierung verbunden

Betriebsfähigkeit

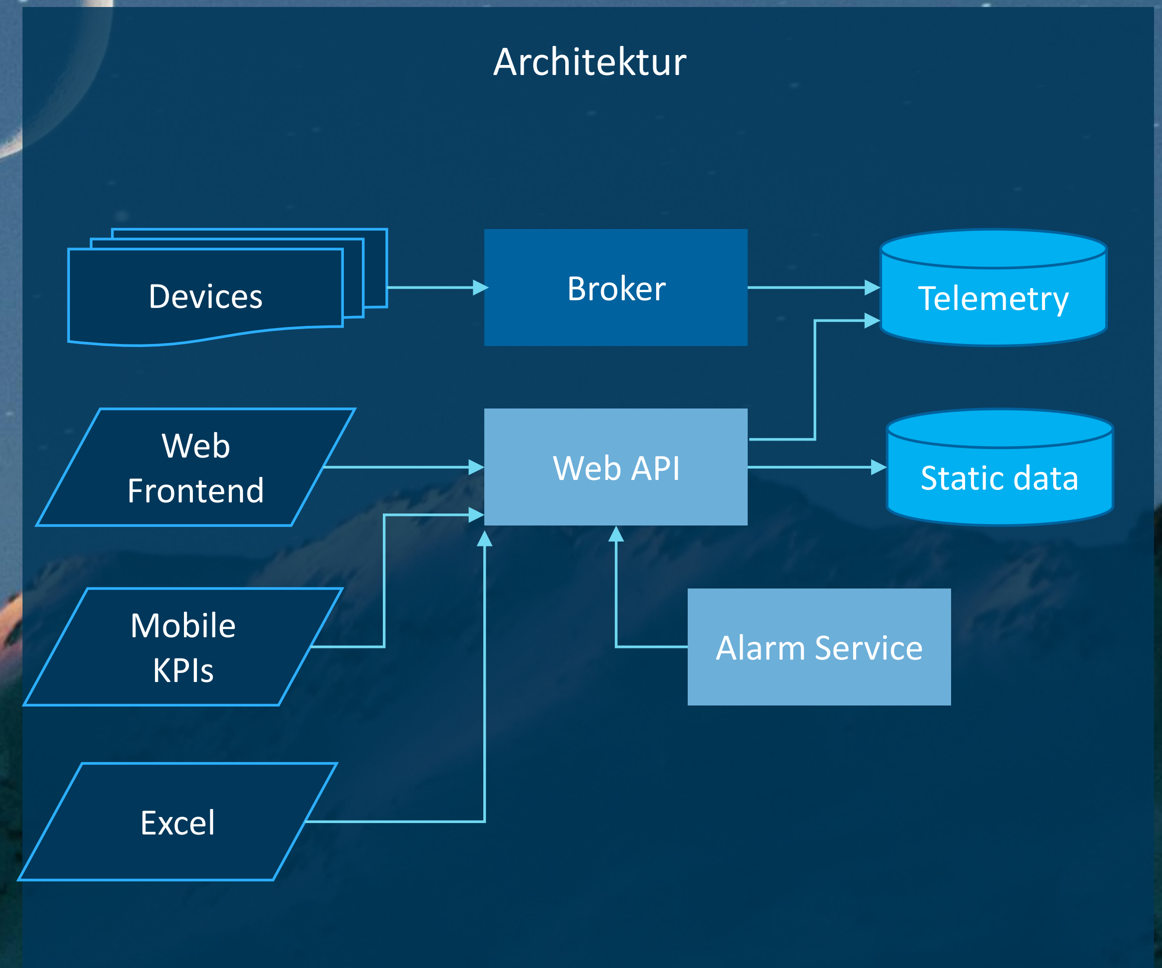
- Requests sind gut sichtbar
- Debugging auch einfach, da nutzbar in Excel

SERVICE TO SERVICE

Auch interne Services wollen miteinander sprechen

Neue Anforderung

Alarme sollen durch Trigger vom Broker evaluiert werden und E-Mail-Benachrichtigungen versenden.



gRPC

```
syntax = "proto3";

service DeviceService {
  rpc GetDevices (GetDevicesRequest) returns (GetDevicesReply);
}

message GetDevicesRequest {
}

message GetDevicesReply {
  repeated Device devices = 1;
}

message Device {
  string id = 1;
  string name = 2;
  string status = 3;
}
```


GRPC

Effiziente Kommunikation zwischen Services

gRPC ist ein Remote Procedure Call Framework

- Ursprünglich von Google veröffentlicht
- Basiert auf HTTP/2, nutzt typischerweise Protobuf

Der zentrale Gedanke:

