HOCHSCHULE
RAVENSBURG-WEINGARTEN

Teil 1: Überblick und grundlegende Konzepte

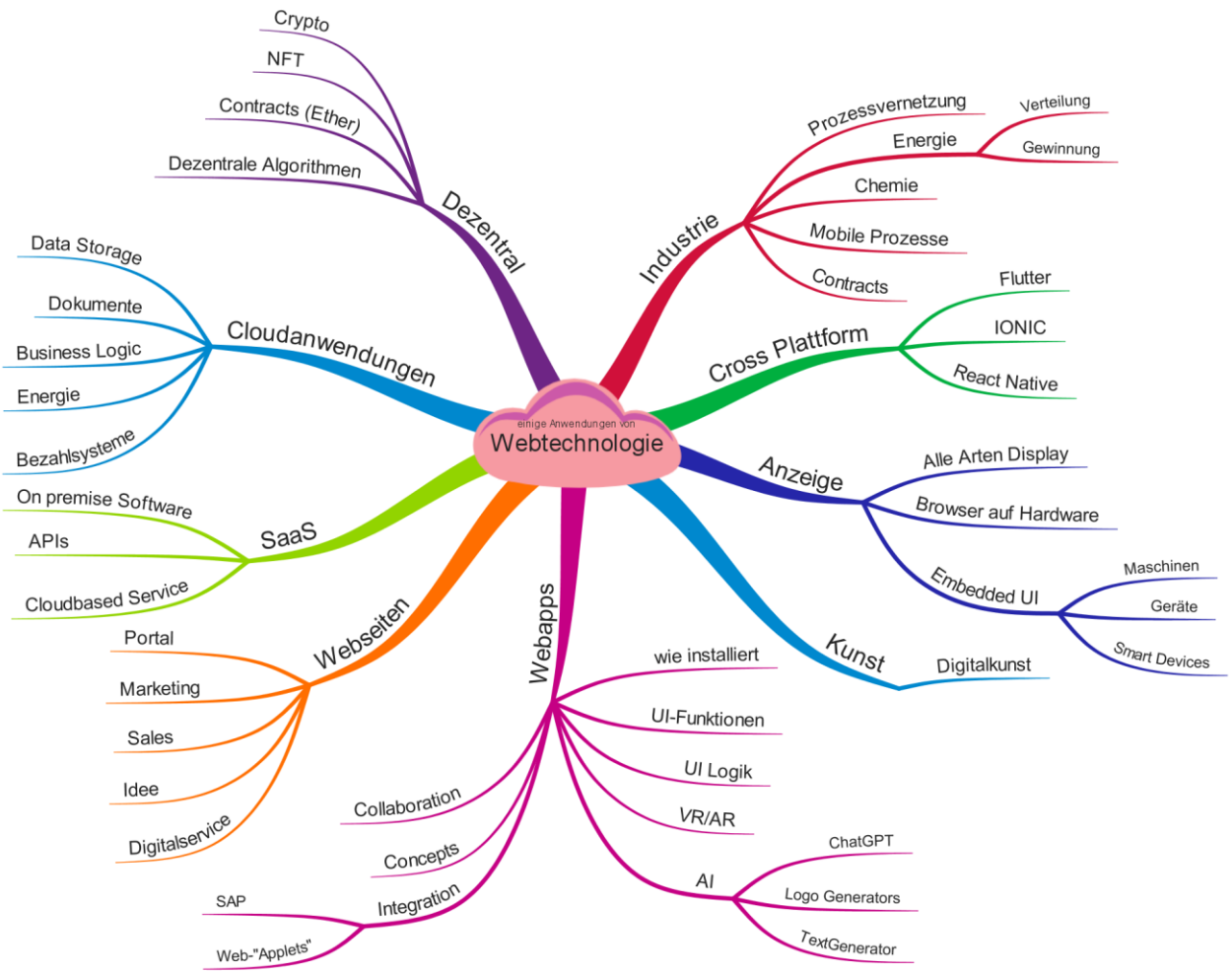
Prof. Dr.-Ing. Thorsten Weiss

Version 1.0

WEBENTWICKLUNG 2
one step ahead

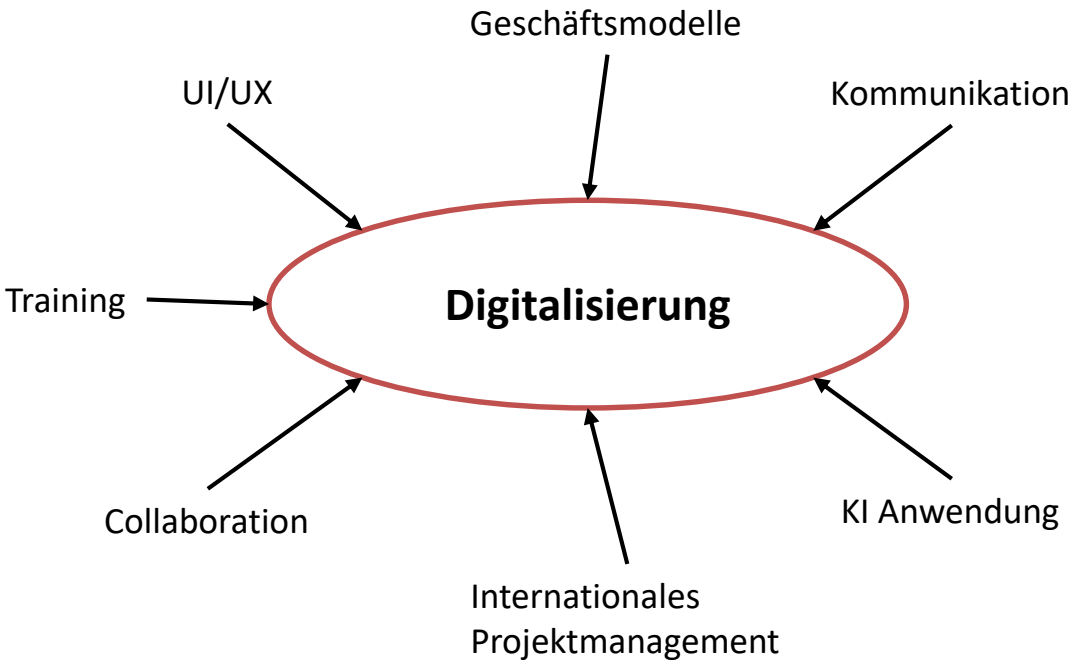
Neue und etablierte Anwendungen von Webtechnologen.

- Webtechnologie wird derzeit in nahezu **allen Branchen massiv ausgebaut.**
- Die Möglichkeiten und der Bedarf sind derzeit enorm. Es **bedarf Erfahrung und fundiertes Technologiewissen**, um digitale Geschäftsmodelle erfolgreich umzusetzen.
- Viele Branchen entdecken erst jetzt die Vorteile der **Webtechnologie, was enorme Marktchancen bringt.**



Digitalisierung und Webtechnik

- Digitalisierung wird oftmals in einem Atemzug **mit Webtechnologie** genannt.
- Digitalisierung ist **allerdings mehr als eine Webseite oder eine App.**
- Folgende Elemente fließen in digitale Produkte und Dienstleistungen ein.



Web 1.0

Die Inhalte waren meist statisch Inhalte. Dynamische Inhalte musste mit aus heutiger Sicht sperrigen Tools

Struktur

HTML

Legt die Struktur einer Seite fest. Zu Beginn waren Texte und Links (Bilder, Hyperlinks…) meist fest im HTML-Code vereint.

Herausforderungen:

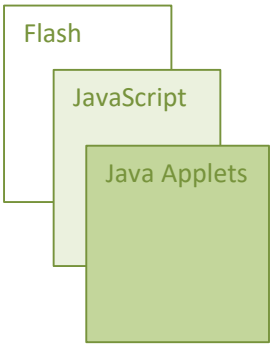
- Browserabhängigkeit.
- Hohe Unflexibilität
- Security
- Hohe Entwicklungszeiten
- Daten (Texte) mit Struktur und Design verschmolzen.

