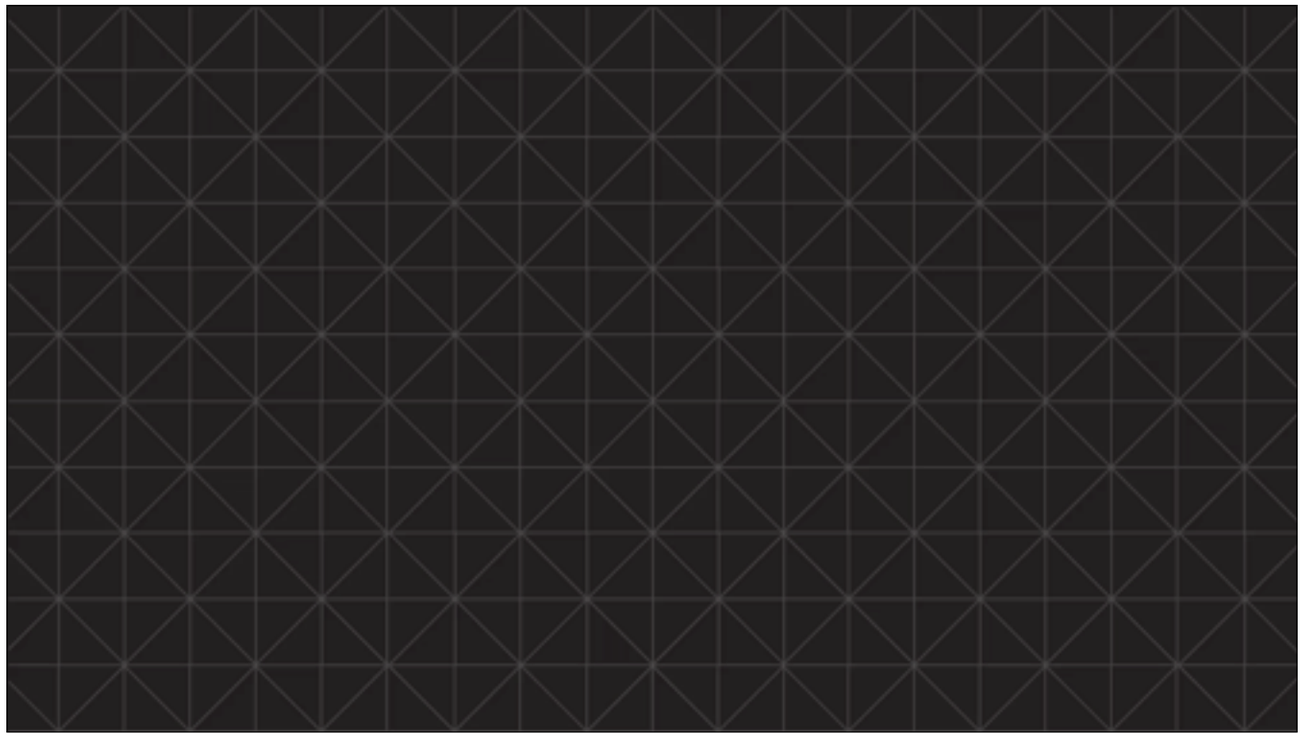


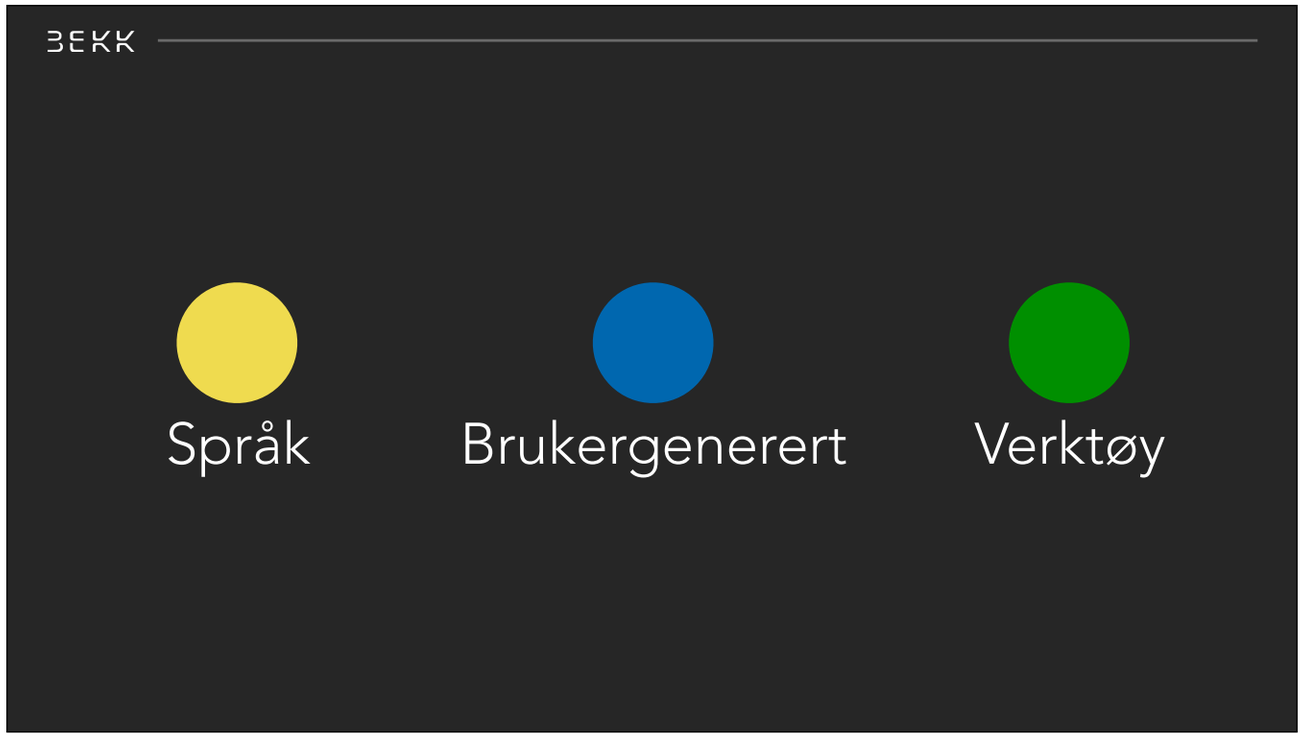

BEKK

FRONTEND ANNO: 2016

EXPLORATION & ITERATION

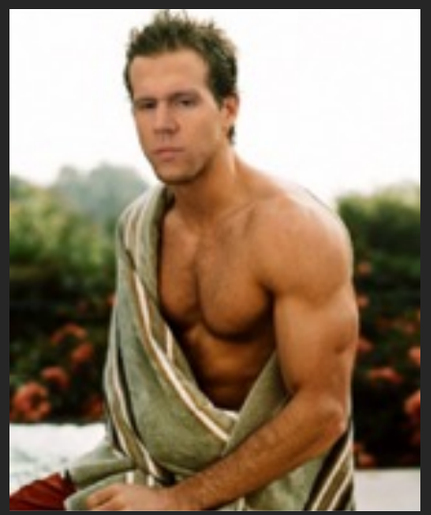
Mikael Brevik
BEKK Open Fagdag, 2016

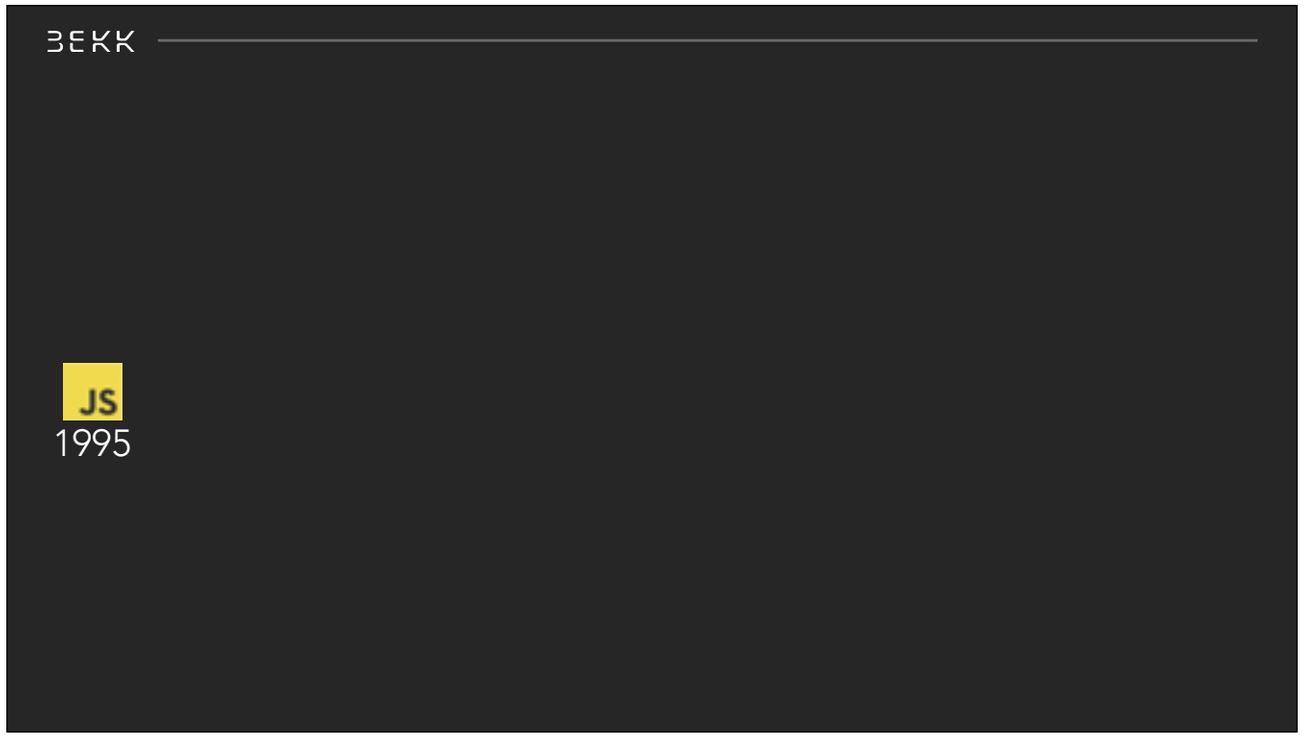
Velkommen til min talk om hvordan frontend er i 2016. Eller mer konkret, hvordan JavaScript er i 2016 og hvordan jeg tror vi er kommet hit vi er.



