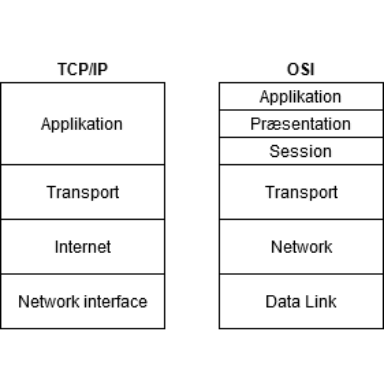
# Basale teknologier

Dette afsnit handler om det basale teknolgoier

## Internettet

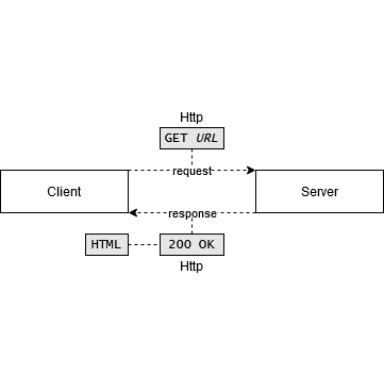
Internettet er et computernetværk bestående af computernetværk som er forbundet gennem routere. For at sende beskeder på internettet, bruger computerenhederne TCP/IP protokolstakken. Det er en gruppe af protokoller udarbejdet af det amerikanske Department of Defense, som tillader computerenheder at kommunikere. Protokolstakken kan inddeles i lag for protokollernes forskellige ansvarsområder. TCP/IP modellen beskriver de fire lag, som protokolstakken består af. Disse lag hedder *network interface*, *internet*, *transport* og *application*. Disse kan sammenlignes med lagene i OSI-modellen, som i sin abstrakte model deler netværkstrafik op i syv lag.



TCP/IP og OSI model

TCP/IP network interface laget svarer til OSI data link laget, som forbinder internettets netværkstrafik til det lokale netværk. Internet laget svarer til netværkslaget, som router netværkstrafikken fra computerenhed til computerenhed. TCP/IP transport laget svarer til OSI transport laget, som etablerer forbindelsen mellem computerenhederne. Endelig svarer TCP/IP applikations laget til applikations-, præsentations- og sessionslagene i OSI-modellen. På dette lag ekisterer adskillige services som e-mail og *world wide web*.

World wide web blev udviklet i 1989 af Tim Berners Lee. Han specificerede HTML formatet til at strukturere tekst, Unique Resource Identifier (URI) specifikationen til at addressere ressourcer og HTTP protokollen til at kommunikere mellem klienter og servere.



World Wide Web

Efterhånden som websider gik fra at være blot at være statiske og præsentere dokumenter, til at være dynamiske og leve af brugerindhold, begyndte man at italesætte dette som *web 2.0*. Begrebet blev opfundet af Tim O’Reilly[^web20].

## JavaScript

JavaScript er det programmeringssprog, som flest udviklere benytter. Det er også det programmeringssprog, som har flest aktive repositories på GitHub. JavaScript anvendes i dag til alt fra simple scripts på websider, til avancerede webapplikationer, som kan køre i browseren eller på næsten hvilken som helst anden platform, herunder webservere eller sågar mikrokontrollere.

### Engine og Runtime Environment

JavaScript tilhører kategorien af *interpreted* programmeringssprog - i modsætning til *compiled* programmeringssprog som C#. På et compiled programmeringssprog oversætter udviklermaskinen programmets kildekode til maskinkode med en compiler. I et interpreted programmeringssprog oversættes kildekoden ikke direkte til maskinkode. I stedet læser oversætter et andet program kildekoden linje for linje til maskinkode instruktioner.

JavaScript kildekode oversættes af et JavaScript *engine*. Et Javacript engine består af en *call stack* og en *memory heap*. Call stacken er en datastruktur, som holder styr på programmets *stack frames*, hvilket svarer til de enkeltvise instruktioner i kildekoden. Fordi JavaScript kun har en enkelt call stack, er JavaScript *single-threaded*. Det vil sige, at der ikke behandles flere instruktioner samtidigt. Memory heapen indeholder den allokerede hukommelse til programmets referenceværdier, hvilket i JavaScript vil sige objekter.

JavaScript består mere generelt også af det *runtime environment*, som platformen udvider det pågældende engines muligheder med. Forskellige platforme har forskellige engines og tilbyder forskellige runtime environments. Det fleste runtime environments har et *event loop* og en *message queue*, som giver JavaScript mulighed for *asynkron* single-threaded afvikling af kildekoden.

Asynkrone instrukser i kildekoden er f.eks. dem, som arbejder med netværkskald og derfor koster tid. For at de ikke skal blokere applikationens eneste tråd, bliver disse instrukser lagt i en kø datastruktur. Event loopet holder øje med, om der er flere frames at køre i call stacken. Hvis ikke, så henter event loopet de asynkrone instruktioner ud fra message queue og ind i call stack, så de bliver kørt. På denne måde tillades asynkron JavaScript kode på trods af, at JavaScript er single-threaded. Browserplatformens runtime environment har også nogle *web APIer*, som er af særlig interesse for webudviklere.

Browseren implementerer internt diverse funktioner med et lavere niveau programmeringssprog som C++. Web APIerne giver adgang til disse browserfunktioner, til f.eks. netværkskald, lyd eller 3D grafik. Document APIen er det mest vigtige for webudviklere, da denne giver adgang til Document Object Model (DOM) træet.

DOM træet er en intern datastruktur i browseren. Datastrukturen er et hiearkisk træ, som repræsenterer siden aktuelle HTML struktur. Hver eneste knude i DOM træet repræsenterer et enkelt HTML element med alle dets attributter. Hvert undertræ består altså af et HTML element og alle dets børn. DOM træet er til hver tid en afspejling af HTML strukturen på den aktuelle webside. Med Document APIen er det muligt at manipulere DOM træet og alle dets knuder, f.eks. ved at oprette nye HTML elementer eller ved at oprette event listeners på eksisterende HTML elementer.