Eigenschaften bzw. „Design“

CSS

Legt fest, welche Eigenschaften HTML-Elemente haben sollten. Beispiele: Styling, UI-Verhalten wie Hover.

Anzeigelogik Interaktion



Falls die Webseite auf Nutzer-Interaktion, die über Links hinausgehen reagieren soll, wird die Logik der Seite mit verschiedenen Techniken umgesetzt.

Monolithischer Server

Eine Serversoftware erledigte alles.

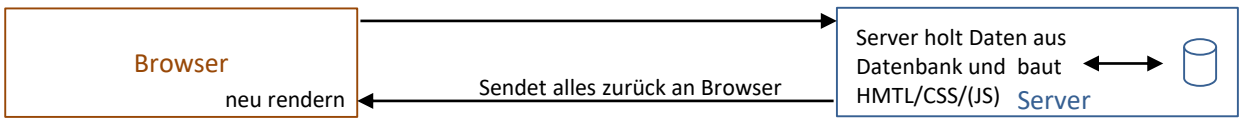
Größte Herausforderungen damals:

- Browserabhängigkeit
- Serverkapazitäten
- Netzausbau (teilweise Modem oder ISDN)
- Hohe Unflexibilität
- Security
- Hohe Entwicklungszeiten
- Keine Webentwickler am Markt
- Niemand wusste, was eCommerce und das Internet möglich machte

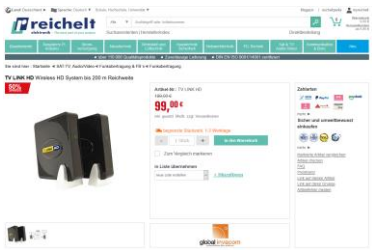
- Daten (Texte) mit Struktur und Design verschmolzen
- Mehrsprachigkeit
- Sehr eingeschränkte UI-Logiken
- Rechenpower und Paketgrößeneinschränkungen.

Web 2.0

- Seit Web 2.0 beginnt die Welt **der Wikis, Foren**, **Spielen und dynamischen Inhalten**. Die Technologien um **Server Side Rendering** wurden weiterentwickelt. Je nach Anforderungen einer Seite, wurden HTML/CSS/ggf. JS damit dynamisch erzeugt. Die allermeisten Seiten arbeiten heute so. Beispiel verschiedene Sprachen bei Wikipedia.
- Hier wurde verschiedene Technologien **im Frontend** (das, was im Browser läuft) und **im Backend** (was auf dem Server läuft) massiv erweitert. Ein Framework löst bis heute das nächste ab, jedoch zeichnen sich grundlegende Technologien als recht stabil ab.
- Um der Seite zu etwas Logik zu verhelfen wurde und wird bis heute oftmals auf PHP gesetzt.
- *Anmerkung: Trotz der hohen Verbreitung hat PHP Schwächen in Bezug auf die Entwicklungsgeschwindigkeit und Performance und Skalierbarkeit. Für neue Anwendungen würde ich es nicht empfehlen, da es deutlich modernere Werkzeuge gibt.*
- Dennoch gibt es zahlreiche etablierte Frameworks, die auf PHP basieren, wie Wordpress (mit allen Plugins wie Woocommerce), Joomla, Typo 3 und weitere große CMS (Content-Management-Systeme). Darum hält es sich noch recht stabil im Markt.
- **Beispiel: Shopsystem:**
  → Aufruf der Seite



Beim Klick auf ein Produkt wird vom Server eine neue Seite zusammengebaut und ausgeliefert. Der Browser zeigt diese Seite dann an. Man erkennt dies am kurzfristigen Neuwahlen.