- Services definieren Methoden
- Protobuf beschreibt den Contract
- Service und Client wird daraus generiert
- Kommunikation stark typisiert

gRPC

```
syntax = "proto3";

package devices.v2;

service DeviceService {
  rpc GetDevices (GetDevicesRequest) returns (GetDevicesReply);
}

message GetDevicesRequest {
}

message GetDevicesReply {
  repeated Device devices = 1;
}

message Device {
  string id = 1;
  string name = 2;

  DeviceStatus status = 3;
}

enum DeviceStatus {
  DEVICE_STATUS_UNKNOWN = 0;
  DEVICE_STATUS_ONLINE = 1;
  DEVICE_STATUS_OFFLINE = 2;
  DEVICE_STATUS_ERROR = 3;
}
```

GRPC GRENZEN

Nicht jedes Performanceproblem braucht gRPC

gRPC ist nicht automatisch die beste Performance Antwort

Browser und externe Clients sind anspruchsvoller

- gRPC nutzt HTTP/2, Protobuf und Message Contracts
- Für externe APIs braucht es ggf. gRPC Web, Transcoding oder REST Fassade

Debugging ist weniger direkt

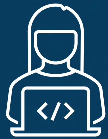
- Protobuf ist binär und nicht einfach im Browser lesbar
- Tools wie grpcurl, Postman oder DIE-Unterstützung werden wichtiger
- Logs sollten Requests fachlich verständlich machen

Versionierung braucht Disziplin

- Felder additiv ergänzen
- Feldnummern nicht wiederverwenden
- alte Felder nicht unkontrolliert entfernen

GRPC

Abschlussbewertung



Dev-Produktivität

- Interne Services erzeugen Server und Client Code
- Manuelle Tests eher schwieriger als über REST (grpccurl oder ähnliches)

Client Flexibilität

- Wenig flexibel, Server definiert fix die Methoden und Messages
- Nicht direkt im Browser nutzbar

Performance & Skalierung

- Sehr stark durch HTTP/2, Protobuf, geringe Payloads und Streaming

Evolution & Breaking Changes

- Evolution über Protobuf Schema Änderungen
- Breaking Changes über Package Versioning

Betriebsfähigkeit

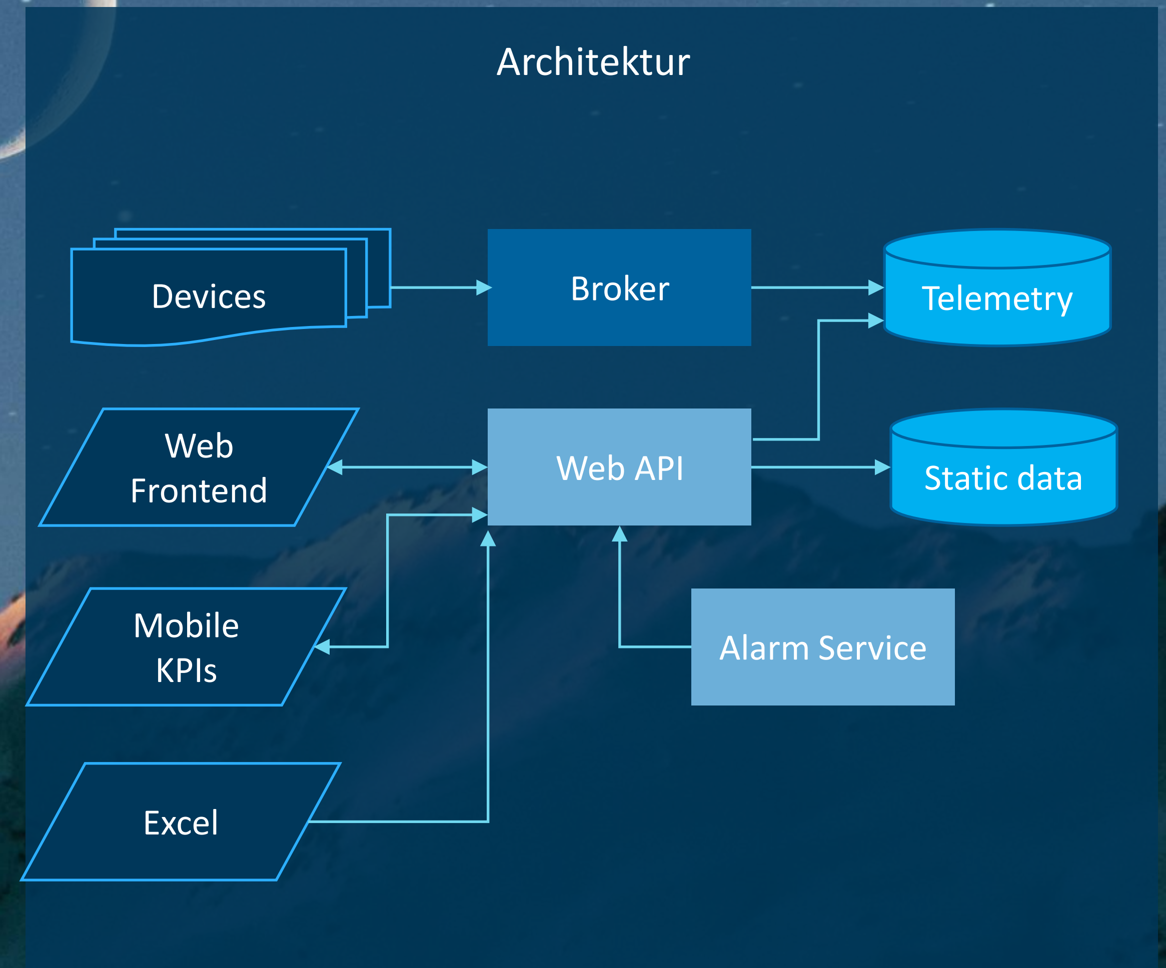
- Messbar über Methodenname, Status Code, Latenz
- Debugging schwieriger, da Payloads binär sind

