HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Teil 1: Überblick und grundlegende Konzepte

Prof. Dr.-Ing. Thorsten Weiss

Version 2.0

Anwendungsfelder

WEBENTWICKLUNG 2
one step ahead

Wozu wird Web-Software verwendet (MD)

Marketing	E-Commerce	Portale
(Firmenwebseiten, Landing Pages)	(Shops, Vertragsabwicklung, Warenwirtschaft)	(Services, SaaS, customized Function)

Wichtigster zu beachtender Punkt:

Technologien sind nicht für alle 4 Felder geeignet!

Was soll das Fach Web 2 erreichen?

- Einordnung der Technologien.
- Welche Technologien sind wofür geeignet?
- Wie kann im Tec-Umfeld argumentiert werden (siehe Softskills & agile Methoden)?
- Wie können Webanwendungen schnell bewertet werden?
- Daher sind die Haupthemen:
  Wie können Marketing Seiten gebaut werden?
  Wie können Portale gebaut werden?
  Wie kann echte reaktive UI gebaut werden?

Was soll das Fach Softskills & agile Methoden beitragen?

- **Umfangreichen Umfrage:** Welche Softskills sind relevant?
- **Ergebnis:** Manche sind sehr relevant, andere weniger.
- **Daher:** Inhalte werden rausgenommen und dafür folgende Themen in Softskills eingefügt:

  - SEO, Marketing, Influencer Marketing, Reichweitenaufbau und Social Media.
  - Webseiten-Optimierung
  - Advanced Git-Know How

Homepage Baukästen (kleiner Auszug)

Im Bereich der Content-Management-Systeme (CMS) gibt es neben WordPress, das mit einem Marktanteil von 61,8 % im Januar 2025 führend ist, weitere Systeme mit signifikanten Marktanteilen. Hier sind die zehn CMS mit den größten Marktanteilen: de.statista.com +2

CMS	Marktanteil
WordPress	61,8 %
Shopify	6,7 %
Wix	3,8 %
Squarespace	2,6 %
Joomla	1,6 %
Drupal	1,2 %
Adobe Systems	0,9 %
Blogger	0,7 %
Bitrix	0,6 %
OpenCart	0,5 %

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:

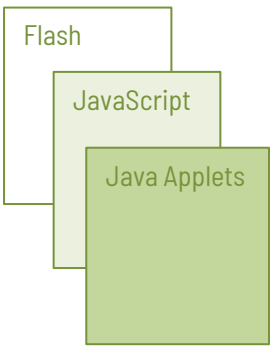
- Browserabhängigkeit.
- Hohe Unflexibilität
- Security
- Hohe Entwicklungszeiten
- Daten (Texte) mit Struktur und Design verschmolzen.

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 Serversystem für die Auslieferung und einfache Formulardaten und Auth.

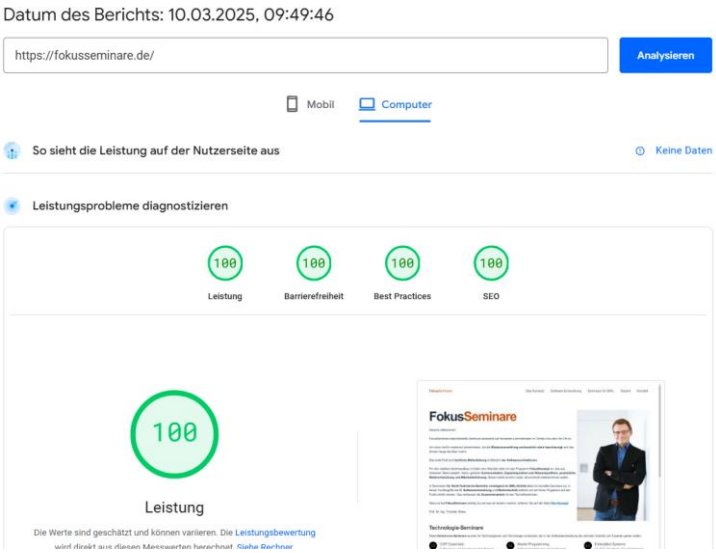
Größte Herausforderungen damals:

- 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.