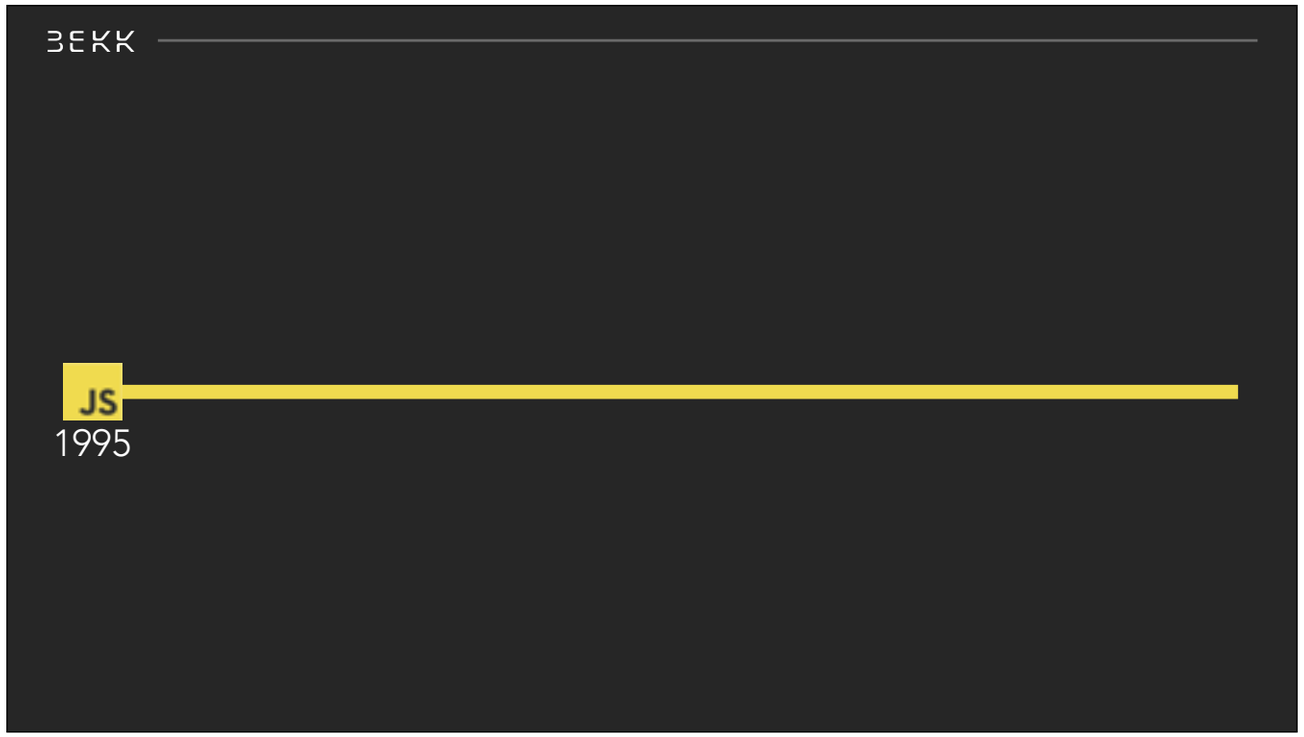
Først kommer jeg til å gå igjennom en timeline av historien delt inn i 3 forskjellige kategorier:

- Språk
- Brukergenerert innhold (rammeverk/biblioteker)
- verktøy



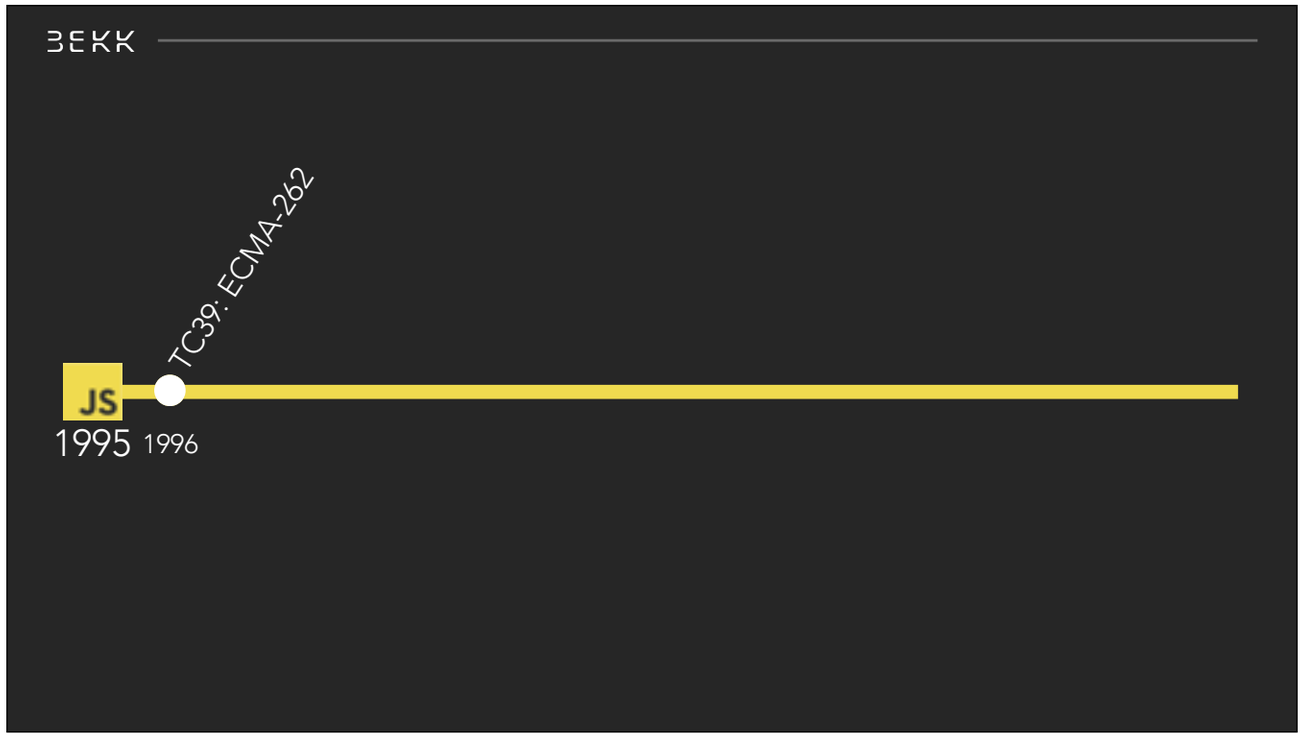


Det hele starta i 1995.

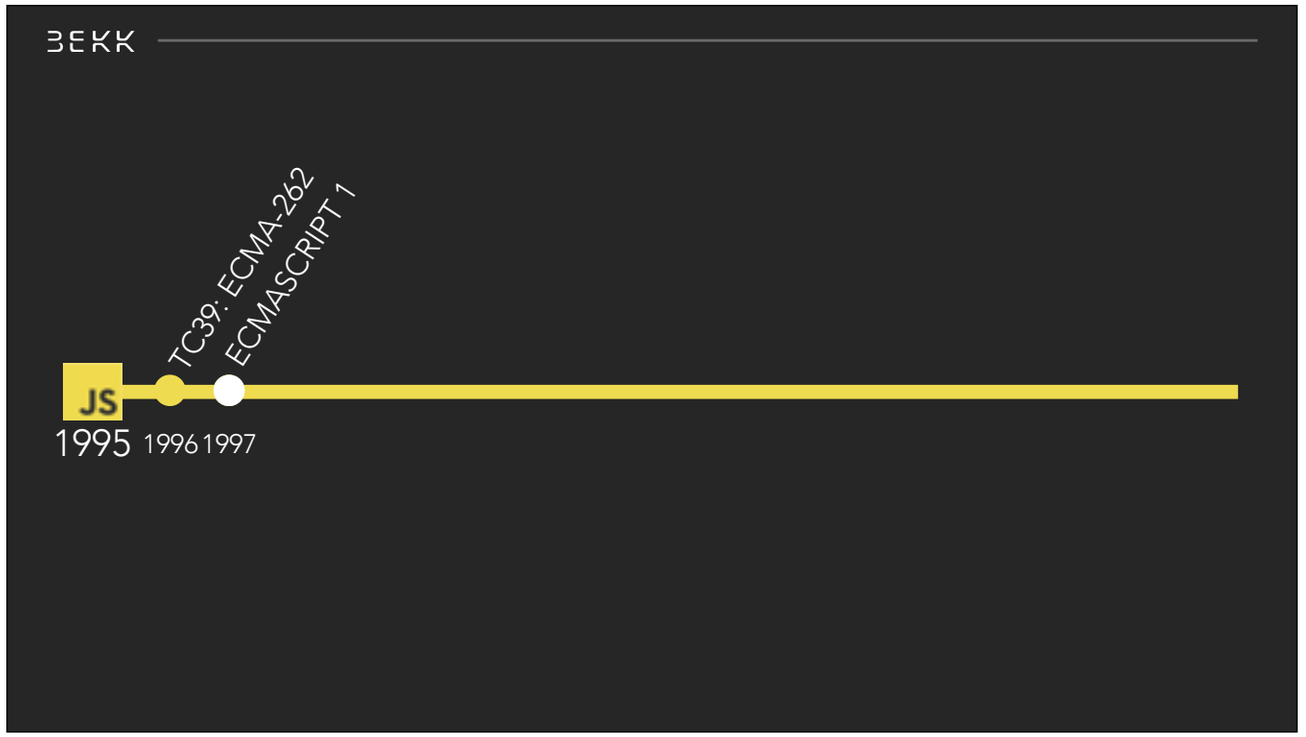


De fleste kjenner vel historien. Netscape skulle ha et språk man kunne bruke i Navigator 2.0, og Brandon Eich fikk en kort tidsfrist på seg til å implementere det. Først ble det kalt Mocha, men deretter LiveScript, men så for å appellere til det populære språket Java, ble det kalt JavaScript - til tross for at det ikke delte noe store trekk med Java. Ikke var det objektorientert, men det var basert på C-språk.

