# Backend, APIs & Server Side Rendering

# Agenda

Einführung

API Konzepte

NodeJS Backend & Server Side Rendering

Backend Design

Backend Tooling

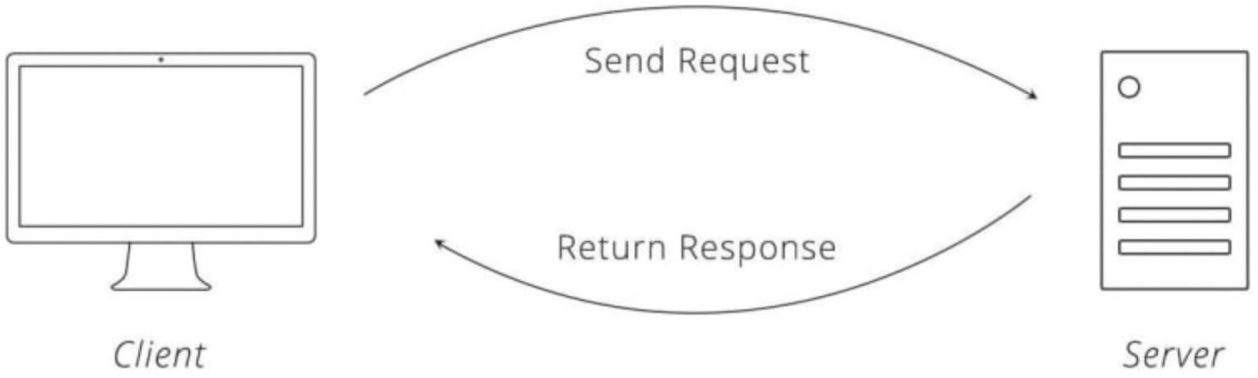
Backend Infrastruktur

# Einführung Backend

# Frontend vs. Backend

- Frontend: Konzentriert sich auf das, was der User sehen kann
- Backend: Alles, was die Applikation im Hintergrund zum Funktionieren bringt
- Veranschaulichung wie ein Auto:
  - Frontend ist das Äußere vom Auto
  - Backend sind die Maschinen im Inneren des Autos
  - Das Auto funktioniert nur, wenn die Maschinen im Inneren auch funktionieren
  - Bestimmte Aspekte des Äußeren können dazu beitragen, dass das Auto besser funktioniert (z.B. schneller fährt)
  - Das Äußere und das Innere müssen einheitlich gestaltet werden, damit das Auto am besten funktioniert

# Request / Response Kreislauf



# APIs (Application Programming Interface)

# APIs

- **Generell:** Schnittstellen zwischen allen möglichen Anwendungen (z.B. Vue bietet eine API zum einfachen Erstellen von Komponenten)
- **Im Web Backend Kontext:** Server, die Endpunkte zum Abfragen oder Verändern von Daten zur Verfügung stellen
- CRUD: Akronym für vier grundsätzliche Datenoperationen
    - Create
    - Read
    - Update / Change
    - Delete

# HTTP (HyperText Transfer Protocol)

- Ein Kommunikationsprotokoll mit einem definierten Set aus Standards, um Daten von APIs zu konsumieren
- Beispiele:
  - Website aufrufen
  - Video streamen
  - Smart Home Aktionen
  - …

# HTTP Methoden

- HTTP Verb für jeden Request erforderlich
  - GET: Abfragen von Ressourcen
  - POST: Anlegen von Ressourcen
  - DELETE: Löschen von Ressourcen
  - PUT: Update kompletter Ressourcen / Anlegen neuer Ressourcen mit vom Nutzer festgelegter ID
  - PATCH: Ändern von Ressourcen

# HTTP Request + Response

Request:

- Kombination aus Methode und API URL bilden die eindeutige Identifizierung für den Request
  - GET:/api/dogbreeds → Abfragen einer Liste
  - GET:/api/dogbreeds/1 → Abfragen einer sepzifischen Hunderasse
  - POST:/api/dogbreeds → Anlegen einer neuen Hunderasse

Response:

- Beinhaltet z.B. Status Code, Status Text und Response Body
  - 200 OK
  - 201 Created
  - 400 Bad Request
  - 401 Unauthorized
  - 404 Not Found
  - ...