### Versioner

Det oprindelige JavaScript blev skabt i 1995 af Brendan Eich, som var ansat hos Netscape, for at understøtte scripting i deres browser, Netscape Navigator 2.0. Det blev udgivet med en pressemeddelse[[1]](#footnote-27) hvori intentionen med sproget blev lagt frem. JavaScript blev præsenteret som et nemanvendeligt *object scripting* sprog til webdokumenter, som supplement til HTML og Java. Sproget blev også præsenteret som en implementation af en *åben standard* under en kommercielt venlig licens, for at tillade det brede udviklersamfund at bidrage til sprogets udvikling.

JavaScript betegnes som sagt et objekt scripting sprog, og man kalder også JavaScript *objekt baseret*. JavaScript understøtter det objekt-orienterede programmeringsparadigme, men sproget adskiller sig fra konventionelt objekt-orienterede sprog som Java og C# bl.a. ved ikke at have statiske typer eller sondring mellem instanser og klasser.

Den fornævnte åbne standard udmøntede sig i JavaScript Reference Implementation (JSRef), som var et open source JavaScript engine, der udviklede sig til . Men eftersom JSRef ikke blev udgivet i tide, havde Microsoft i mellemtiden analyseret JavaScript som det var implementeret i Netscape Navigator, og på den baggrund udviklet deres egen engine, som de kaldte Chakra. Fordi navnet JavaScript var varemærkeregistreret[[2]](#footnote-29), blev Microsofts udgave af det samme sprog kaldt JScript.

For at harmonisere de forskellige JavaScript engines, blev JavaScript indsendt til ECMA International, som håndterer webstandarder. De udgav i 1997 en åben specifikation af sproget i form af ECMAScript standarden[[3]](#footnote-31). Dermed kunne konkurrende JavaScript engines overholde en fælles standard, så alle JavaScript engines kunne mindske jævnbyrdes afvigelser i håndteringen af JavaScript kildekode. Der findes i dag mange forskellige JavaScript engines til forskellige platforme, som alle afvikler JavaScript kode efter denne samme standard. ECMAScript standarden hedder formelt ECMA-262, og implementeringerne af standarden i form af programmeringssprog og engines hedder formelt ECMAScript i stedet for JavaScript pga. varemærkebeskyttelsen.

I daglig tale og i denne hovedopgave er ECMAScript synonymt med JavaScript, selvom det faktisk stadig er et registreret varemærke, der i dag er eget af Oracle, som endda i sjældne tilfælde[[4]](#footnote-33) stadig beskytter deres ophavsret på navnet. Der var på et tidspunkt tale om, at Oracle skulle donere varemærket til ECMA International[[5]](#footnote-35) i erkendelse af, at navnet JavaScript de facto var blevet et offentligt domæne, men dette blev aldrig til noget.

“ECMAScript was always an unwanted trade name that sounds like a skin disease” — Brendan Eich

ECMAScript standarden dikterer altså udviklingen af JavaScript sproget. Det betyder, at ældre browsere ikke understøtter alle moderne JavaScript funktioner, fordi de bruger et JavaScript engine, som implementerer en ældre udgave af ECMA-262. Der har været flere store udgivelser af ECMA-262, hvoraf de første tre udkom hurtigt efter hinanden: ES1 i 1997, ES2 i 1998 og ES3 i 1999. ES4 var under udvikling længe, men blev af forskellige årsager droppet. Først ti år efter udgivelsen af ES3, kom efterfølgeren ES5 i 2009, hvilket blev en milepæl for JavaScript, og det er den seneste udgave af sproget, som er understøttet af alle mainstream browsere inkl. Internet Explorer. Seks år senere kom ES6 i 2015, og herefter begyndte ECMA at udgive en ny version hver juni, så disse udgaver navngives typisk efter deres udgivelsesår: den syvende udgave hedder ES2016, den ottende ES2017, osv.

### Værdier og Variable

JavaScript har overordnet to forskellige datatyper af værdier. Det er *primitiver*, som opbevares på stack, og *objekter*, som opbevares på heap.

Primitiver er simple værdier, som JavaScript har syv forskellige typer af: Number, Boolean, String, Null, Undefined, Symbol (introduceret i ES6) og BigInt (introduceret i ES7).

Objekter består af *properties*, som er navngivne primitiver eller objekter. Properties kan også indeholde *functions*, som i Javascript er specielle objekter af typen Function, og *arrays* af primitiver og/eller objekter, som i JavaScript også er specielle objekter, der har typen Array. Functions og arrays er altså som næsten alt andet i JavaScript objeker.

JavaScript har som forventeligt variable til at indeholde værdier. I JavaScript kan ikke bare værdien af variabel ændre sig, når programmet kører, men også variablens type. Man kan lave variable med nøgleordene *var*, *let* eller *const*. Det er muligt at lave flere variable kun med brug af et enkelt nøgleord. Variabler oprettet med *var* kan ligesom

JavaScript kører som sagt i et runtime environment, som gør nogle specifikke objekter tilgængelige for udvikleren. Alle browsere giver fx adgang til document objektet, som er udviklerens indgang til DOM træet. Den giver metoder til at manipulere det hierarki af objekter, som hver især repræsenterer de HTML tags, som udgør den aktuelle DOM.

### Objekt-Orienteret Programmering

JavaScript er et objektorienteret sprog, men det har (i modsætning til fx. C#) ikke klasser og instanser, bare objekter. For at kunne lave objekter af en bestemt type, bruger man, i stedet for klasser og instanser, en *constructor funktion*. Denne specielle metode navngives konventionelt med stort begyndelsesbogstav. På constructor funktionen er der en property prototype, som indeholder et objekt. Ved at sætte properties på prototype objektet, angiver man hvilke felter, som skal være på objekter skabt med constructor funktionen. Når man laver objekter ved at kalde constructor funktionen med nøgleordet new, vil man kunne tilgå de properties og metoder, man har angivet i prototypen.