Frontend-Technologien

- Verschiedene Webtechnologien eignen sich für verschiedene Einsatzzwecke. Dieses Kapitel soll hier Klarheit bringen, was wofür eingesetzt werden sollte.
- Um die Performance zu messen eignet sich zum Beispiel **Lighthouse Erweiterung** für Chrome-Browser.



Ebene 1: SSG – Static Side Generation

- SSG (Static Site Generation) ist eine Methode zur Erstellung statischer Webseiten, bei **der die Inhalte bereits zur Build-Zeit generiert. Diese „Prebuild" Seiten können sehr schnell ausgeliefert werden.**
- Dies führt zu schnelleren Ladezeiten und höherer Sicherheit, da keine serverseitige Verarbeitung bei jeder Anfrage erforderlich ist.
- Wenn Inhalte geändert werden sollen, muss ein neuer Build erfolgen.
- Viele Meta-Frameworks unterstützen SSG. Ein gutes Framework unterstützt mehrere Technologien und erlaubt einzelne Seiten per SSG auszuliefern.

Einsatz:

- **Schnelle Landing-Pages** sind essentiell für SEO und Online-Marketing, da die Suchmaschinen die Performance von Webseiten.
- Viele Frameworks integrieren Möglichkeiten für SEO (Search Engine Optimization) , SEM (Search Engine Marketing) und SEA (Search Engine Advertising).
- Für **Content**, der sich sehr selten ändert, ist SSG die Wahl, da es sehr effizient läuft.
- Für **Artikel** und **Blogs** ist es geeignet, wenn der Blog nach der Erstellung in eine SSG-Seite gebaut wird.

Einschränkungen:

- Es ist ungeeignet für Seiten, die veränderbare Daten anzeigen, zum Beispiel Daten aus einer Datenbank darstellen.
- SSG braucht einen Build-Prozessor. Der Build kann unter Umständen länger dauern, wenn keine performanten Build-Systeme genutzt werden.

Ebene 2: SSR – Server Side Rendering

- Über eine URL mit Routen und Params wird eine Seite vom Server angefordert. Dieser generiert (bei jeder Anfrage) HTML und schickt es ggf. mit CSS und ggf. mit JS an den Browser.
- Wenn hier aufwendige Prozesse wie komplexe Datenbankabfragen zu erledigen sind, kann der Server stark belastet werden. Hier ist Caching ein wichtiges Thema.
- **Höhere Serverlast:** Da jede Anfrage eine neue HTML-Generierung erfordert, kann SSR mehr Serverressourcen beanspruchen. Die Serverlast muss aber auf die maximale Anzahl von Nutzern ausgelegt werden. Viele Seiten haben Stoßzeiten (z.B. Shops, wo die Seite dann ggf. langsamer wird, wenn nicht genug Power zur Verfügung steht).
- Es wird gesagt, dass die Darstellung im Browser schneller sei, weil kein JS ausgeführt werden muss. Das stimmt aber nur für statische Darstellungen. Sobald reaktive Komponenten reinkommen ist SSR eher langsam.
- SEO/SEA/SE-Vorteile: Suchmaschinen können die Inhalte direkt indexieren, da sie bereits im HTML vorhanden sind.
- Dynamische Inhalte: Daten werden bei jeder Anfrage frisch vom Server geladen, was für sich häufig ändernde Inhalte ideal ist, aber auch sehr viel Power braucht (siehe Shop Seite). Oft müssen Duzende Dateien ständig neu geladen werden.