LIVE UPDATES

Polling ist keine Option

Neue Anforderung

Das Dashboard im Frontend soll Live mit neuen Werten aktualisiert werden, ohne dass die Clients pollen müssen.



Web Sockets & SSE

```
builder.Services
    .AddGraphQLServer()
    .AddQueryType<Query>()
    .AddSubscriptionType<Subscription>()
    .AddInMemorySubscriptions();

var app = builder.Build();
app.UseWebSockets();
app.MapGraphQL();


public class Subscription
{
    [Subscribe]
    public IEnumerable<Device> DeviceStatusUpdated(
        [EventMessage] IEnumerable<DeviceStatusDto> devices)
    => devices;
}

public class DeviceService(ITopicEventSender sender)
{
    public Task UpdateStatus()
        => _sender.SendAsync(
            nameof(Subscription.DeviceStatusUpdated),
            new DeviceStatusDto("online"));
}
```

LIVE UPDATES

Nicht alles ist Request Response

Polling ist der einfache Start

- Client fragt regelmäßig nach
- Wird bei vielen Clients schnell ineffizient

Es gibt mehrere Streaming Stile

- **SSE** für Server zu Client Updates
- **WebSockets** für bidirektionale Kommunikation
- **Long Polling** für Legacy
- **GraphQL Subscriptions** für Live Updates im GraphQL Modell

AsyncAPI als Spezifikation für Event Driven APIs

Web Sockets & SSE

```
builder.Services.AddSignalR();

app.MapHub<DeviceHub>("/hubs/devices");

public class DeviceHub : Hub
{
    public async Task SubscribeToSite(string siteId)
        => await Groups.AddToGroupAsync(Context.ConnectionId,
siteId);
}

await hubContext
    .Clients
    .Group(siteId)
    .SendAsync("DeviceStatusChanged", update);
```

SIGNALR

Abstraktion für bidirektionale Echtzeit

SignalR abstrahiert Echtzeit Kommunikation

- Server kann Nachrichten aktiv an Clients senden
- Clients können Methoden am Server aufrufen
- Hubs strukturieren die Kommunikation

SignalR nutzt passende Transports

- WebSockets, wenn verfügbar
- Alternative Transports, wenn nötig
- Automatische Verbindungsverwaltung

Worauf man achten sollte

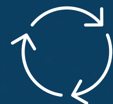
- Skalierung über mehrere Instanzen
- Auth und Reconnect Verhalten
- Backpressure und langsame Clients
- Monitoring aktiver Verbindungen

ENTSCHEIDUNGSKRITERIEN

Wie entscheide ich mich denn jetzt?

Consumer

Wer nutzt die API?



Kommunikationsmodell

Wie sollen die Daten fließen?

Daten & Schnittstelle

Wie stabil oder flexibel müssen Daten und Schemata sein?

Nicht funktionale Anforderungen

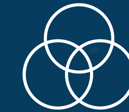
Was muss optimiert werden?

Betriebsrealität

Was kann das Team langfristig tragen?

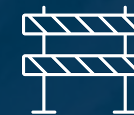
RECAP

API-Stile lösen unterschiedliche Probleme



Tradeoffs sind wichtiger als Hype

Standards verhindern eigene Sonderwege



Gemeinsame Logik hält mehrere APIs wartbar

LUNARIS
Digital Solutions

LUNARIS
Digital Solutions