Når constructor funktionen kaldes med new, får Prototype objektet som standard en constructor property, som peger tilbage på den constructor funktion, som prototypen tilhører. For eksempel peger Animal.prototype.constructor tilbage på Animal. og \_\_proto\_\_ property, som angiver hhv. typens constructor funktion, og constructor funktionen for den nedarvede type (i sidste ende Object).

Hvis man vil specialisere sin constructor function, fx. lave en Mammal som specialisering af Animal, laver man en ny constructor function, og sætter prototypen til en instans af overtypen.

Når man tilgår en property, fx. (???) vil JavaScripts engine søge efter en property med det navn: først blandt objektets properties, dernæst i dens prototypes properties, så prototypens **proto**’s properties, osv. indtil roden af nedarvingshiearkiet, som er constructor funktionen Object.

I ES6 blev en ny syntaks indført for at gøre det lettere at arbejde med prototypiske nedarvningskæder. Man kan nu bruge class nøgleordet til at definere sine *klasser*. Syntaksen for klasser i JavaScript minder meget om klasser i andre objektorienterede sprog, fx. C#, men resultaten af klasse definitionen minder mest om prototypisk nedarvning, så man kan næsten betragte sprogkonstruktionen som syntaktisk sukker for at definere en constructor med dens prototype felter og metoder. Det forklarer bl.a. hvorfor, at klasser i JavaScript endnu ikke officielt understøtter private feltvariable.

### Funktionel Programmering

Jeg havde da skrevet en masse fedt, hvor blev det af…

### Moderne Syntaks

JavaScript er navngivet efter Java, fordi det blev skabt med ideen om at være et supplement til Java. Derfor er syntaksen også inspireret af Java, og nogle vil sige mere generelt af C. Hovedformålet var dog, at det skulle være nemt for nybegyndere at tage sproget og syntaksen til sig. Efterhånden som JavaScript blev mere udbredt, opstod behovet for at udvikle sproget og syntaksen. Især ES6 introducerede nogle vigtige nye sprogfunktioner og syntaks.

#### Lambda Udtryk

#### Moduler

I JavaScript er det muligt at importere *moduler*. Et modul er en kodefil, som kan importeres af andre kodefiler. Det nuværende modulsystem i JavaScript blev udgivet med ES6.

Moduler tillader at eksportere *named exports*, så man f.eks. kun eksporterer én funktion fra en fil. Der er også mulighed for \*\*default exports\*, som kan importeres nemmere, og generelt bruges, hvis moduler har en primær funktionalitet, som skal fremhæves. Default exports behøver ikke navngives, da de navngives ved importeringen. Endelig Før i tiden var der ingen understøttelse af moduler, så problemet blev løst med kodebiblioteker som CommonJS og RequireJS (hvorfor bruges requirejs stadig i Node?)

ES6 moduler

# Udbredte Teknologier

## Værktøjer og Udviklingsmiljøer

For meget nutidig JavaScript udvikling gælder det, at man ikke bare er nødt til at kende til web platformen og dens teknologier, men der eksisterer også mange værktøjer til at løse flere problemer, end JavaScript umiddelbart er designet til.

### Node

Node er et runtime environment til at køre JavaScript på serveren i stedet for klienten. Node bruger samme JavaScript engine som Chrome browseren, men giver adgang til andre APIer i stedet for browserens web APier. Med Nodes APIer har man fx adgang til computerens filsystem. Man kan afvikle JavaScript kildekode på alle styresystemer med Node installeret, hvilket gør det velegnet til cross-platform udvikling af f.eks. webservere.

Node bruger modulsystemet CommonJS, fordi Node udkom før ES modulerne blev introduceret i ES6. De indbyggede APIer i Node er moduler, som man indlæser med CommonJS. For eksempel kan man få adgang til filsystemet med modulet fs, som indlæses med den indbyggede require funktion.

Node installeres sammen med *NPM*, som er et program til at styre et projekts afhængigheder. NPM samarbejder med et online register af JavaScript biblioteker. Ved brug af NPM kan man nemt installere biblioteker fra registeret til sit projekt og styre afhængihederne med semantisk versionsstyring. Node installeres også sammen med NPX, som kan køre biblioteker, der er installeret som afhængighed, eller hente biblioteket fra registeret for at køre det, og herefter fjerne det igen.

### Webpack

Webpack er det, man kalder en *module bundler*. Det vil sige, at Webpacks primære formål er på buildtime at samle koden til én samlet JavaScript *bundle*, som indeholder alle applikationens moduler og deres indbyrdes afhængigheder. I konfigurationsfilen for Webpack definerer man et *entry point*, som Webpack skal starte i. Webpack starter i den fil, man har defineret som entry point, og gennemløber derfra kildekodens import/export udtryk, som den undervejs bruger til at bygge en *dependency graph*, som repræsenterer alle webapplikationens ES6 moduler og deres indbyrdes afhængigheder.

Med udgangspunkt i denne dependency graph, samler Webpack alle ES6 modulerne til en samlet JavaScript kildekode, som er klar til at blive downloadet og præsenteret af browseren. Dette giver i sig selv en gevinst, for brugeren kan få hele webapplikationen i ét request. Det kan gøre indlæsningen af alle modulerne betydeligt hurtigere på en traditionel HTTP/1.1 forbindelse med høj forsinkelse, da HTTP/1.1 håndterer hvert request synkront og dermed blokerer renderingen indtil hver static asset er downloaded og eksekveret.