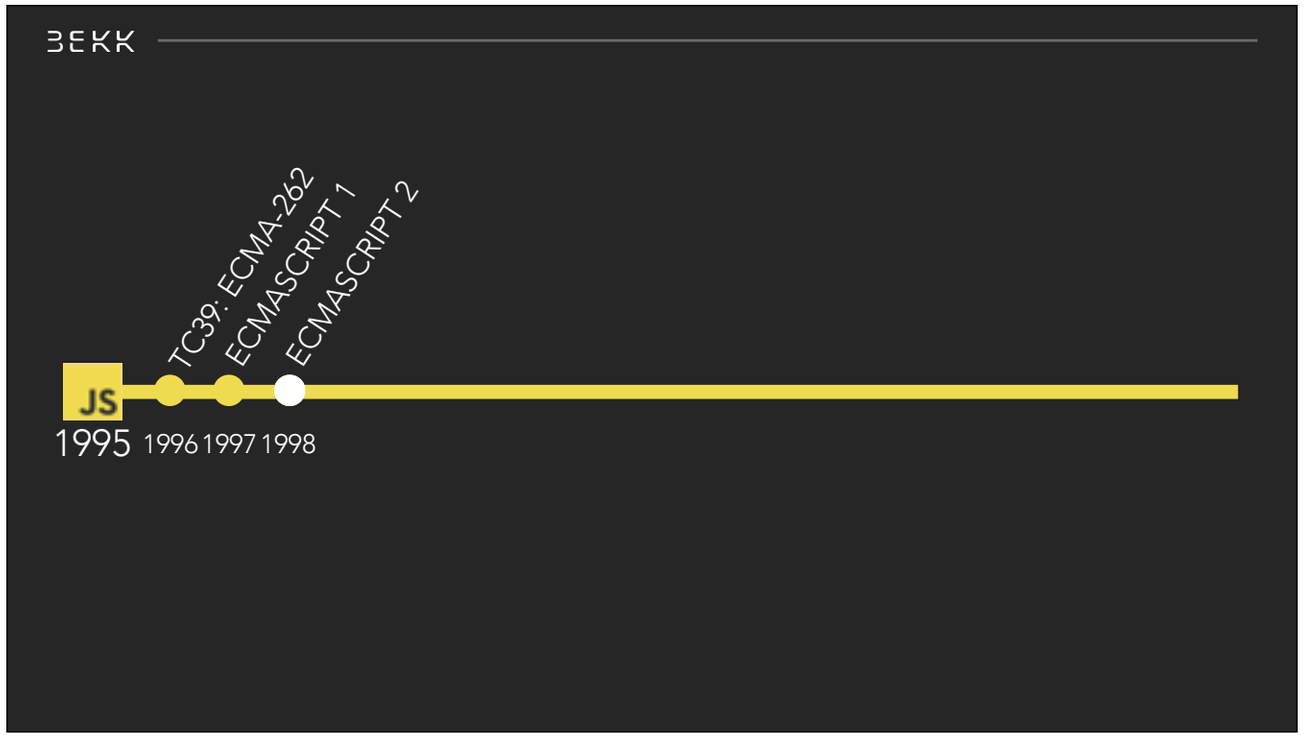
Etterhvert lanserte og Microsoft sin nettleser, IE 3, med støtte for JavaScript, bare under navnet JScript.

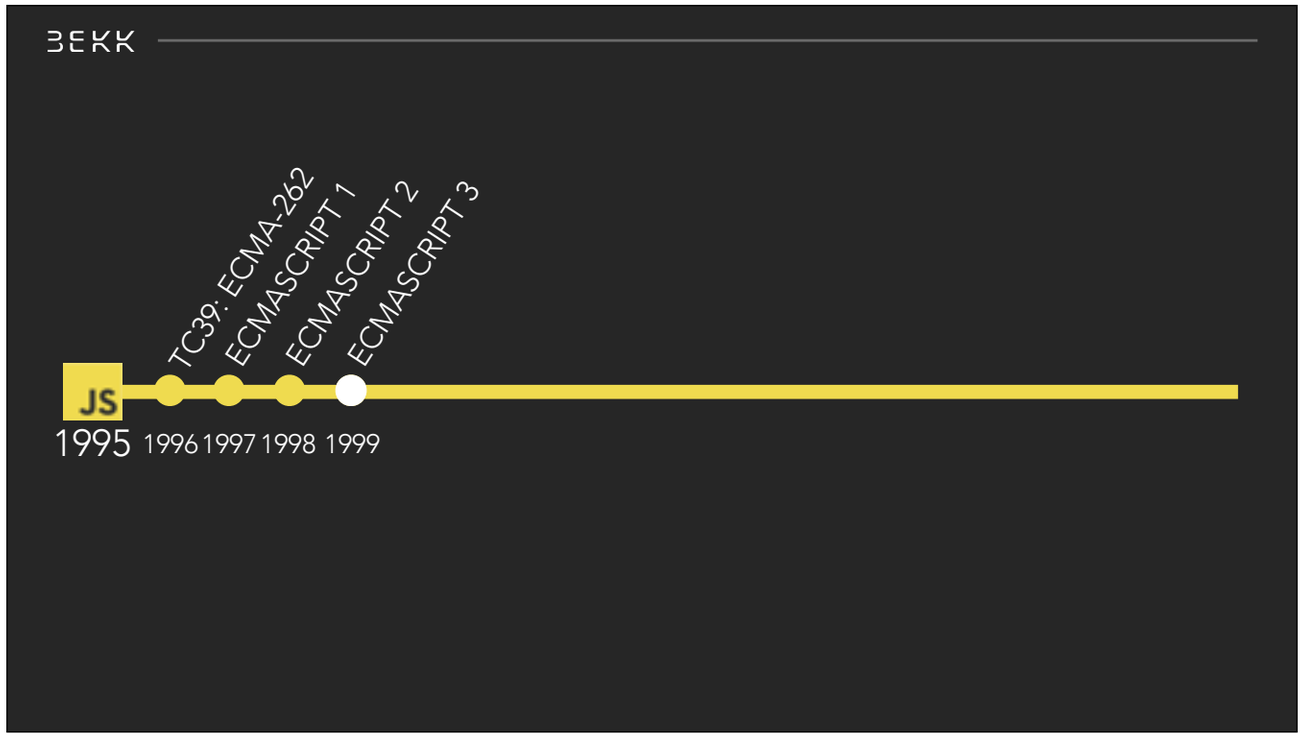


Året etter, blir JavaScript sendt til ECMA-komitéen for å få standarden med det fengende navnet ECMA-262. Komitéen som skulle ha ansvar for standarden heter TC39, eller Technical Committee 39.

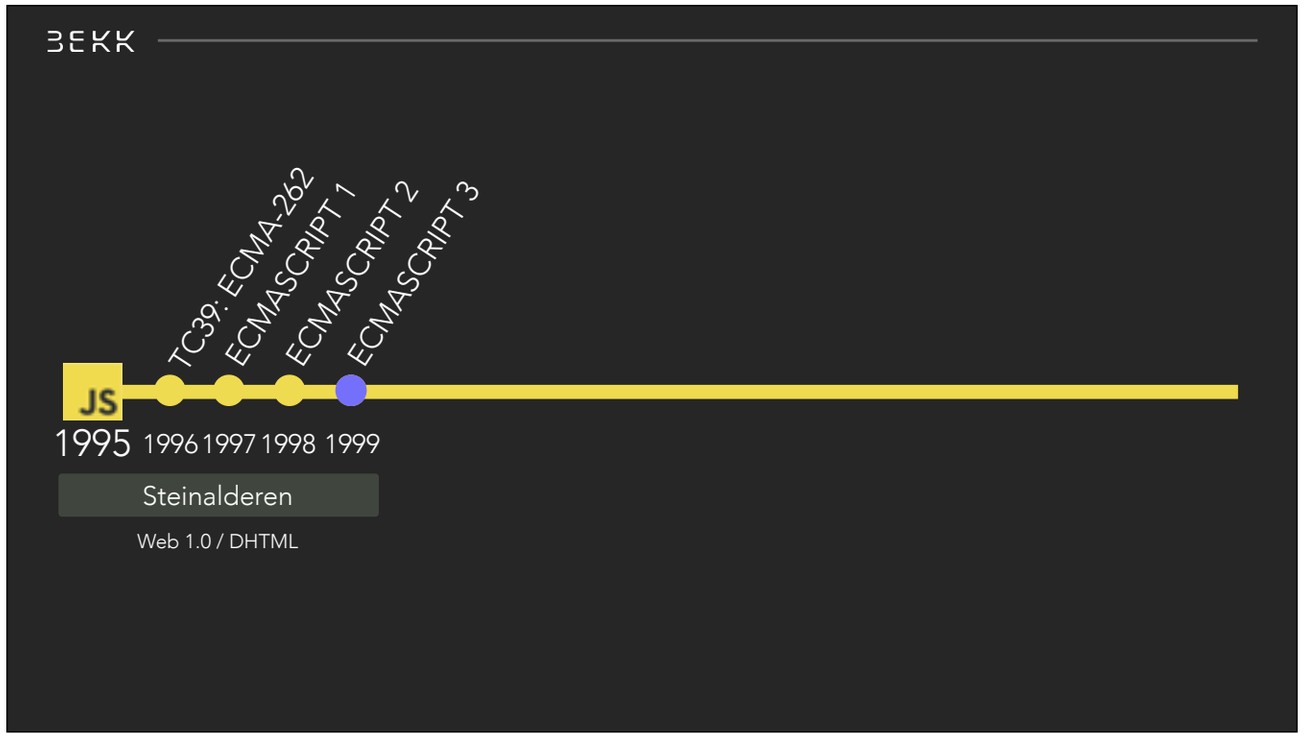


Arbeidet med standarden fortsatte, og i 1997 ble versjon 1 av standarden lansert. Kalt ECMAScript 1.