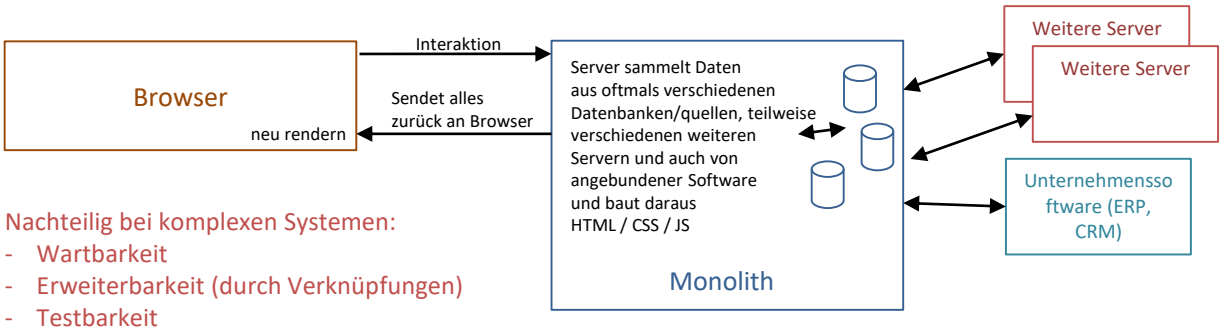


- Diese Vorgehensweise funktioniert im Grunde gut, aber sie braucht viele Ressourcen auf dem Server, da ständig viele neue Seitenteile generiert werden. Gut besuchte Seiten haben **Hunderte oder Tausende Nutzer** gleichzeitig, was enorme Serverkapazitäten bedarf. Es muss **dann immer die volle Serverpower bereit stehen**, was **zu hohen Infrastrukturkosten und enormen Energieverbräuchen** führt.
- Wenn der Nutzer etwas eingeben soll, wie beispielsweise seine Adresse in einem Shop, dann werden die Daten über sogenannte **Forms / Submits** an den Server geschickt, der diese entgegennimmt, verarbeitet und wieder eine ganze Reihe an HTML, CSS und
- **Dynamische Inhalte** (Animationen, direkte Nutzereingaben etc.) müssen mit **zusätzlicher Logik** ausgestattet sein, was heute mit Javascript umgesetzt werden.
- Nachteilig ist hierbei, dass die sogenannten **Anliegen** (Texte, die Preise, Bilder etc.) **nicht sauber getrennt** sind.

- Mitte der 2000er erleichterten Frontend-Frameworks wie **jQuery** schließlich, das Problem mit den verschiedenen Browsern zu lösen.
  Anmerkung TW: Heute ist dieses Framework immer noch im Einsatz aber in meinen Augen vollkommen outdated.
- Früher hatte man immer das Problem, dass Firefox, Internet Explorer, Chrome und Opera die selben HTML/CSS Dateien verschieden darstellten. Darum sahen Seiten auf einem Browser wie gewünscht aus und auf anderen anders. Dies führte dazu, dass **Webportale mit viel Funktion nur mit enormem Aufwand realisierbar waren**. eBay zum Beispiel musste hier sehr innovativ sein, genau wie Amazon und andere frühe Internetgrößen.
- **Die UI/UX war zudem stark eingeschränkt**, da die Umsetzung von UI-Logik für alle Plattformen und Browser sehr aufwendig war.
- Wirklich reaktive Elemente wurden erst mit den **Single Page Apps** (SPA) einfacher möglich.

Monolith:

- Ein Monolith fasst zahlreiche Funktionen in einer großen Serversoftware zusammen.
- Verschiedene Services wie Authentifizierung, Ausliefern, Anfragen vom Frontend und Apps werden in dem **einen großen Projekt** zusammengefasst und laufen auf einem Server.