Webpack kører også forskellige optimeringer på kildekoden, når bundler, for eksempel for at mindske filstørrelsen af den endelige bundle. Alt dette gør webpack ud af boksen med sin standard konfiguration, men langt de fleste bruger deres egen webpack konfiguration skræddersyet til deres projekt. Med webpacks konfigurationsfil er det muligt at ændre alle

For at komme i gang med at bruge webpack, er Node en forudsætning. Man kan bruge NPM til lave et nyt tomt projekt og installere Webpack:

Hvis overstående konfigurationsfil er til stede i rodmappen, vil Webpack automatisk samle den op, når man kører kommandoen webpack fra projektets rodmappe som arbejdsmappe. Hvis der findes en main.js, vil Webpack køre og resultere i en bundle i ./dist/bundle.js.

For at ændre output filnavnet til fx. output.js, tilføjer man en output property til config objektet, med en filename property. For at placere output placeringen til fx. ./build/, tilføjer man yderligere en path property til output objektet.

Webpack understøtter plugins.

Webpack giver mulighed for, at man kan indlæse trejdeparts *loaders* til at håndtere andre filtyper end JavaScript. Man konfigurerer webpack til at håndtere forskellige filtyper med forskellige loaders. Hver loader er en preprocessor, som Webpack anvender, når den støder på en angiven filtype. Loaders kan bl.a. sørge for, at Webpack også kan bundle stylesheets og andre assets. En hyppig anvendelse af loaders er også at få Webpack til at samarbejde med andre værktøjer som Babel og TypeScript.

### Babel

Babel Overstående HTML-lignende syntaks er den syntaksudvidelse, som kaldes JSX. Det transpileres (af Webpack) med en loader (Babel) til standard JavaScript. Webpack er en modul bundler, som samler projektets ES 6 moduler (ved import/export statements) til én samlet kildekode, og med Babel oversætter speciel syntaks som f.eks. JSX til standard JavaScript. Overstående kode bliver oversat af Babel til noget lignende:

### TypeScript

TypeScript er en udvidelse af JavaScript, som udvider JavaScript med type annotationer. TypeScript introducerer syntaks, man kender fra typestærke sprog som f.eks. interfaces, enums og generics. TypeScript introducerer også syntaks, man ikke traditionelt kender fra typestærke objekt-orienterede sprog, men til gengæld fra typestærke funktionelle sprog, som f.eks. *mapped types* og *algebraiske data typer*.

Mapped types er en syntaks, der kan transformere en type definition til en anderledes tilstand. F.eks. mapper Partial<T> typen alle properties på den generiske type til at være optional.

Algebraiske data typer er en syntaks, som tillader at kombinere typer for at skabe nye typer.

Fordelene ved TypeScript er mange, men vigtigst er naturligvis den typesikkerhed, som man kan opnå med TypeScript på compile-time. Typesikkerhed er uvurderligt, når man skal refaktorere gammel kode, da det giver mulighed for at ændre koden med en vis grad af tillid til, at der hvert fald ikke er introduceret nye fejl pga. inkompatible typer.

En anden fordel TypeScript har er, at det typestærkheden forbedrer dokumentation for koden. Hvis man importerer JavaScript moduler som har en korrekt type definition, kan man med sin IDE f.eks. udforske modulets eksporterede objekter, metoder, interfaces, etc. uden at man behøver nærlæse eller afvikle kildekoden. Med DefinitelyTyped registret, er det muligt at finde korrekte type definitioner til de fleste JavaScript kodebiblioteker, som endnu ikke har en officiel type definition (.d.ts fil).

En af fordelene ved TypeScript er også, at det ligesom Babel er en transpiler, der oversætter syntaks, og man ser ofte nye syntaks funktioner i de nyeste versioner af TypeScript, før de kommer til ECMAScript.

TypeScripts type system kaldes *structural typing*, hvilket adskiller sig fra f.eks. C#’s nominal typing ved at være mindre streng, når TypeScript skal bestemme hvorvidt et objekt overholder en type definition. Det bestemmes udelukkende ved, om et objekt har properties, som er tilsvarende alle de properties, som type definitionen indeholder, om den har yderligere properties, en anden type annotation men de samme properties, eller måske slet ingen type annotation. Det er kun de properties, som udgør måltypens interface, der bliver taget i betragtning.

## Biblioteker og Frameworks

Som udviklere er det vores opgave at vælge de rigtige biblioteker og frameworks til ethvert givet projekt. Dette kan blive kompliceret, fordi det ikke kun er projektets umiddelbare behov, der betyder noget. Man er del af et team og et større socialt system i en virksomhed, som har behov for at kunne vedligeholde systemet uden store omkostninger ved omskoling af sin medarbejderstab.

Mulighederne for at vælge JavaScript værktøjer er mange. Det er enormt mange JavaScript biblioteker tilgængelige. Man kan dele de tilgængelige biblioteker op i grupper af *generelle* biblioteker, *specialiserede* biblioteker og *frontend framework* biblioteker.

Der er generelle biblioteker, som giver mange hjælpemetoder til at gøre diverse opgavere lettere eller mindre repetetive. Et eksempel på det kunne være JQuery (~90kb), som i lang tid var et populært bibliotek til at gøre udvikleren i stand til at udtrykke mere med færre linjer JavaScript.

Der er specialiserede biblioteker, som indeholder relativt få linjer kode og løser et meget specifikt problem. Et eksempel kunne være Redux (~2kb), som kun har ansvar for at håndtere applikationstilstand.

Endelig er der biblioteker, der fungerer som frontend frameworks. Selvom det er muligt at opbygge en simpel frontend kun med HTML, CSS og JavaScript, giver forskellige frontend frameworks mulighed for bl.a. at strukturere kildekoden, så den ikke bliver for kompleks og uvedligeholdbar. Der er også behov for en struktureret måde at håndtere komponenter, styling og applikationstilstand. Frontend frameworks opstiller på forskellig vis et stillads til at løse disse problemer på en mere eller mindre fleksibel måde.

Frontend frameworks vil for denne opgave betragtes som værende biblioteker, der varierer i størrelse, og evt. bruges sammen med andre biblioteker, med det formål at bygge og vedligeholde *single page applications*. Et eksempel kunne være Angular (~500kb) , som er en “alt-i-en” løsning til at bygge single page applications, der kan håndtere applikationstilstand med den indbyggede data-binding i biblioteket. Et andet eksempel kunne være Vue(~80kb), som ofte bruges sammen med Vuex(~10kb) biblioteket til at håndtere applikationstilstand.