Ebene 3: SSR mit Islands

- Um reaktive Oberflächen zu erhalten, wird JS benötigt. Hierzu kann VanillaJS (also JS ohne Library) über leichtgewichtige Libraries wie HTMX bis hin zu aufwendigen Frontend-Frameworks genutzt werden wie React oder Angular.
- Um diese Möglichkeiten zu nutzen, bieten manche Meta-Frameworks die Möglichkeit, nach dem Laden einer Seite mit SSR oder SSG JS code nachträglich „reinzuladen" (Hydration), um die Seite schnell laden zu lassen und trotzdem UI-Logik zu ermöglichen.

Einsatz:

In Baukästen wie WordPress, Joomla und Typo-3 sind ab dieser Ebene viele Frickellösungen am Markt, da eine saubere Island Architektur nicht integriert ist.

Einsatz:

- **Schnelle reaktive Seiten – je nach Architektur sinnvoll für kleinere UI-Logik-Einheiten.**
- Prominenter Vertreter ist Astro, das Islands für alle gängigen Frontend-Libraries ermöglicht.

Einschränkungen:

- Für komplexe Webapplikationen wie Google-Docs oder Office 365 sind Islands nicht gemacht. Hierzu werden SPA genutzt.
- Wenn zu viele Libraries vermischt werden, dann ist der Pflegeaufwand enorm.

Ebene 4: SPA – Single-Page-Apps

- Bei Single Page Apps wird nur ein HTML-File geladen – daher der Name Single Page App. In dieser HTML wird ein JS-Paket geladen, das alle UI-Aufgaben übernimmt.
- Unterschiedliche Seiten werden über interne Routing Mechanismen übernommen.
- Da das initiale Paket oft größer ist, ist die initiale Ladezeit höher.
- Wichtige Frontendvertreter sind: React, Preact, Vue, Angular, Svelte, Solid und leichtgewichtige HTML-Erweiterungen wie HTMX.

Vorteile:

- SPA werden für echte Webapplikationen eingesetzt, sprich für Software, die im Browser läuft und sich so reaktiv anfühlt wie installierte Software.
- UI-Logiken können schnell und intuitiver entwickelt werden.

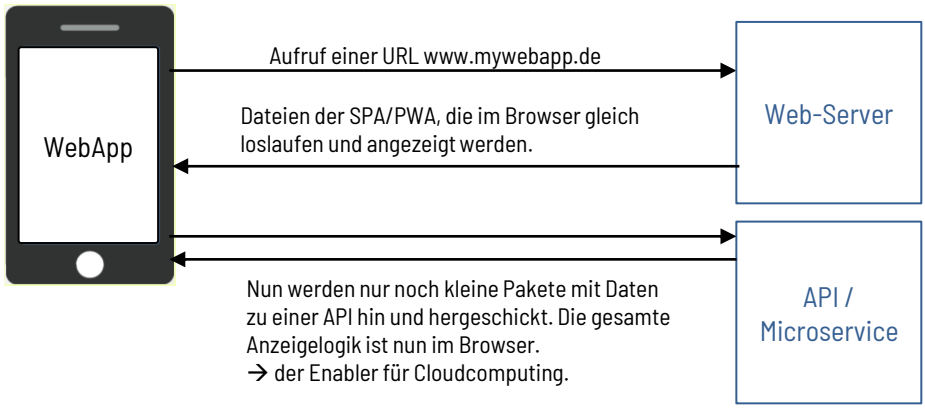
Einschränkungen:

- SPA sind ohne weiteres nicht gut für Bereich SEO/SEA/SEM gerüstet. Hier braucht es Workarounds oder leichtgewichtige Libraries.

SPA – Single-Page-Apps – weitere Eigenschaften

- 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.
- SPA 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).
- Ein Datenaustausch mit dem Server erfolgt nur bei Lese/Schreiboperationen auf eine Datenbank.
- Vorteilhaft gegenüber Desktopanwendungen ist, dass **keine Installation von Software notwendig** ist, was teilweise viel Ressourcen im Unternehmen bindet.
- **Updates** sind **sofort beim Laden verfügbar**.
- Bundling ermöglicht es, nicht immer die ganze Apps sondern einzelne Teile (chunks) bei Bedarf zu laden. Das verringert das erstmalige Laden.