# HTTP URL Query Parameters


Protokoll Domain API URL Query Parameters

- Fragezeichen markiert den Beginn der Query Parameter
- Kaufmännisches “Und” trennt Query Parameter
- Gleichheitszeichen trennt Parameter Schlüssel und Parameter Wert

# HTTP Request Headers

- Stellen Informationen zum Request Kontext zur Verfügung
  - (Body Content, Autorisierung, Plattform…)

```
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101
Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
```

# HTTP Request Body

- Kann verschiedene Formate haben (z.B. JSON, XML, HTML Form Data)
- Typ wird mit Header “Content-Type” festgelegt

# REST (Representational State Transfer)

- Nutzt das HTTP Protokoll
- Ist kein Standard, sondern ein Architekturstil für APIs
- Ressourcen:
  - Zentraler Baustein von REST Architekturen
  - Jede Ressource ist eindeutig identifizierbar
  - z.B. Bild, Website, aber auch abstrakte Modelle wie Filme mit Information in einer Datenbank
- Repräsentationen:
  - Server muss den Zustand der Ressourcen verwalten
  - Client gibt an, mit welchem Repräsentationstyp er mit den Ressourcen interagieren möchte
- Einheitliches Interface
  - Jeder Client muss mit den gleichen Operationen mit den Ressourcen interagieren
- Zustandslosigkeit
  - Alle API Aufrufe dürfen keine geteilten Zustände benutzen

# OpenAPI

- Spezifikation eines Standards für HTTP APIs
- Einheitliche API Spezifikation und Dokumentation via YAML oder JSON
- Jeder User kann die API Spezifikation einsehen und direkt verstehen, wie die API funktioniert
- Programmiersprachenunabhängig
- → Leichte Integration von APIs mit anderen Services
- Swagger Toolset
    - Editor für das Bearbeiten der API Spezifikation
    - Codegen für das Generieren von Server und Client Code basiert auf der Spezifikation
    - UI für die Visualisierung der Spezifikation und Dokumentation
    - ...

# GraphQL

- API Query Sprache und Spezifikation
- Nur ein Endpoint in HTTP, der Queries entgegennimmt
- Vergleich zu HTTP Methoden:
  - Query zum Lesen von Daten
  - Mutation zum Ändern von Daten
  - Subscription für das Streaming von Daten

# NodeJS Backend

# NodeJS vs. JavaScript im Browser

NodeJS:

- Kann Backend und Frontend Code (Server Side Rendering)
- Typische Nutzung:
  - Dateioperationen
  - DB Operationen
- Der Entwickler kann die NodeJS Version frei wählen

JavaScript im Browser:

- Kann nur Frontend Code ausführen
- Typische Nutzung:
  - DOM API
  - Cookies
  - fetch API
- JavaScript Version nicht kontrollierbar, da die User verschiedene Browser und Versionen nutzen

# NodeJS Backend

- NodeJS läuft in einem einzigen Prozess, ohne neue Threads zu eröffnen
- Asynchrone Codestruktur ermöglicht gleichzeitige Ausführung von Operationen
- Durch NodeJS können auch Frontend-Entwickler Backend Code entwickeln
- Viele Standardbibliotheken für die Interaktion mit Web Technologien (z.B. HTTP)

Übung: Schreibe einen NodeJS Server, der einen Endpunkt publiziert und einen Query Parameter zurückgibt und führe das NodeJS Programm aus. Rufe die publizierte URL im Browser auf und teste, ob das Ergebnis korrekt angezeigt wird.

Tipp: Benutze NodeJS Dokumentation als Hilfe (https://nodejs.org/en/learn/getting-started/introduction-to-nodejs)

# NodeJS Frameworks

- Verschiedenste Framework für NodeJS sind verfügbar
- Fokussieren sich auf unterschiedliche Bereiche
- Erleichtern grundsätzlich die Erstellung von komplexen Backend APIs
- Beispiele:
  - Express.js — Benutzerfreundlichkeit und Leichtgewichtigkeit
  - Meteor.js — Support von Frontend Frameworks + Mobile
  - Koa.js — Modularität und Fehlerbehandlung
  - Next.js — React Kompatibilität

# Express.js

- Integriertes Routing
- Request und Response Objekte werden geparsed zur Verfügung gestellt
- Kein Boilerplate Code notwendig
- Einfach zu lernen
- Weniger geeignet für große und komplexe Anwendungen

Übung: Schreibe ein ExpressJS Backend mit einem POST und einem GET Endpoint, starte den Express Server und rufe beide Endpoints in dieser Reihenfolge auf, sodass die Daten, die in den POST Request übergeben wurde, auch wieder vom GET Request zurückgegeben werden. (Tipp: Daten in einer Datei ablegen).