Single page applications er en tilgang til webapplikationer, som samler hele hjemmesidens indhold på en webside, i stedet for at sprede det ud på flere hyperlinkede websider. Når brugeren af webapplikationen navigerer rundt på siden, bliver det efterspurgte indhold hentet dynamisk ned fra webserveren og vist på websiden uden genindlæsning. Tilgangen er i kraftig vækst[^spa-growth] og bliver brugt af store virksomheder som Facebook, Twitter og Google. En meget udbredt arkitektur for single page applications, er en JavaScript SPA, der kommunikerer vha. HTTP og *JSON* med en *RESTful* webservice.

JSON er en data formatterings standard til at beskrive strukturen af JavaScript objekter. Selvom JSON er navngivet efter JavaScript, er det et selvstændigt data format, som kan bruges af alle programmeringssprog. Syntaksen for JSON tilsvarer JavaScript object og array literal syntaksen. Man kan repræsentere både primitiver, objekter og arrays med JSON.

RESTful webservices er karakteriseret ved, at de giver webadgang til ressourcer, som fx kan ligge i en database. Hver resource kan identificeres fra dens URL-adresse. Man manipulerer ressourcerne med HTTP beskedens operationsverbum: GET, PUT, POST eller DELETE som bruges til henholdvis at læse, opdatere, oprette eller slette ressourcer.

### Popularitet

Eftersom det, for de fleste webudviklere, ikke er overkommeligt at lære alle tilgængelige populære frontend frameworks, bør man gøre nogle overvejelser, før man vælger hvilket frontend framework at lære. For denne opgave vil jeg se på de mest populære frontend frameworks blandt udviklere og arbejdsgivere.

Når man dykker ned i de seneste års udvikling indenfor SPA frameworks, opstår der et mønster. Blandt udviklere er de mest populære frontend frameworks React, Vue og Angular.[[6]](#footnote-51) Det er umuligt at forudse, hvilke af disse frontend frameworks, der overlever længst, eller hvornår de bliver erstattet af noget nyt. Men der tegner sig et billede af, at flere udviklere benytter og udtrykker tilfredshed med Vue og React end Angular. React ligger i 2019 placeret på andenpladsen af mest anvendte biblioteker (med jQuery fortsat på førstepladsen). I kategorien “mest elskede” frontend frameworks, er React den mest populære, med Vue lige i hælene, mens Angular halter bagefter på en niendeplads. Vue har flere stjerner på dets GitHub repository end React og Angular.

Hvis man ser på, hvor meget de respektive frontend frameworks downloades af udviklere, ligger React væsentligt højere end de andre med op imod seks millioner downloads om ugen, sammenlignet med to millioner for Angular og en million for Vue.

### React

React er et JavaScript kodebibliotek til at bygge brugergrænseflader baseret på komposition af komponenter. Alting i React er en komponent, og selv hele applikationen er en komponent, som er sammensat af de øvrige komponenter. Komponent strukturen gør det muligt at genbruge og indkapsle alt lige fra en simpel checkbox til en hel applikation. Det gør det også nemt at ændre komponenterne og f.eks. tilføje yderligere state, uden at det skaber en dominoeffekt af side effekter i resten af applikationen. Komponent modellen passer ikke så godt med objekt modellen, og principper vi kender som søjler i objekt orienteret programmering, er ikke anvendelige eller har bedre løsninger i React. Facebook promoverer stærkt ”Composition over Inheritance”, og udtaler at de aldrig har brugt nedarvning i deres tusindevis af komponenter. [komponenter-billede]

React koncentrerer sig kun om brugergrænsefladen, og det er både en fordel og en ulempe. Langt de fleste React projekter vil have behov for at hente yderligere tredjeparts pakker ned for at håndtere problemer, som f.eks. Angular har indbyggede løsninger på. React omtales ofte heller ikke som et framework men et kodebibliotek, og man sammenligner det ofte med View-laget i MVC. Det gode ved Reacts minimalistiske tilgang er, at der er stor fleksibilitet i hvordan man løser et givent problem, og der er mange højt specialiserede værktøjer til at håndtere nicher som f.eks. routing eller global state.

React har hvad man kalder en deklarativ API. I stedet for at beskrive hvert trin og i hvilken rækkefølge, man opdaterer de enkelte elementer på brugergrænsefladen (imperativt), så beskrives hvordan man ønsker (deklarerer), at brugegrænsefladen skal se ud afhængigt af dens tilstand.

#### Komponenter og Props

En React komponent er ikke andet end en funktion, som returnerer noget, der i bund og grund bare er HTML. Strukturen og indholdet kan evt. afhænge af nogle funktionsparametre – eller hvad man i React termer ville kalde props. Disse props følger et princip om unidirectional data flow, eller en-vejs databinding. Props kan kun sendes fra forældre-komponent til barn-komponent.

Det hænger sammen med en anden egenskab ved props, som er, at de skal være immutable. En komponent kan læse sine props, men ikke overskrive dem. Eftersom en komponent modtager sine props som input, betyder det at man undgår side effekter i sine komponenter. Med andre ord, vil de samme props som input altid generere det samme HTML som output. Man siger, at komponenten er en pure function med hensyn til sine props. Reacts eneste regel er, at alle komponenter skal være pure functions mht. sine props. For at en funktion kan være en pure function, må den for det a) ikke have sideeffekter b) ikke kalde non-pure functions (som Date.now) og c) behandle sine argumenter som immutable (readonly). Der findes JavaScript kodebiblioteker som ImmutableJS til at håndhæve dette, eller code linting værktøjer som ESLint.