- Darum sind SPA ist ein wichtiger bei der Digitalisierung und beim Cloud Computing.



Ebene 5: PWA – Progressive Web Apps

- PWA (Progressive Web Apps) gehen den nächsten Schritt zu mehr Autarkie. PWA können so gebaut werden, dass quasi alle notwendigen Software-Teile als Bundle gebaut werden und auf einmal auf das Smartphone geladen und dort auch gespeichert werden können. So kann auch komplexe Logik wie bei einer installierten App genutzt werden. Es können auch Synchronisierungen mit dem Server gemacht werden, sodass die App auch ohne Datenverbindung nutzbar ist (wie eine installierte App).
- Nur zum Datenaustausch wird dann mit dem Server interagiert.
- So fühlt sich die App sehr ähnlich zu einer installierten App an.
- Dies ist ein Weg, aufwendige Logik ohne App-Store auf ein Mobil-Device zu bringen.
- *Anmerkung TW: Das finden Apple und Google nicht ganz so richtig super, da sie mit den Stores sehr große Marktmacht haben. Wenn Leute nun PWAs verstärkt „am Store vorbei nutzen" könnte das in der Fantasie eines Informatikers eine Erklärung dafür erwachsen, warum ein paar Funktiönchen nicht so recht supportet werden, obwohl das technisch vielleicht recht sicher gehen würde.*

Vorteile:

- Sehr reaktiv, sehr flexibel, plattformunabhängig.

Einschränkungen:

- Die Nutzung von Hardware wie Kamera etc. ist eingeschränkt. Aufwendige Daten zu speichern ist ebenfalls vor allem aus Sicht der Security möglich.

Ebene 6: Native Apps

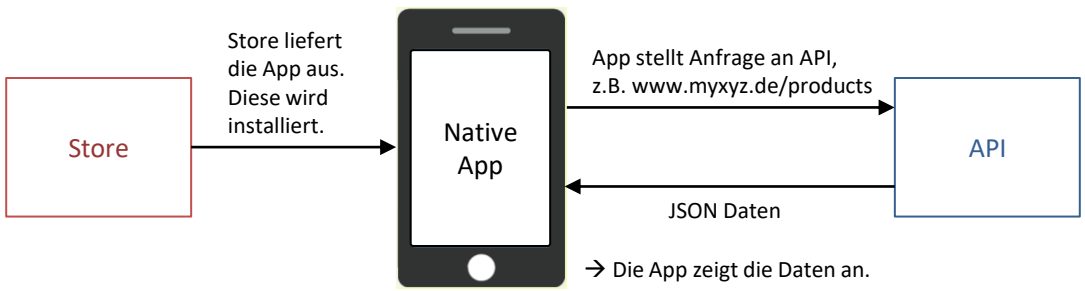
- **Native Apps** werden auf einem Smartphone **installiert**. Marktführer sind hier Google Android und Apple iOS.
- Diese Apps müssen mit **der jeweiligen Entwicklungsumgebung programmiert** werden (Android Studio mit Kotlin/Java, Xcode mit Swift). Die Apps werden über die Stores vertrieben und auf dem Smartphone/Tablet installiert.
- Heutzutage lassen sich Apps mit Cross-Plattform-Werkzeuge – alles voran Flutter gefolgt von React Native – mit einer höheren Developer-Experience entwickeln als die nativen Systeme.
- Trotzdem muss hier beim Projektstart genau evaluiert werden, für was man sich entscheidet. (Siehe VL: Mobile Anwendungen).