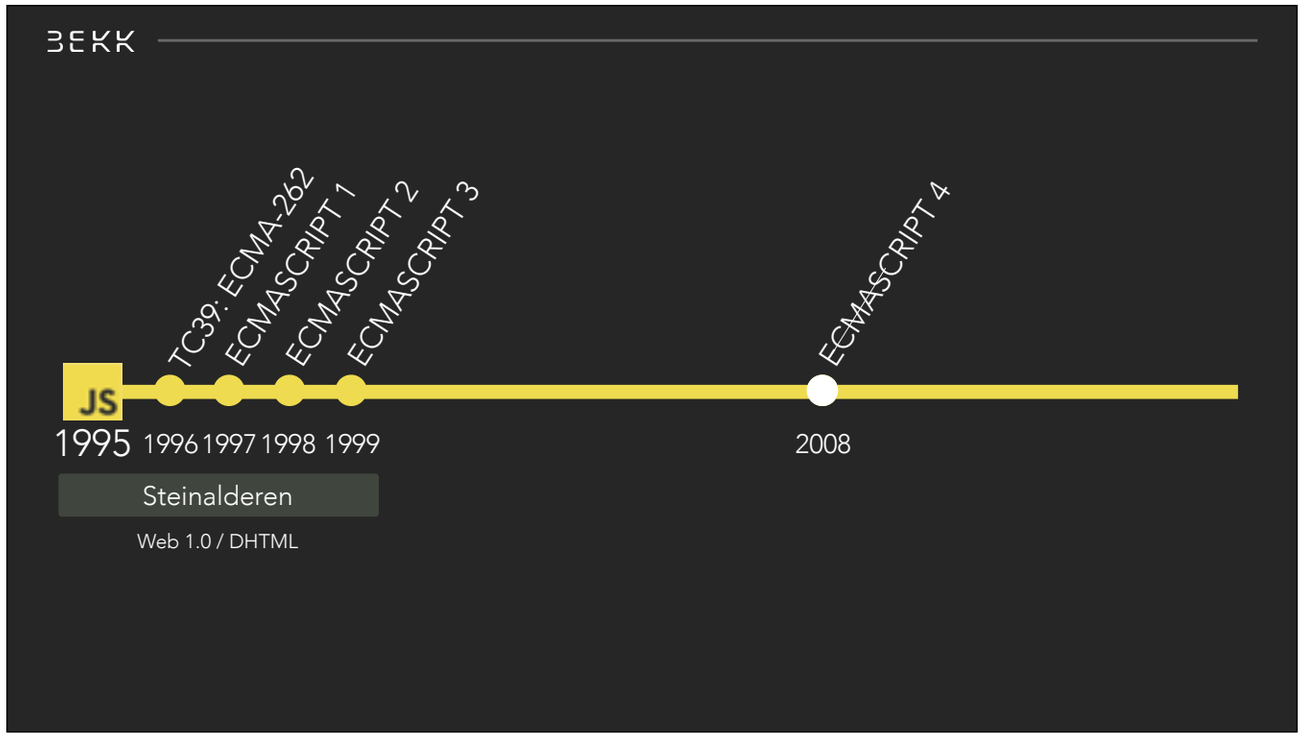

JS
1995
1996
1997
1998
TC39: ECMA-262
ECMASCRIPT 1
ECMASCRIPT 2



ES3 var en stor oppdatering. Her kom ting som regulære uttrykk, try/catch, tall-formatering, og bedre streng-håndtering.

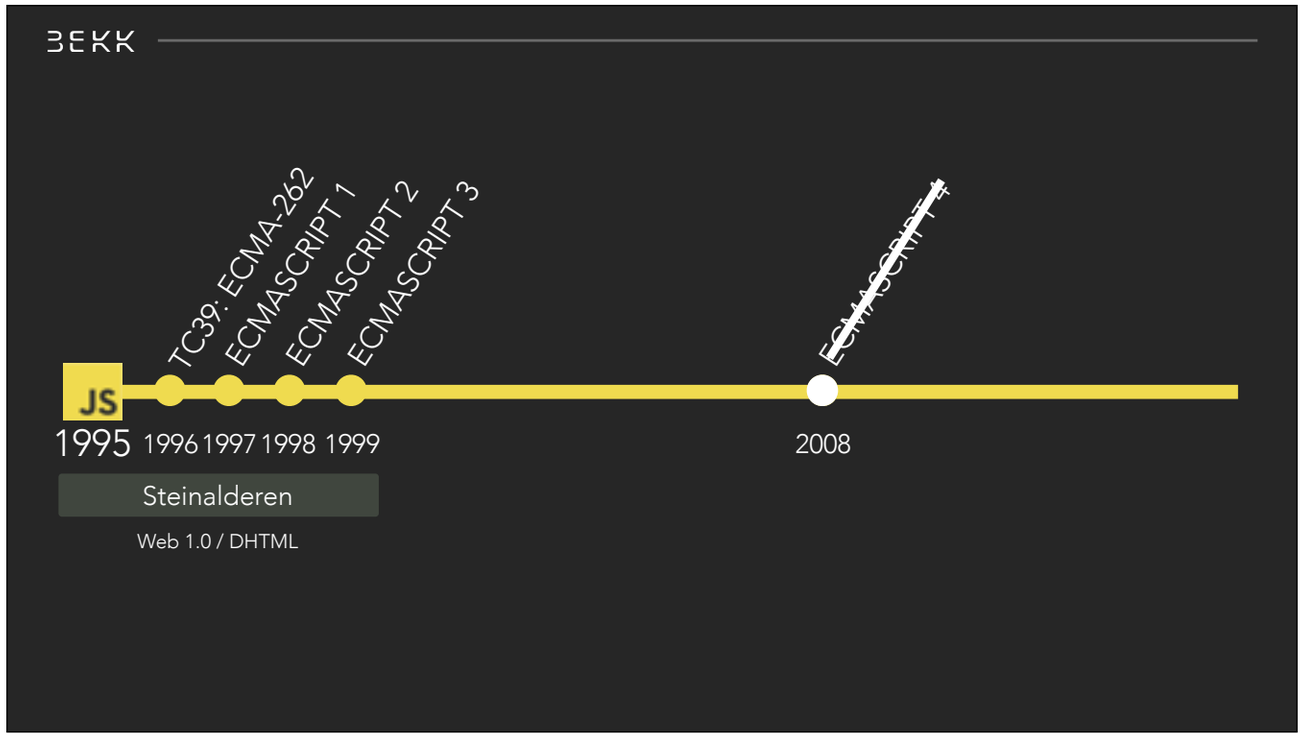


Denne fasen av utviklingen er ofte kalt DHTML. Jeg har kalt den steinalderen. Her var det ikke noe annet enn å kjøre noen små kodesnutter som opererte på DOM-en. Man kunne ikke hente noe ekstern data på noen måte. Dette var en stor periode av copy/paste utvikling. F.eks hvordan få ting til å automatisk scrolle. Hvordan få til marquee osv.

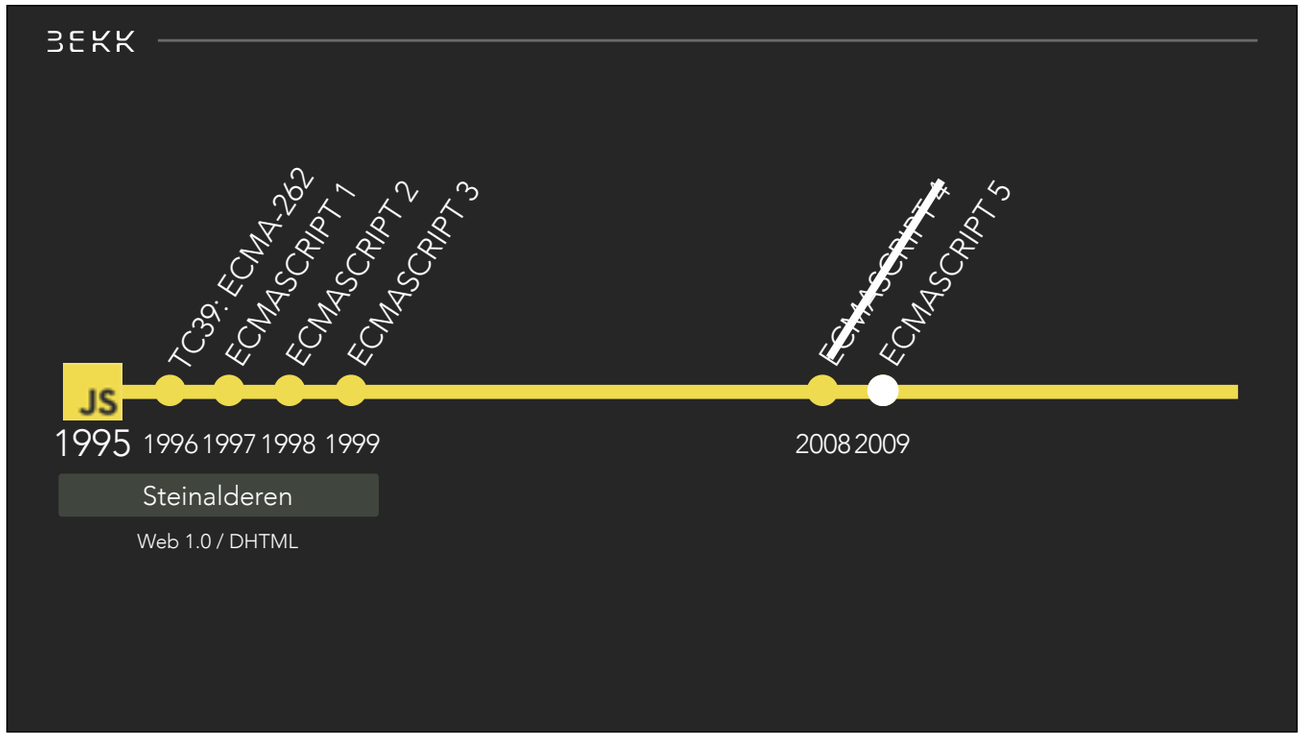


Så, etter nesten 10 år uten noe ny versjon skulle det komme en stor oppdatering i språket.
Ting som var diskutert til ES4 var klasser, modul-håndtering, destructuring, statisk typing, generatorer og iteratorer og algebraiske datatyper.