Props kan indeholde callbacks, så en forældre-komponent kan sende en funktion ned til barn-komponenten, som barn-komponenten kan kalde med en eller flere parametre for at sende data til forældre-komponenten. Det er den eneste umiddelbare måde komponenter kan sende data på, til andre end sine børn. Selvom det er med til at gøre programmet nemt at debugge og forstå, virker det umiddelbart som en begrænsning. For eksempel når når to komponenter A og B langt fra hinanden i træhiearkiet skal bruge data fra hinandens props. Det kræver at en fælles forældre-komponent til komponent A og komponent B sender dataen ned i sine props gennem flere mellemled. Dette pattern kaldes *lifting-state-up*.

Når man har en stor applikation med mange komponenter, som deler mange props, kan det blive uoverskueligt at vedligeholde med at løfte tilstanden, da det overholder ikke DRY princippet, eftersom den manuelle proces med at sende dataen igennem komponenten via props skal gentages for hvert enkelt mellemled mellem A og B. Dette anti-pattern kaldes *prop-drilling*.

#### Redux

En måde at omgå denne “prop-drilling” er ved at benytte Reacts indbyggede Context API. Den tillader komponenter at dele props indenfor en kontekst uden at mappe props mellem komponenter.

En anden mere avanceret måde, er at gøre som Facebook og bruge software design mønstret Flux. Flux applikationer har tre hovedbestanddele: dispatcher, stores og views. Ideen med Flux er, at den delte state ligger i en single source of truth dvs. i stores. State i stores kan kun opdateres ved at trigge en action. Actions fortæller sine stateændringer (payload) til dispatcher. Stores lytter på ændringer fra dispatcher, og opdaterer deres egen state i overeensstemmelse hermed.

Redux har også mulighed for middleware, som kan manipulere actions på vej ind og ud af reduceren. Dette giver mulighed for f. eks. asynkrone actions til API kald med redux-thunk middlewaren. Redux API’en giver også mulighed for at kombinere reducers med en higher order reducer funktion kaldet combineReducers. Det er altså en funktion, der tager flere reducers som parameter, og returnerer en ny kombineret reducer. Typisk har man en reducer slice for hver feature / domæne i projektet, som kombineres til en enkelt root reducer med combineReducers. Root reduceren bruges af Redux til at lave store objektet.

For at forbinde sine React komponenter til sin Redux store anbefales det at bruge Redux API’ens connect funktion. Connect tager to funktioner som input og returnerer en funktion, som er en higher order component, der tager en React komponent som input og returnerer en ”forbedret” udgave af komponenten med adgang til store og action creators via props.

#### Client-Side og Server-Side Rendering

Alt React kode som har direkte med browserens DOM at gøre, er splittet op i sit eget ReactDOM bibliotek. Man kan også udvikle med React mod en anden målplatform end browseren, ved at erstatte ReactDOM med et kodebibliotek til f.eks. cross-platform mobilapps (React Native) og konfiguration.

ReactDOM fungerer ved, at der fra webapplikationens index side kører en JavaScript fil i browseren, som holder browserens DOM synkroniseret med applikationens komposition af React komponenter og state. Alt logikken til at generere HTML dynamisk er således flyttet over i klienten/browseren. Denne client-side-rendering gør det muligt at have en interaktiv hjemmeside uden brug af en webserver.

Selvom man har en SPA uden en webserver, kan man stadig have pseudorouting med JavaScripts History API eller React udgaven, React Router. Men det er altid index siden, der indlæser hele webapplikationen, som præsenterer indholdet ud fra URL routen. Det medfører en bedre brugeroplevelse, da klienten aldrig indlæser en ny side efter index siden, og dermed slipper for nogensinde at genindlæse siden.

React opdaterer kun de DOM elementer i browserens hukommelse, som det er nødvendigt for at matche det tilsvarende React elements state. Denne optimisering fremkommer af Reacts VDOM. Det er en virtuel kopi af DOM’en, som eksisterer med den nyeste opdaterede state i hukommelsen. Før React opdaterer DOM, hvilket er en meget dyr operation, sammenligner meget hurtige algoritmer VDOM med DOM og finder de steder, hvor der er forskel i browserens DOM state fra Reacts interne VDOM state, så kun de absolut nødvendige DOM elementer behøver at blive opdateret

React understøtter også server-side-rendering, hvor serveren foretager den første indlæsning af webapplikationen og sender et snapshot af det fuldt formede HTML dokument til klienten. På denne måde understøtter man bedre søgemaskine crawlerne og opnår dermed en bedre rank i søgemaskinen. Man mindsker også tiden før first meaningful paint (når man kan se sidens primære indhold).

# Innovative teknologier

Everett Rogers har beskrevet udbredelsen af innovative teknologier i et socialt samfund. Hans *technology adoption lifecycle* model beskriver fem faser i anvendelse af ny teknologi, med udgangspunkt i de, som tager teknologien til sig.

Den første gruppe til at tage en ny teknologi i brug kaldes *innovators* (innovatør). Når teknologien ikke længere er ukendt, kaldes den næste gruppe *early adopters*. Når en teknologi begynder at blive meget udbredt, kaldes de som tager teknologien i brug *early majority*. Når en teknologi er nær toppen af sin popularitet, kaldes denne gruppe *late majority*. Den sidste gruppe er *laggards*, der anvender en teknologi, som i de flestes øjne er forældet.

Det er mest risikofyldt at være innovatør, fordi der på dette stadie kræver en dybdegående forståelse at vurdere, om den nye teknologi har potentiale. Der er også størst potentiel gevinst ved at være innovatør, da man kan bruge den innovative teknologi til at udkonkurrere sine konkurrenter. De fleste ledere af store virksomheder vurderer, at deres virksomhed ville blive udkonkurreret inden for syv år, hvis de ikke formår at følge med innovationen.[[7]](#footnote-58).