Tipp: Benutze ExpressJS Dokumentation als Hilfe (https://expressjs.com/en/starter/hello-world.html)

# Server Side Rendering

- Code zum Laden der Website wird auf dem Server ausgeführt
- Verwendung einer Template Sprache, um JavaScript Code in HTML einzubinden
- Komplexere Interaktion mit Frontend APIs (DOM, window, …)
- Zustandsverwaltung wird komplizierter, da der Code nicht mehr nur im Browser ausgeführt wird
- Website wird im Browser schneller geladen
- Auf dem Server wird mehr Last erzeugt

# Backend Design

# Monolithische Architektur

- Ein großes Repository
- Ein großer Tech Stack
- Kaum Entkopplung von Software möglich
- Vorteile:
  - Nur eine Technologie (mehr Know-How in den Team(s))
  - Insgesamt einfachere Struktur (nur einen Testing Prozess, nur eine Deployment Pipeline, nur ein Debugging Setup, …)
- Nachteile:
  - Kann schnell sehr unübersichtlich und komplex werden
  - Keine unabhängigen Code Updates
  - Schwierigere Einarbeitung
  - Keine modulare Skalierbarkeit
  - Wechsel der Technologie fast unmöglich

# Microservices Architektur

- Ein Service ist für einen gebundenen Kontext zuständig (Domain Driven Design)
- Eine Service wird nur von einem Team entwickelt
- Services sind modular aufgebaut und ersetzbar
- Services sind isoliert (ggf. Komplett unterschiedlicher Tech Stack)
- Jeder Microservice hat seine eigene Datenbank (doppelte Daten werden doppelt implementiert, weil ggf. andere Anforderungen an die Daten gestellt werden)
- API Gateway ist notwendig, um Services zu bündeln

# Backend Tooling

# Datenbanken

- Werden genutzt, um alle möglichen Daten zu speichern, die gespeichert werden müssen
- Beispiel Webshop:
  - Produktdaten
  - Bestellungsdaten
  - Benutzerdaten
  - Inventardaten
  - ...
- Werden üblicherweise über APIs angesteuert

# Cache

- Oft genutzte Daten werden häufig in einem Cache gespeichert
  - Verringerung der Last auf die Datenbank
  - Minimierung der DB Zugriffe
  - Schnellere Ladezeiten von Websites
- Beispiele: Software Redis (In-Memory Cache)
- Komplexität:
  - Der Cache muss regelmäßig invalidiert werden
  - → Vermeidung von Anzeigen veralteter Daten
  - Cache muss regelmäßig überwacht werden
  - → Vermeidung von Ineffizienz

# Search Engines

- Bieten einen Weg schnelle Suchanfragen über große Datenmengen zu tätigen
- Spezialisiert auf Volltextsuche
- Beispiel Use Case Webshop:
  - Webshop integriert neue Produkte
  - Produkte werden von der Search Engine indiziert (unstrukturierte Daten)
  - Benutzer schickt einen Suchauftrag nach Produkten
  - Search Engine sucht nach passenden Daten basiert auf Benutzereinstellungen
  - Search Engine liefert relevante Ergebnisse
- Eingebaute Algorithmen für das Erkennen von Relevanz
- Eingebauter Support für Filtermethoden
- Produktbeispiel: ElasticSearch

# Job Queues

- Für die Ausführung von länger laufenden Prozessen, wenn ein Endpoint aufgerufen wird
- Vermeidung von langen Laufzeiten für den Benutzer
- Prozess wird in eine Queue geschrieben
- Queue wird sequentiell abgearbeitet
- Benutzer bekommt keine Info über Ergebnisdaten / erfolgreichen Abschluss des Prozesses

# Backend Infrastruktur

# Compute Infrastruktur (Website & API Hosting)

- Hardware Server:
  - Viel Wartungsaufwand
- Virtuelle Maschine:
  - auf einem Hardware Server können viele virtuelle Maschinen betrieben werden
  - Eine VM beinhaltet ein komplettes Betriebssystem → erfordert viel Leistung
- Container in virtueller Maschine:
  - Kein komplettes Betriebssystem, nur notwendigste ausführbare Dateien in einer Laufzeitumgebung
  - → Minimaler Fußabdruck
  - Kaum Administration notwendig (ein Container für eine Anwendung)
- Pods (Kubernetes):
  - System zum Betreiben von containerisierten Applikationen
  - Optimierung von Konfigurierbarkeit und Skalierbarkeit von Containern
- Serverless (Cloud)
  - “API as a Service”
  - Konfigurierbarkeit von Endpunkten, für die serverless Funktionen definiert werden können
- Webhosting Service

# Weitere Infrastrukturkomponenten

- Storage
- Network
- DNS (Domain Name Service)
- Backup
- Monitoring