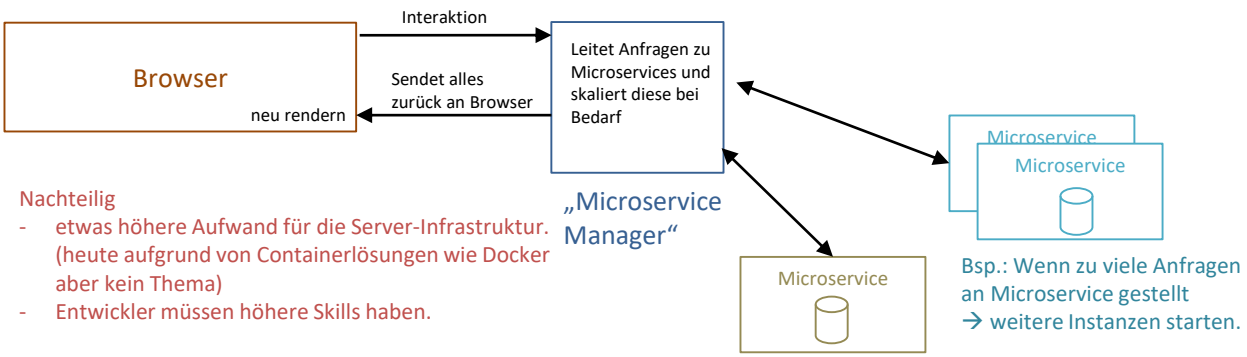


Nachteilig bei komplexen Systemen:
- Wartbarkeit
- Erweiterbarkeit (durch Verknüpfungen)
- Testbarkeit
- Auf ein Tool festgelegt (z.B. Java Server)
- Keine Skalierung

Vorteil: Bei wenig komplexen Servern kann die Struktur ausreichen.

Microservice:

- Heute kommt mehr und mehr eine **Architektur mit Microservices** zum Einsatz. Das sind kleine Serverprogramme, die jeweils für **eine klar abgegrenzte Aufgabe** zuständig sind. Sie werden **getrennt entwickelt, upgedated, gewartet, getestet, erweitert und kaskadiert.**
- Grundlegende Idee ist es, kleine Softwarepakete zu schreiben, diese für sich **zu testen** und **anpassbar** an zukünftige **Anforderungen** zu machen. So kann **ein Entwickler(team)** einfach an **„seinem" Microservice** arbeiten, ohne andere Aufgaben zu beeinträchtigen. Zudem können einzelne Microservices ein Update bekommen und
- Manchmal eignen sich für **verschiedene Aufgaben** verschiedene Systeme und verschiedene Programmiersprachen. Dies ist bei Microservices möglich.
  Bsp.: Big Data mit Python, IoT mit Java, schnelle REST-Server mit Node.
- Bei Wartungsarbeiten **wird immer nur ein Teil der Seite nicht verfügbar sein** (wenn es gut gemacht ist).
- **Skalierbarkeit: Alle Services können in mehreren Instanzen bei Bedarf gestartet werden und so skalierbar sein. Das spart enorme Kosten und enorme Energie.**



Nachteilig
- etwas höhere Aufwand für die Server-Infrastruktur. (heute aufgrund von Containerlösungen wie Docker aber kein Thema)
- Entwickler müssen höhere Skills haben.

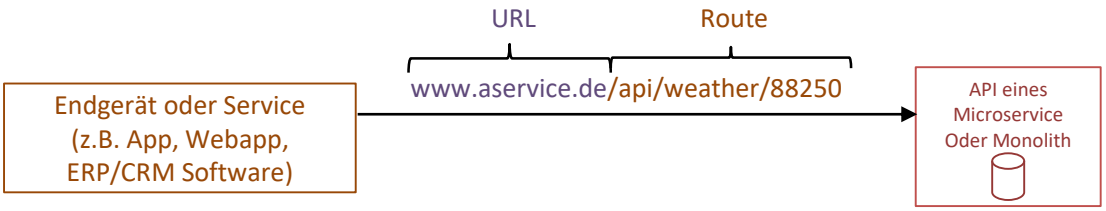
Vorteile:
- Wartbarkeit
- Erweiterbarkeit (durch Verknüpfungen)
- Testbarkeit
- Auf ein Tool festgelegt (z.B. Java Server)
- Skalierbarkeit

„Serverless“ Functions

- Serverless Functions werden **über URL oder andere Methoden aufgerufen**.
- Sie sind also Funktionen, die **quasi direkt aus dem Internet aus aufgerufen** werden können.
- In **Wirklichkeit** läuft ein **Serversystem** im Hintergrund, das vom Nutzer aber nicht eingerichtet und gehostet werden muss.
- Diese Services werden häufig **nach der Dauer oder Intensität der Nutzung bezahlt**, was ein hochskalierbares System ermöglicht. Viele Services sind so bereits enthalten, was häufig über Infrastrukturkosten bezahlt wird. Beispiele sind AWS (Amazon Webservices), Google Firebase und viele weitere Anbieter.
- Durch entsprechende **Containerlösungen wie Docker** sind auch der Betrieb eigener Plattformen für Serverless Functions denkbar.