For at følge med innovationen, laver virksomhederne cost/benefit analyser af innovationen, hvor de altså ser på omkostninger kontra fordele ved at tage innovationen i brug. Ron Adner har beskrevet situationen, når en ny innovation skal tages i brug, som en konflikt mellem *innovatøren* og *forbrugeren*. Her forstås innovatøren som budbringeren af den nye innovation, som skal overbevise forbrugeren om at tage innovationen til sig. Man kan forestille sig produktproducenten som innovatør, der vil overbevise kunden i butikken som forbruger. Man kan også forestille sig udvikleren på et team som innovatøren, der vil overbevise resten af teamet som forbrugeren om at anvende et nyt bibliotek eller framework. Konflikten består af en forskel i innovatørens og forbrugerens opfattelse af omkostninger og fordele ved at tage innovationen til sig.

Innovatøren tænker på omkostninger med hensyn til den umiddelbare omkostning, der er ved at tage teknologien i brug. Det kan f.eks. være omkostninger til køb af licenser for at bruge innovationen. Forbrugeren tænker på omkostninger i forhold til den samlede omkostning af innovationen og alle de nødvendige ændringer, som er nødvendige for at bruge innovationen, som f.eks. omskoling af medarbejderstab og andre indirekte omkostninger.

Innovatøren tænker på fordelene med hensyn til den absolutte fordel, som forbrugeren modtager. Det kan være fordelene ved et nyt bibliotek eller framework, som gør arbejdet lettere for udvikleren. Forbrugeren tænker over benefit i forhold til den relative fordel, som innovationen giver sammenlignet med de tilgængelige alternativer. Det kan være i forhold til de fordele, som teamets eksisterende udvalg af biblioteker og frameworks tilbyder.

Innovatøren har altså en tendens til at sammenligne den absolutte fordel med den umiddelbare omkostning, mens forbrugeren sammenligner den relative fordel med den samlede omkostning. Den optimale situation ville være, at der ikke var nogen forskel mellem den innovatørens og forbrugerens perspektiv på fordele og omkostninger.

For at tilstræbe den situation, i forhold til udvikleren og resten af teamets perspektiv på omkostninger og fordele ved at tage et nyt bibliotek eller framework i brug, har mange virksomheder set lyset microservice arkitekturen til i at splitte deres monolitiske systemer op i microservices, som består af løst koblede moduler. Dermed behøver alle udviklere i virksomheden ikke at arbejde på den samme, komplekse kildekode. Hvert team er ansvarlige for en service, og der er en udvikler ansvarlig for hvert modul. Det giver udviklerne mulighed for at bruge de biblioteker og frameworks, som bedst egner sig til løse et givent problem. Det udligner forskellem mellem udviklerens absolutte fordel og teamets relative fordel, da resten af teamet ikke påvirkes af udviklerens beslutning. Det gør også forskellen mellem udviklerens umiddelbare omkostning og teamets samlede omkostning mindre, da hvert modul i en microservice arkitektur er afkoblet og nemt kan skiftes ud.

## Micro Frontends

Microservice arkitekturen er blevet meget udbredt, og størstedelen af store virksomheder er enten ved at tage microservices til sig, eller bruger allerede udelukkende microservices[[8]](#footnote-61). Brugergrænsefladen bliver for mange alligevel stadig udviklet som et monolitisk system. Microfrontend arkitekturen går ud på at overføre ideen om microservices til brugergrænsefladen.

Microfrontends kommer med mange af de samme fordele som microservices. Den centrale fordel er muligheden for en fastlægge en individuel kode-, test- og driftproces for hver microfrontend.

Dermed kan mindre teams være ansvarlige for egne microfrontends, og hurtigere komme med inkrementelle opdateringer uden at vente på hinanden. Det understøtter en agil procesmodel, hvor hvert team autonomt kan komme med en ny version af deres microfrontend efter et sprint.

Man kan opdele microfrontends i “vertikale” hensyn til forretningsdomænet, i stedet for “horisontale” hensyn til tekniske ansvarsområder. Dermed får hvert team en bedre helhedsforståelse af det domæne, som de med deres microfrontend er ansvarlige for.

Med microfrontends er kildekoden for hver microfrontend afkoblet fra hinanden. Fordi undersystemet er mindre end hele systemet, vil kildekoden for hver microfrontend også være mindre og nemmere at vedligeholde. Hvert team har også bedre fleksibilitet i deres valg af biblioteker og frameworks, som passer til lige præcis deres microfrontend.

## Module Federation

I beta version 5 af Webpack er der en ny funktion, som spiller en vigtig rolle i at understøtte en microfrontend arkitekturen. Det er indtil videre kun i beta version, og bliver officielt udgivet sammen med Webpack 5. Module federation tillader at bundle hver eneste individuelle microfrontend med alle sine afhængigheder. Dermed kan man have en individuel proces/pipeline for at udvikle, teste og udgive hver enkelt microfrontend uafhængigt af hinanden, men alligevel indlæse dem dynamisk med JavaScript i runtime og bruge dem som en samlet helhed.

Men module federation giver også mulighed for, at man angiver *shared dependencies*. Hvis man f.eks. har angivet react som shared for to forskellige microfrontends, og indlæser begge microfrontends, vil kun den ene af de to tilsvarende afhængigheder blive indlæst i runtime, men til gengæld vil dén ene afhængighed blive delt af begge microfrontends. Dermed opnår man *decoupling* af de enkelte microfrontends på en ressourceffektiv måde.

Module federation bruger betegnelsen *hosts* for applikationer, der indlæser en *remote* applikation. En applikation kan fungere som både host og remote på samme tid.

Man angiver hosts og remotes i sin webpack.config.js konfigurationsfil. Module federation er inkluderet i Webpack som et plugin, der først skal indlæses med RequireJS:

ModuleFederationPlugin initialisers i config objektets plugins array med et konfigurationsobjekt som parameter. Her angiver man den nuværende applikations navn, hvis den skal indlæses som remote. Her angiver man også hvilke remotes, som Webpack skal forvente at indlæse i denne host, og hvilke interne moduler, som skal udstilles som remote.