As a side note: Blant annet av features som var foreslått var ECMAScript for XML. Som egentlig er ganske likt JSX som ikke er en del av ECMAScript i dag, men som Facebook bruker i sitt rammeverk.

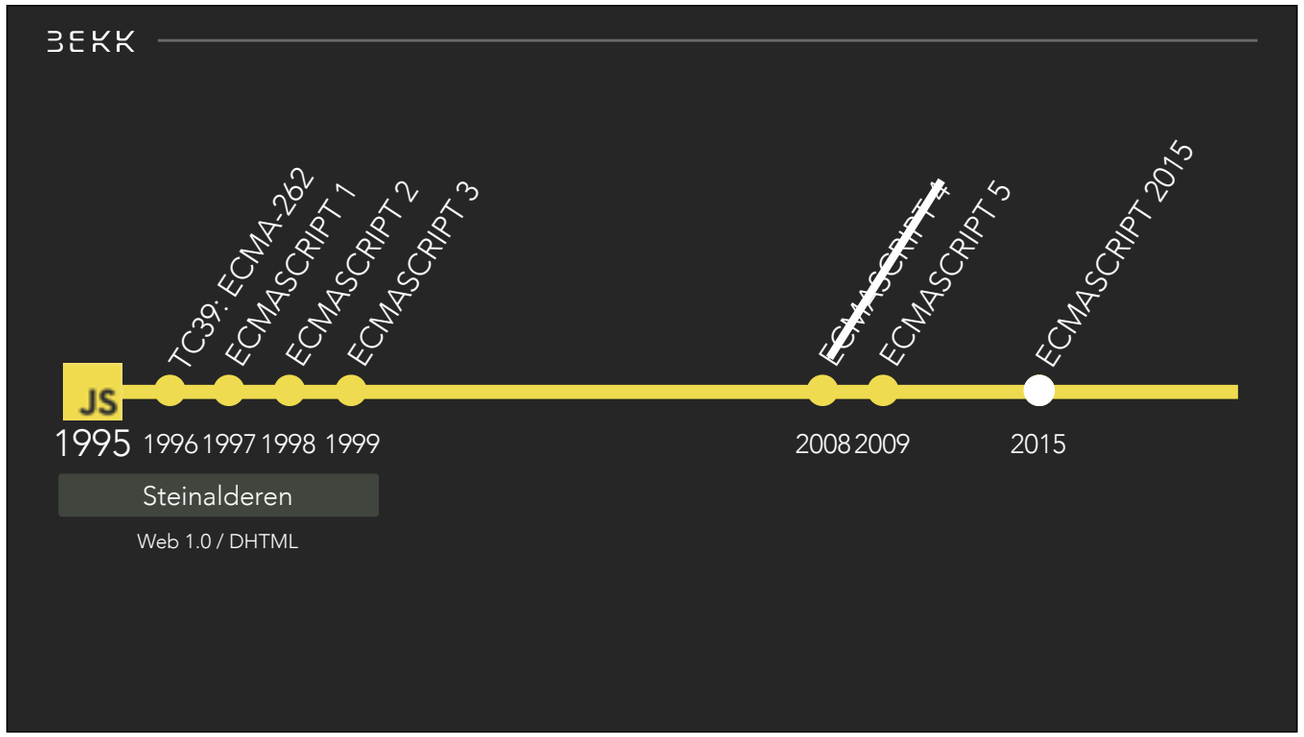


Men det ble for mange interessenter og de kunne ikke bli enige om features. Det var mange kontroverser og anklagelser i forskjellige leirer. Det endte opp med å bli forlatt. Et språk som tok i bruk ES4 var ActionScript, selv om det ikke ble en del av standarden.



Etter ES4 “fjaskoen”, møtte blant annet Yahoo (med Douglas Crockford), Microsoft og Google igjen, men denne gang som en mindre komité og bestemte seg for å oppdatert ECMAScript 3 men på en litt mer kontrollert måte. Det ble igjen mye frem og tilbake, men endte opp med ES5 til slutt i 2009.

Ikke like omfattende som ES4, men introduserte strict-mode som er et subsett av JavaScript som er strengere og gir flere feilmeldinger. F.eks dersom man bruker udeklarerte variabler. Andre ting som ble innført var mapping, filter og reduce over lister. Muligheten til å definere getters og setters via f.eks object.defineProperty og bedre object reflection.



These goals were agreed on at a meeting in July, 2008:

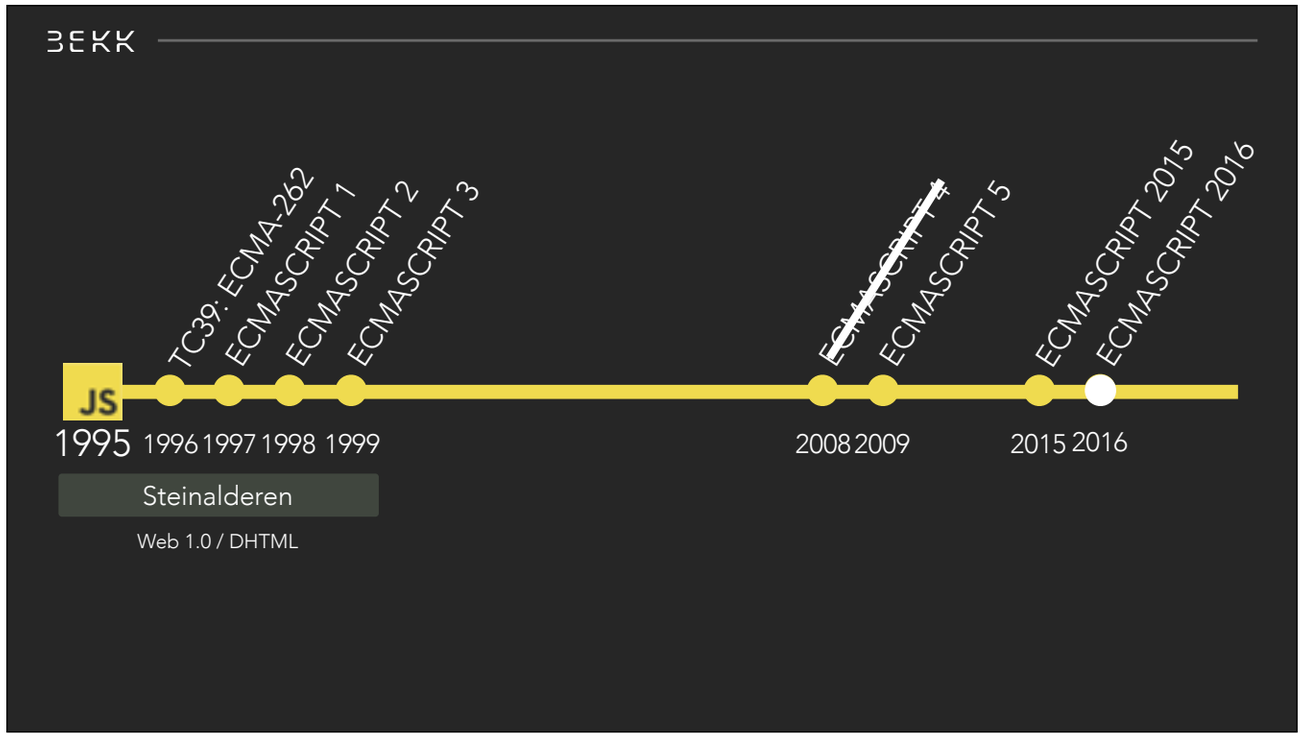
1. Be a better language for writing:
    1. complex applications;
    2. libraries (possibly including the DOM) shared by those applications;
    3. code generators targeting the new edition.
2. Switch to a testable specification, ideally a definitional interpreter hosted mostly in ES5.
3. Improve interoperation, adopting de facto standards where possible.
4. Keep versioning as simple and linear as possible.
5. Support a statically verifiable, object-capability secure subset.

—-

Så begynte arbeidet til det som nå blir kalt ECMAScript 2015. Her skjedde det og skjer det mye. Standarden er låst, men ikke alle nettlesere har implementert alt enda.