Vorteile:

- Installiert auf dem Smartphone / Tablett.
- Voller Hardwaresupport.
- Alle Möglichkeiten der Datenspeicherung.

Einschränkungen:

- Die Nutzung von Hardware wie Kamera etc. ist eingeschränkt. Aufwendige Daten zu speichern ist ebenfalls vor allem aus Sicht der Security möglich.

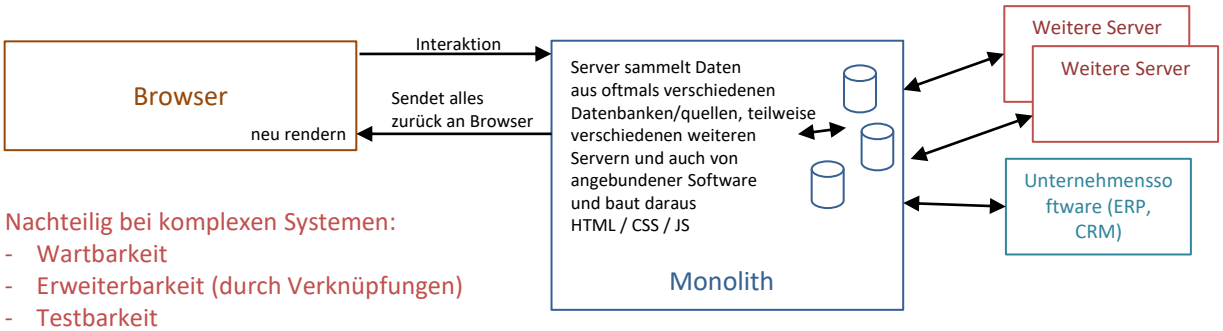


HybridApps

- Bei HybridApps werden **SPA/PWA in nativen Apps ausgeführt.**
- Die Webapp kann neugeladen werden beim Aufruf der nativen App.
- Alternativ können die Dateien der SPA/PWA **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).

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.

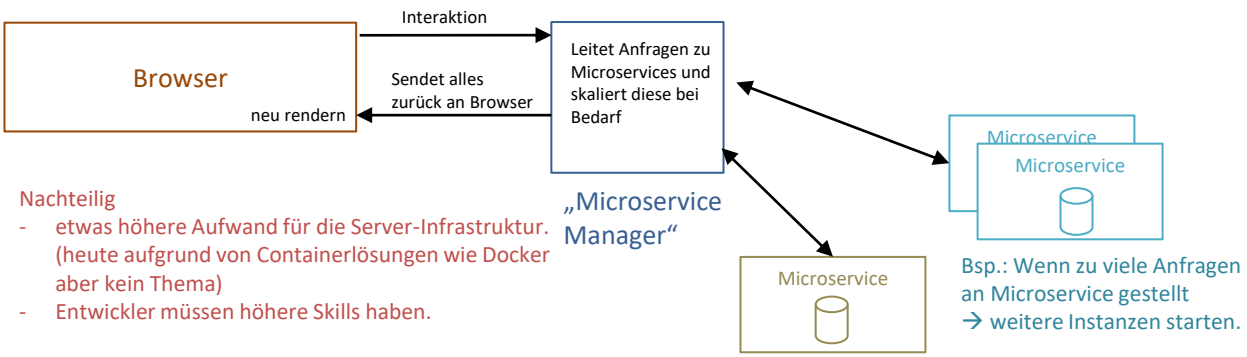


Nachteilig bei komplexen Systemen:
- Wartbarkeit
- Erweiterbarkeit (durch Verknüpfungen)
- Testbarkeit
- Auf ein Tool festgelegt (z.B. Java Server)
- Keine Skalierung

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.

Bsp.: Wenn zu viele Anfragen an Microservice gestellt → weitere Instanzen starten.