API (Application Programming Interface)

- Eine API ist eine Schnittstelle (hier auf einem Server), der Daten bereit stellt, den eine App oder eine Webapplikation anzeigen kann oder ein anderer Service verarbeiten kann.
- Eine API hat meist eine URL oder eine IP-Adresse und stellt ihre Daten über Routen zur Verfügung.
- Gute APIs sind gut dokumentiert.



SaaS (Software as a Service)

- SaaS bestehen meist aus einer webbasierten Software und einem Geschäftsmodell (im B2C oftmals Freemium)
- SaaS können Frontends enthalten und so Software zur Verfügung stellen.
- Bekannte Beispiele sind Figma, Jira, Google Maps, Google Docs, Bitbucket und unzählige weitere Tools, die in der Cloud laufen.

- Manche SaaS (Software as a Service) bestehen oder beinhalten nur eine API. SAP geht beispielsweise den Weg, ein SaaS zu werden und Frontends separat anzubieten oder entwickeln zu lassen. Die Firmenlogik läuft im Hintergrund.

Gängige Serversysteme

WEBENTWICKLUNG 2
one step ahead

- Heutzutage sind **unzählbare Hoster mit verschiedenen Services** am Markt.
- Prinzipiell bieten diese Hoster verschiedene Server-Technologien an. Folgende Systeme sind besonders häufig anzutreffen.
- **Welche Technologie für welchen Einsatzzweck genutzt wird, ist eine gut zu durchdenkende Fragestellung zum Projektstart.**

outdated

Linux mit Apache-Server mit PHP, mySQL (LAMP)

- PHP ist eine weitverbreitete **synchrone Skriptsprache**, die um OOP-Elemente erweitert wurde.
- Wichtige Frameworks, die häufig auftretende Problemstellungen adressieren: Laravel, Symfony (Einarbeitung aufwendig)
- Wird von den meisten Hostern angeboten.
- mySQL ist ein weitverbreitetes Datenbanksystem
- Apache ist eine Servertechnologie.

Java Server (Swing)

- Serversoftware in **Java** geschrieben.
- Für **große Projekte** häufig im Einsatz.
- Basiert auf Apache.
- Viele Frameworks verfügbar.

nginx

- Webserver, derzeit bei ca. 2/3 der großen Webseiten im Einsatz
- Routet Anfragen an Server.

Node.js

- Node.js ist ein JavaScript Runtime basierend auf der Chrome V8 Engine.
- Node.js ist ereignis-getriggert, damit asynchron.
- Non-blocking I/O model.
- Lightweight.
- Komponentenbasiert.
- Schnelle Einarbeitung und viele nützliche Pakete erhältlich.
- Für Webserver (EJS/JADE), REST und Sockets sehr schnell in der Umsetzung.

Python

- Django bietet Serverapplikationen in Python

Serverless Functions

- Direkte API gelinkt zu einer Funktion.
- Server ist abstrahiert und der Entwickler muss sich darum nicht kümmern.
- Bsp.: Firebase, AWS

NodeJS

- Node.js wurde von **Ryan Dahl** entwickelt. Die Idee war es, ein sehr schlankes Serverprogramm zu schreiben und nahezu alle Funktionalitäten in Packages auszulagern.
- Nach mehrmaligen Trennen und Zusammenführen wird Node.js heute von der **Node.js Foundation** gepflegt.

- Mit NodeJS lassen sich **einfache Kommandozeilenwerkzeugen bis hin zu komplexen Webanwendungen** umsetzen.
- NodeJS basiert auf der **Javascript Chrome V8 Engine** und erlaubt das Programmieren der **Serveranwendung in Javascript.**
- **Javascript** beherrschen die **viele Programmierer**, was eine Einarbeitung in Node.js einfacher macht.
- NodeJS nutzt keine Javascript Sprachdialekte, sondern **Plain ES6 Javascript**, wodurch neue Sprachfeatures genutzt werden können.