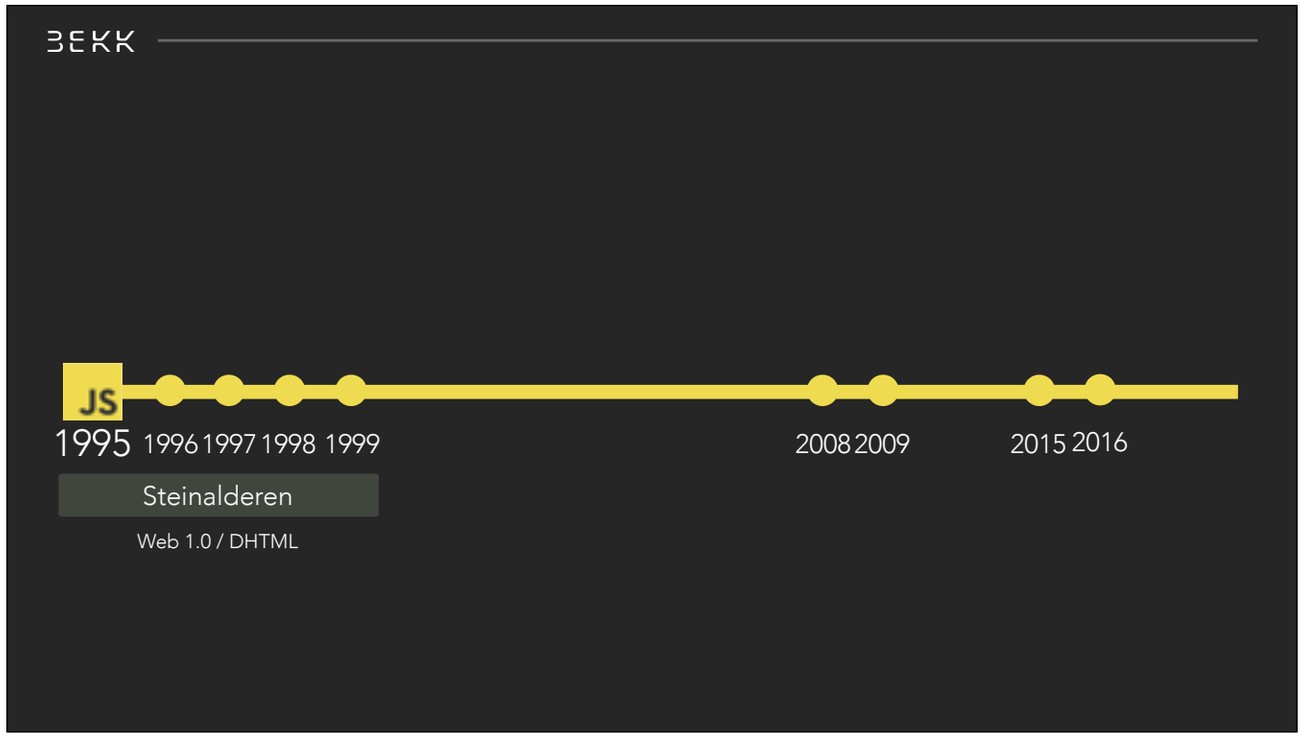
Arbeidet gikk over flere år, men ble først fullført i 2015. Det ble og inngått en enighet om at standarden skulle ha årlige releases og har derfor gått over til å navngi med årstall.

Featurelisten er massiv, men noen av de med mest impact er arrow functions, “klasse”-lignende syntaktisk sukker, iteratorer, lisp-aktig destructuring, Maps og Sets, Språk proxier, bedre støtte for reflection, symboler, binær data, bedre støtte for matte (f.eks med støtte for negativ 0), osv.



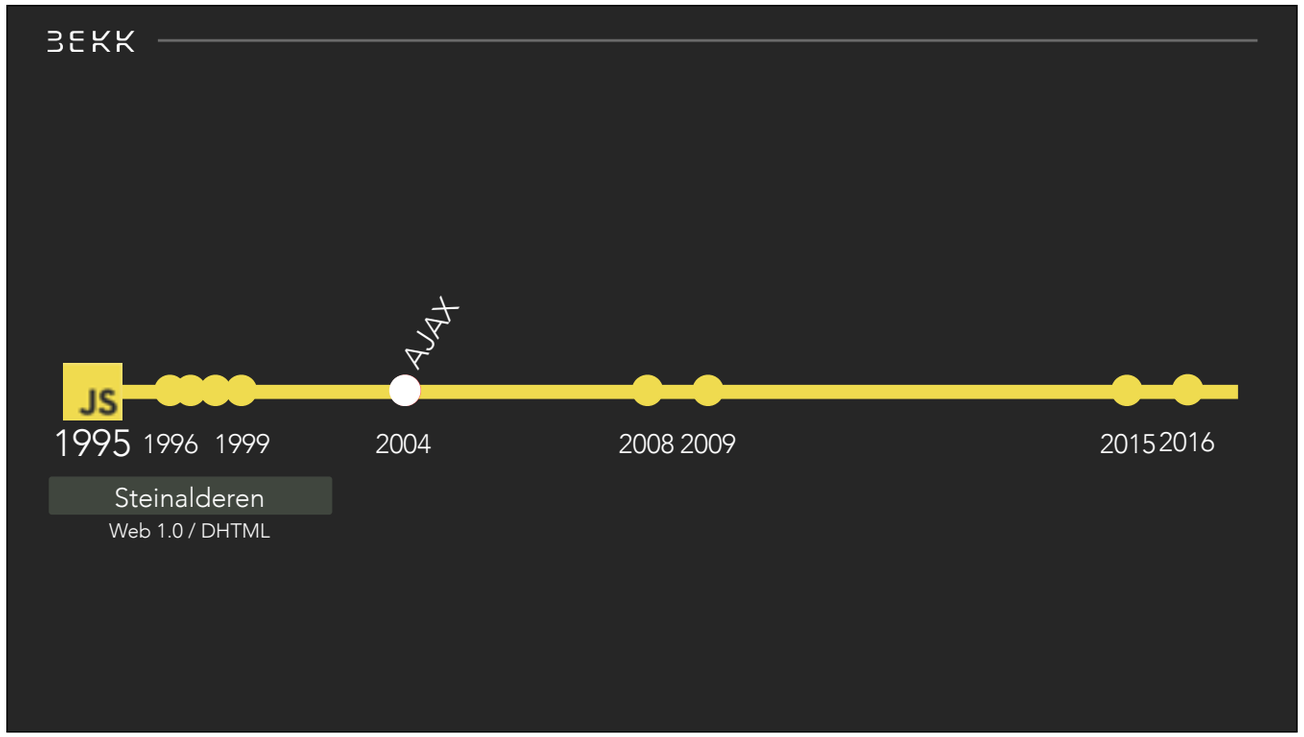
Ettersom arbeidet på standarden nå skal være årlig, er det og utvidelser til språket i år. Men det er langt fra like mange features og høy hastighet i denne releasen.

Det er to ting som blir innført:
Array.prototype.includes
og 2 ** 2 (potens operator)



Det er hele 10 år før det kom en oppdatering til ECMA-standarden mellom ES3 og ES5. Og i mellomtiden der skjer det veldig mye i bransjen, og der tror jeg vi har litt av kjernen til JavaScript, om du vil. Som kan forklare litt av hvordan plattformen er.

For hva skjer dersom det ikke er noe endringer på språk siden og ting har stagnert? Brukersiden kan ta opp stafettpinnen og innovere. Mye av det var mulig på grunn av og eller samtidig som en stor ting skjedde i nettleser-API-land.

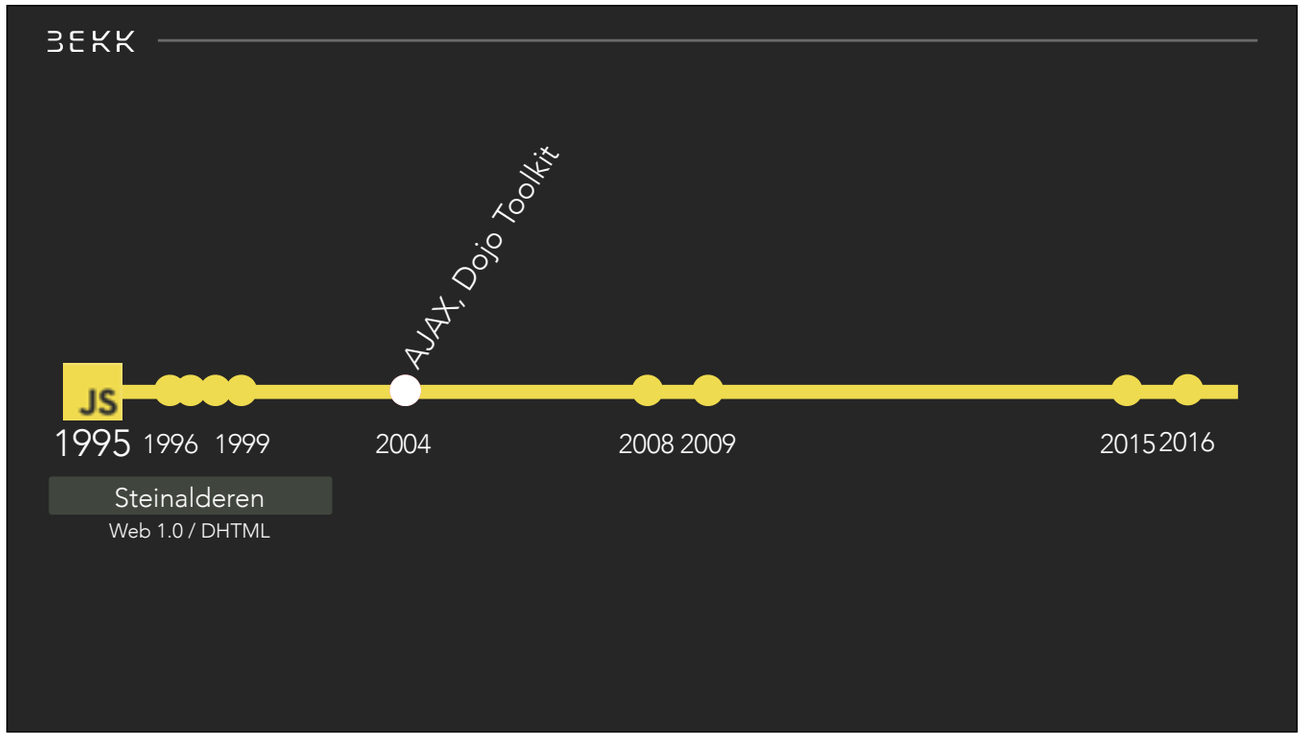


AJAX ble implementert i de store nettleserene!