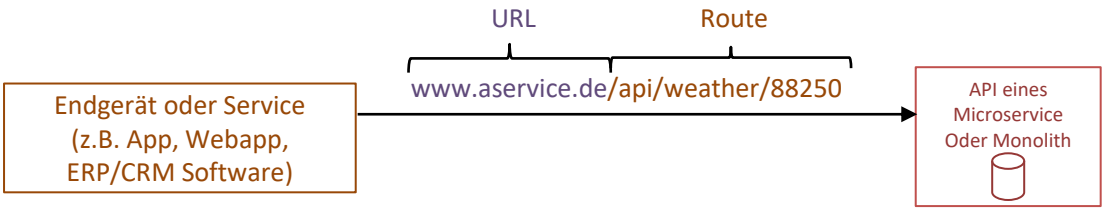
Vorteile:
- Wartbarkeit
- Erweiterbarkeit (durch Verknüpfungen)
- Testbarkeit
- Auf ein Tool festgelegt (z.B. Java Server)
- Skalierbarkeit

„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.

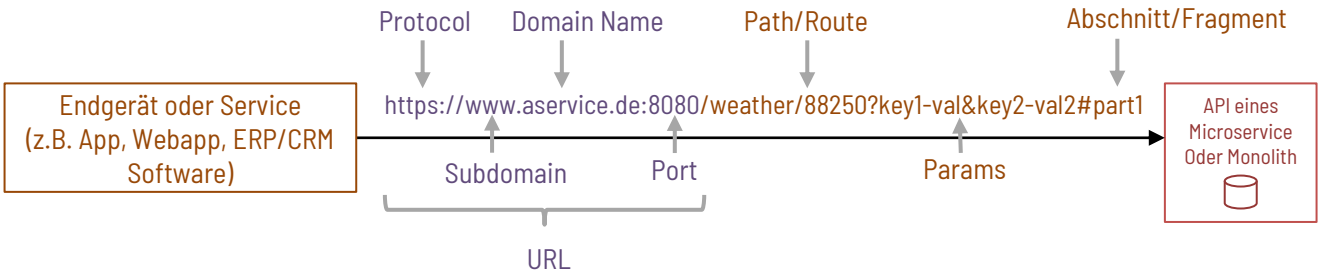
- Manche SaaS (Software as a Service) bestehen oder beinhalten nur eine API. SAP geht beispielsweise den Weg, ein SaaS zu werden und Frontends separat anzubieten oder entwickeln zu lassen. Die Firmenlogik läuft im Hintergrund.

„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.
- Das **Backend muss nicht eingerichtet** und **gehostet** werden.
- 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 die sehr interessante Open-Source Alternative Supabase.
- 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 IoT-Anwendung, 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 / Cloud-Software. Das Geschäftsmodell gehört hier dazu.**
- SaaS enthalten häufig Frontends oder Dashboards und stellen so Software zur Verfügung.
- 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 **nur als API und stellen Services zur Verfügung.**

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 **A**pache-**S**erver mit **P**HP, **m**ySQL (**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

Metaframeworks

- Meta-Frameworks stellen zahlreiche Serverfunktionen zur Verfügung,
- Serverless Functions.
- SSR, SPA, SSG
- Oftmals ist an SEO gedacht.
- Für alle großen Web-Frameworks gibt es ein oder mehrere Meta-Frameworks.
  Beispiele :
  Vue → Nuxt
  React → Next
  Solid → SolidStart
  Svelte → Svelte
  Angular hat schon viel an Board.
  Alle können in Astro eingebunden werden.

Serverless Functions

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

Prof. Dr. Thorsten Weiss

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

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.

Klassische, schöne, schnelle effiziente Webseite

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

Solid / Sverve

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

Node

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

Werkzeuge für leistungsfähige Portale.