- NodeJS arbeitet **asynchron** (ereignisgesteuert), was Vorteile (z.B. Eventhandling) und auch Nachteile (z.B. verschachtelter Code, Lösungen sind gegeben) mit sich bringt.
- **Non-Blocking-I/O:** Sämtliche Funktionen, die nicht direkt im Node.js Code ausgeführt werden, **blockieren die Anwendungen nicht**, wie z.B. Laden von Dateien → dies wird an das Betriebssystem ausgelagert. (Beispiel: Während dem Laden einer Datei kann Node.js weitere Anfragen bearbeiten).
- Ein **Garage Collector** ist enthalten.

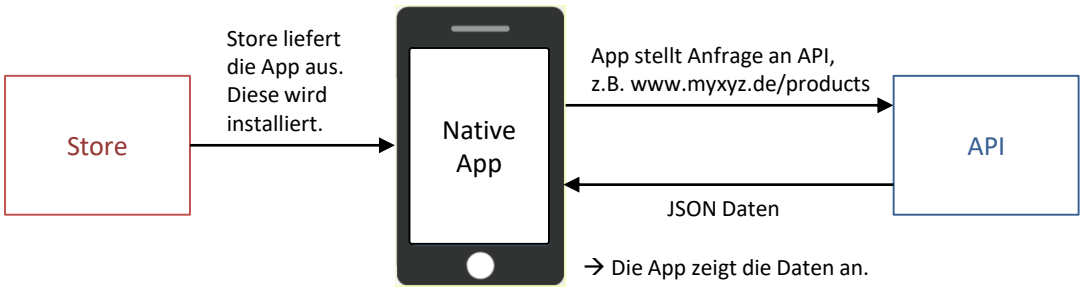
- **Node.js ist Single Threaded** → Javascript Code blockiert folglich die Anwendung
  → **Ressourcenschonendes Programmieren notwendig** (was aber mit ein paar Regeln kein großes Problem darstellt).

- Manche Server-Services (z.B. Amazon Web Service, openshift) bieten an, bei hoher Last mehrere Instanzen der Anwendungen auch auf verschiedenen Kontinenten zu starten. Damit sind auch größere Projekte denkbar.

Konzept der Webapp und nativen Apps

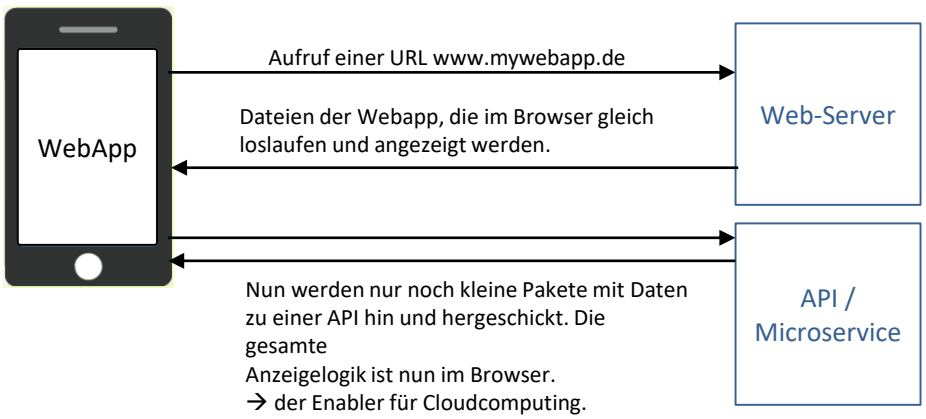
Native Apps

- **Native Apps** werden auf einem Smartphone **installiert**. Marktführer sind hier Google Android und Apple iOS.
- Diese Apps müssen mit **der jeweiligen Entwicklungsumgebung programmiert** werden (Android Studio mit Kotlin/Java, Xcode mit Swift). Die Apps werden über die Stores vertrieben und auf dem Smartphone/Tablet installiert.