Arbeidet med AJAX hadde egentlig foregått lenge. Hvor Microsoft startet med en ActiveX implementasjon tilbake i 1999. Men først i 2000 implementerte Mozilla sin versjon av det, og i 2004 og 2005 ble det støtta av Safari og Opera.

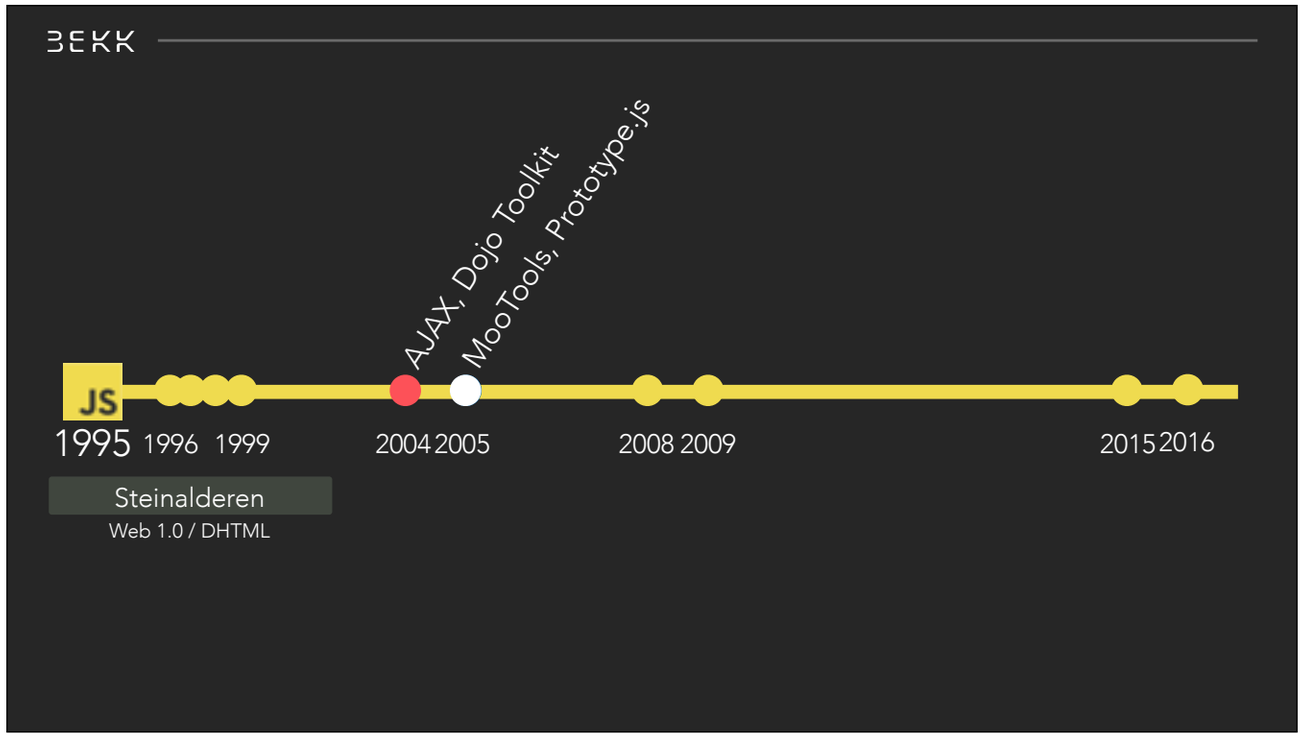
Men plutselig ble kanskje JavaScript mer aktuelt igjen. Fremfor å laste hele siden på nytt hver gang, kunne man hente et lite utsnitt av sidene og laste det.

JSON, som da egentlig bare er et subsett av Javascript objekter, var også etablert i tidlig 2000 og de kunne kombinere AJAX og JSON til å hente data og vise den i klienten. En massiv revolusjon og man kan se det reflektert i brukerverden, for nå begynner det å komme store rammeverker som etablerer web-en som en plattform får å kjøre kode i en litt mer seriøs forstand.



Først ut var Dojo Toolkit. Dojo Toolkit starta i 2004, med en e-post titulert “Selling the future of DHTML”, etterfulgt av en sommer av jobbing og diskusjoner. Dojo kan og ansees som en stor success for open source, for innen 2005.

Dojo Toolkit hadde og har mange widgets for oppbygging av grensesnitt utelukkende på klienten. Dojo er fremdeles i bruk i dag av enkelte tjenester og CMS-er. F.eks EPiServer bruker det enda som sitt fundament for redaktørgrensesnittet.



Neste store utviklingen var rammeverk som MooTools og Prototype.js.

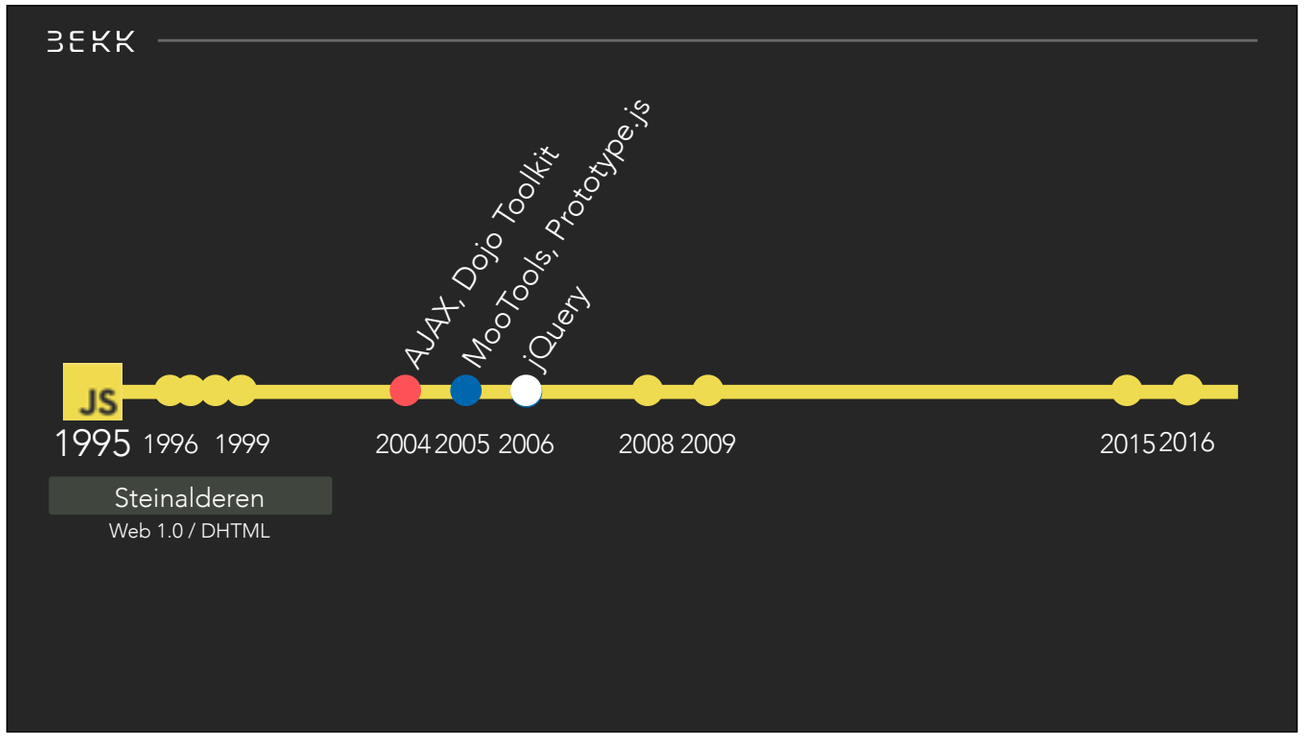
Prototype.js kan egentlig sees på som et direkte resultat av lite utvikling i språket i seg selv. Den hadde fokus på å utvide kjernespråket i seg selv med nye funksjoner. Utvide prototypen i JavaScript. Prototype prøvde å legge opp manipulasjon av innebygde datatyper.

Tror og at Prototype.js var en av de første rammeverket som tok i bruk $-syntaksen som en hjelper mot å koble til DOM-en. De hadde og en måte å simulere klasser på og gjøre AJAX enklere.

For man måtte gjøre AJAX enklere. Det var mange forskjellige implementasjoner og implementasjonene var tungvindt å bruke.

Mootools er og en viktig milepæl for JavaScript. Det var inspirert av prinsippene bak Prototype.js, men i tillegg til å utvide prototypene til kjernefunksjonalitet i JavaScript som strings, arrays, functions osv, tok MooTools det et steg lengre og utvidet DOM-objekter med funksjonalitet og.