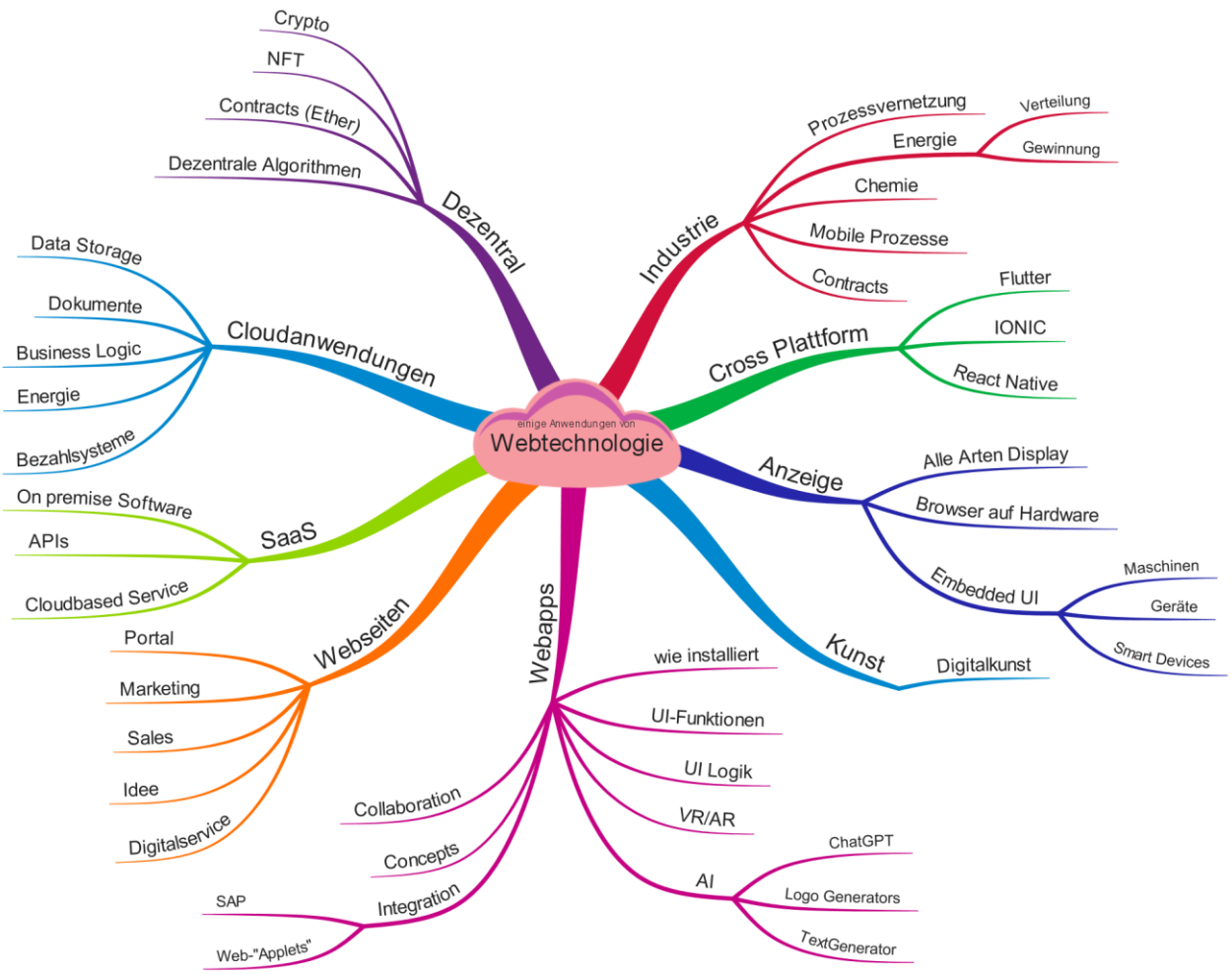
Spekulation TW:
Bei Portalen und Digitalisierung ist noch viel Faktor Mensch, Unternehmen und Prozess drin. Hier könnte es noch eine Weile dauern.

Prof. Dr. Thorsten Weiss

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.