Hermed vil Webpack håndtere det korrekt, i stedet for at melde fejl, hvis den støder på et import udtryk, som skal findes på en anden webadresse i stedet for det lokale filsystem.

Hvis man ønsker at eksponere dele af applikationen som remote, er det for øjeblikket nødvendigt at angive applikationens offentlige webadresse, som den udgives på, i output property på config objektet til Webpack. Der arbejdes på en løsning, hvor dette kan angives fra konsummenten (hosten) eller udledes automatisk fra importeringen af remote.

Når man herefter kører Webpack, vil den som buildprodukt også lave en lille fil ved navn remoteEntry.js. Denne fil er host-applikationens indgangspunkt til remote-applikationen. Den indeholder ikke selve de udstillede moduler, men eksisterer for at indlæse. Denne fil skal refereres til i host’en, fx ved at indsætte et <script> tag i <head> tagget af websidens index.html:

Hermed er det muligt at importere alle de moduler, som remote-applikationen udstiller, så længe applikationen er i drift på den angivne webadresse. Man bruger en dynamisk import, hvor det navn, som man angiver for sin remote-applikation i host-applikationens ModuleFederationPlugin, fungerer som et namespace, som om remote-applikationen ligger på det lokale filsystem.

## Web components

Man kan betragte indbyggede DOM elementer, fx. input og form, som genbrugelige komponenter, der indkapsler et template (HTML), en style (CSS) og en intern logik (JavaScript). På samme måde kan man med web components lave genbrugelige komponenter af ren HTML, CSS og JavaScript, som kan anvendes med eget HTML tagnavn i sin HTML markup. anvendes direkte som brugerdefinerede tags i sin HTML markup.

Web components er altså integrerede i webplatformen. Det kommer med den fordel, at man fjerner behovet for frontend frameworks, hvilket gør størrelsen af ens kildekode mindre. Hvis man alligevel ønsker det, kan man bruge sine web components i ønskede frontend framework. At web component er integrerede i webplatformen betyder, at de bygger på en samling af nye web APIer, som udbydes af alle moderne browsere runtimes. Det er især *Custom Element API* og *Shadow DOM API* som anvendes til web components.

### Custom elements

Custom Element API tillader udvikleren at skabe *custom elements*. Custom elements er brugerdefinerede HTML tags med indbygget funktionalitet, som kan bruges ligesom alle andre HTML tags. Man tilgår API’en med den nye customElements property på det globale window objekt. customElements er et objekt af typen CustomElementRegistry, som tillader at registrere et hvert objekt, som nedarver fra HTMLElement, med et HTML tagnavn. Når objektet er registreret, kan HTML tagget bruges som markup i HTML dokumentet. Selvom det i forvejen er tilladt at navngive sine egne HTML tags uden videre, har man altså brug for Custom Element API for at skabe sammenkoblingen mellem tagnavn og objektet af typen HTMLElement.

Der er forskel på *autonomous custom elements* (selvstændige) og *customized built-in elements* (tilpassede). De selvstændige custom elements skaber man ved at nedarve fra HTMLElement. De har bedre browser understøttelse[^caniuse\_customelements] end tilpassede custom elementer. Tilpassede custom elements udvider i stedet funktionaliteten af browserens indbyggede HTML elementer.

Man knytter sit selvstændige eller tilpassede custom element til et HTML tagnavn med define metoden på førnævnte customElements globale property. Hvis man laver et objekt, der (ligesom browserens indbyggede HTML elementer) nedarver fra klassen HTMLElement, kan man give det objekt med som argument til customElements.define for at registrere sammenkoblingen. Dette vil tillade at bruge custom-element tagget i sin HTML markup ligesom de standard HTML elementer. Indefra klassen har man så adgang til alle de properties, som udgør elementet.

Custom elements har

For at give CustomElement noget mere funktionalitet, er det nødvendigt at kende til dens *component lifecycle*. Component lifecycle er en række metoder på det brugerdefinerede element, som kaldes på forskellige tidspunkter, fx. når elementet tilføjes til DOM eller skal opdateres.

Den første af disse component lifecycle metoder er connectedCallback, som bliver kaldt, når komponenten føjes til DOM. Denne hører sammen med disconnectedCallback metoden, som kaldes, når komponenten fjernes fra DOM.

### Shadow DOM

Shadow DOM er i stand til at give hver web component sit eget indkapslede DOM træ. Det er indkapslet i den forstand, at dette undertræ ikke kan tilgås uden for komponenten. CSS regler i Shadow DOM træet påvirker heller ikke den øvrige DOM. Derfor kan man style sine komponenter uden at frygte, at utilsigtede dele af websiden bliver ramt af komponentens stylesheet. Omvendt er komponenten også beskyttet fra udefrakommende stylesheets.

En lille gevinst ved Shadow DOM er også, at komponenten har adgang til sin Shadow DOM før den er føjet til websidens DOM. Derfor kan man undlade connectedCallback metoden og i stedet bare bruge klassens constructor.

1. <https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html> [↑](#footnote-ref-27)
2. <http://tsdr.uspto.gov/#caseNumber=75026640&caseSearchType=US_APPLICATION&caseType=DEFAULT&searchType=statusSearch> [↑](#footnote-ref-29)
3. <https://www.ecma-international.org/ecma-262/> [↑](#footnote-ref-31)
4. <https://adtmag.com/articles/2018/04/18/javascript-trademark.aspx> [↑](#footnote-ref-33)
5. <https://twitter.com/BrendanEich/status/986605049987002368> [↑](#footnote-ref-35)
6. <https://www.jetbrains.com/lp/devecosystem-2019/javascript/> [↑](#footnote-ref-51)
7. <https://devops.com/survey-sees-massive-adoption-of-microservices/> [↑](#footnote-ref-58)
8. <https://www.leanix.net/en/blog/beyond-agile-time-to-adopt-microservices> [↑](#footnote-ref-61)