WebApps

- Die Grundlegende Idee der Webapp ist **ähnlich zur nativen App**. Der Unterschied ist, dass **die App selbst sehr klein** ist und beim **Aufruf der URL ausgeliefert wird**, also nicht aus einem Store installiert wird.
- Da Webapps im Browser laufen, werden sie in **Javascript/Typescript** entwickelt.
- WebApps sind häufig als **Fat Client** konzipiert (viel Logik und „Software" im Browser, daher vergleichsweise wenig Datenmengen in den Übertragung mit dem Server, da nur kleine Pakete ausgeliefert werden).
- Daher lassen sich WebApps wie Programme auf dem PC bedienen. Auf die **Ladezeit der WebApp ist zu achten, da mehr Code im Browser geladen. Nach dem Laden** die Nutzung sehr viel fließender.
- **Komplexe Logik** ist hierdurch im Browser umsetzbar.
- Eine **Verbindung zum Server ist aber meist nur dann notwendig**, wenn **Lese/Speicheroperationen** auf eine Datenbank getätigt werden sollen.
- Vorteilhaft gegenüber Desktopanwendungen ist, dass **keine Installation von Software notwendig** ist, was teilweise viel Ressourcen im Unternehmen bindet.
- **Updates** sind **sofort beim Laden verfügbar**.
- Nachteilig ist, dass **der Zugriff auf Hardware erschwert möglich ist** und Browserunterschiede berücksichtigt werden müssen, was aber heutzutage mit geringem Aufwand lösbar ist.



HybridApps

- Bei HybridApps werden **WebApps in nativen Apps ausgeführt.**
- Die Webapp kann neugeladen werden beim Aufruf der nativen App.
- Alternativ können die Dateien der WebApp **im nativen Code eingebettet sein** (dann sind die Seiten nur updatefähig, wenn die App upgedated wird. Allerdings lädt die Seite fast ohne Zeitverzögerung auch ohne Datennetz).

Moderne **Webanwendungen heute und in Zukunft**

WEBENTWICKLUNG 2
one step ahead

Moderne Systeme

- Heute finden sich leider zahlreiche enorm schlecht gemachte Websoftware im Markt, die immer noch auf veraltete Technologien aufgebaut werden.
- Grund ist das Fehlen von modernen Webkompetenzen bei vielen Mitarbeitern im Webbereich und zunehmenden Bedarf an Entwicklern.
- Modernste Websysteme **nutzen eine Kombination aus schnellen SSR und intelligenten Komponenten oder Single Page Apps,** womit modernste Marketingorientierte Lösungen gebaut werden können bis hin zu leistungsfähiger optimal bedienbarer Software im Browser.
- *Anmerkung TW: Dies alles benötigt umfangreicheres fundiertes Wissen als das Zusammenklicken aus Tutorials und das Zusammenflicken von überteuerten Wordpress Seiten. Diese Vorgehensweise war für das Scheitern sehr vieler Projekte verantwortlich, was endlich aufhören muss. Und das geht nur über das Wissen und Können moderner Konzepte.*
- **Daher wurde für diese Vorlesung ein Toolset gewählt, das einen Einstieg in diese Welt bietet, der seitens Toolset nicht so sehr frustriert.**

Zukunft?
Keine Ahnung! Das ändert sich jeden Monat.

Javascript

- UI Logik basiert im Web auf Javascript, deswegen sind Programmierkenntnisse und der sichere Umgang mit Javascript unverzichtbar.
- Asynchrone Denkweise

Solid / Sverve

- Intelligente Komponenten für erweiterte UI Logik.
- Integrierbar in die Seiten von Astro.

Astro

- Als Beispiel für ein modernes integriertes System für SSR, das Features wie Markdown beherrscht.
- Einfaches Deploying
- Integration von fast allem, was derzeit angesagt ist.

Node

- Ein Serversystem, das komplexe Businesslogik abbilden kann.
- Kann Microservices umsetzen.
- Login und komplexe Serversysteme

Werkzeuge für leistungsfähige Portale.

Klassische, schöne, schnelle effiziente Webseite

Spekulation TW:
HIer wird ChatGPT und Co bald sehr viel automatisieren.

Spekulation TW:
Bei Portalen und Digitalisierung ist noch viel Faktor Mensch, Unternehmen und Prozess drin. Hier könnte es noch eine Weile dauern.