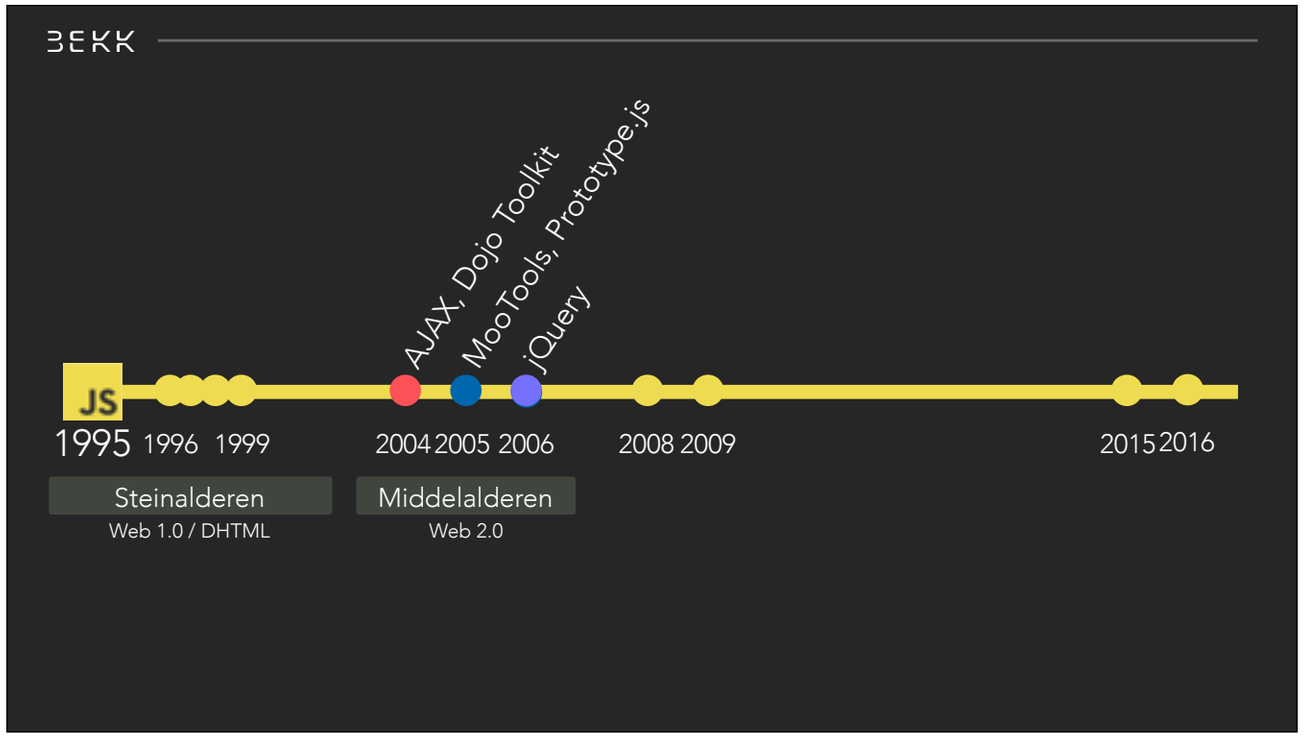
Mange patterns vi bruker i dag ble først konseptualisert via MooTools og. F.eks module pattern. MooTools hadde og en måte å simulere klasser på.



En gigant av denne æran som trolig alle kjenner til er jQuery. Alle verktøy her er fremdeles vedlikeholdt, men det er kanskje jQuery som er det som fremdeles blir en del brukt - selv om det kanskje er i ferd med å avta litt.

jQuery var igjen basert på et bibliotek som kanskje ikke mange kjenner til, men som het Behaviour. Det prøvde å kombinere CSS selectorer med JavaScript. John Resig, som skapte jQuery, var ikke helt fornøyd med implementasjonen og mente den manglet mye. Det var på mange måter starten av jQuery.

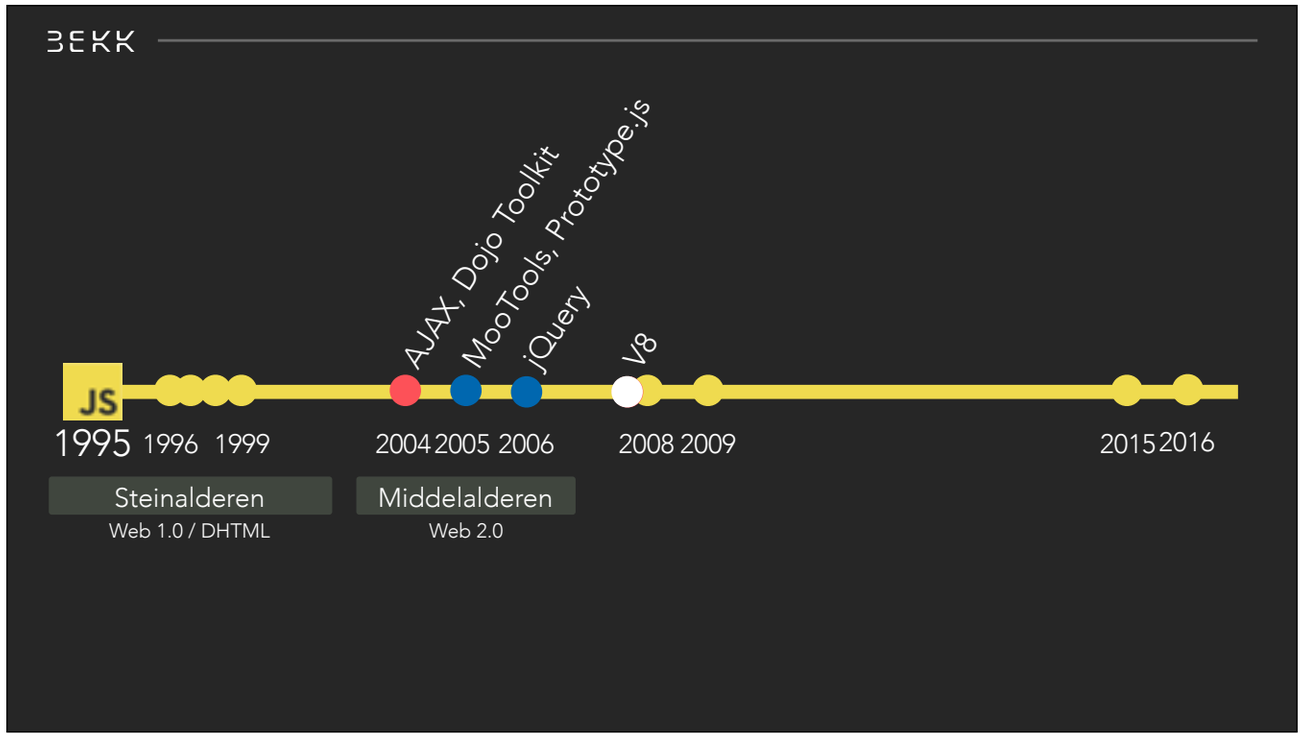
jQuery har fokus på operasjoner mot DOM-en, enklere ajax-kall og gode muligheter for plugins. Dette var fokuset helt fra starten, og det er på mange måter fremdeles fokuset i dag. Vil påstå at jQuery endret helt måten vi kunne skrive grensesnitt på.



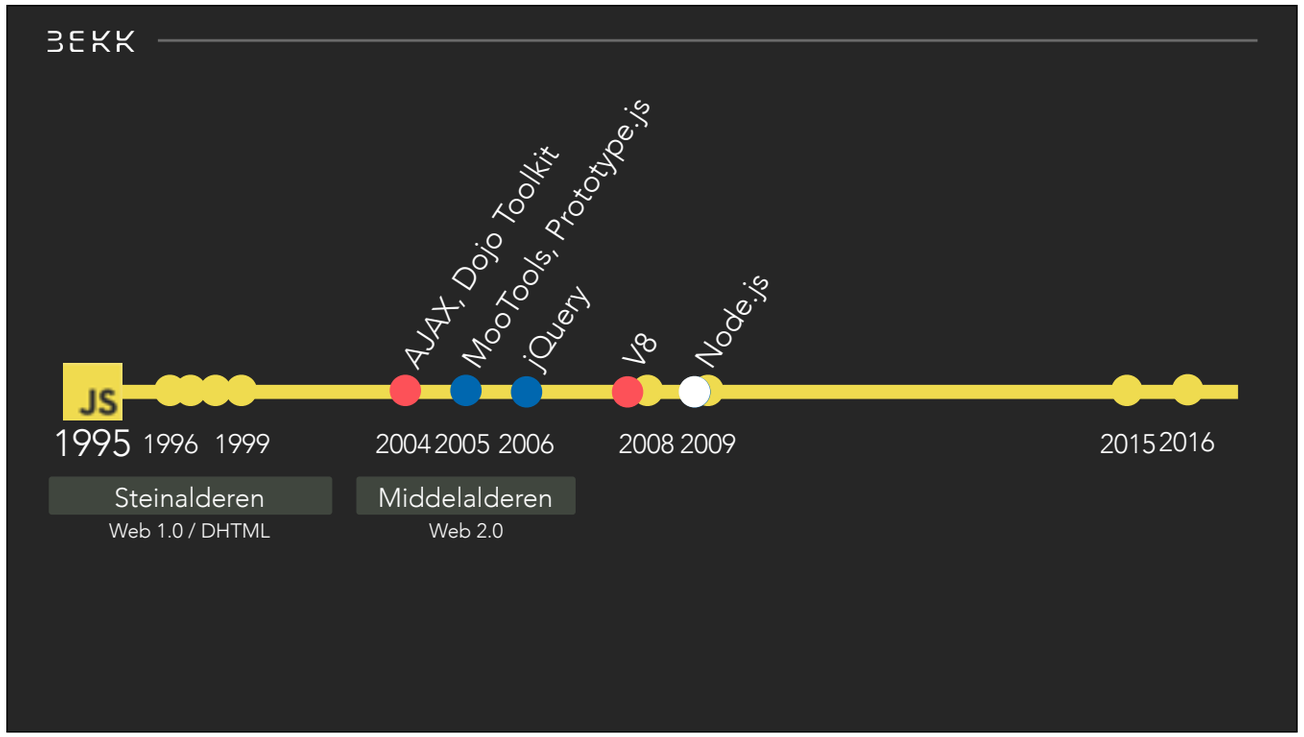
En annen ting som er interessant er å se hvordan de forskjellige implementasjonene påvirket hverandre. MooTools var basert på Prototype.js, jQuery så på mindre biblioteker, men tok og en del fra både MooTools og Prototype.js, som en bedre måte å gjøre AJAX på. Vi ser at de bygger på hverandre blir bedre av det. Selv om det var flere konkurrerende rammeverker, vil jeg påstå at jQuery er den fleste kjenner til. Mange kjenner sikkert fremdeles til Prototype, MooTools og Dojo, men vanskelig å argumentere for at jQuery ikke var den med størst impact og fremdeles er ganske viktig en dag i dag.

Denne fasen av web-utvikling blir ofte referert til som Web 2.0, eller middelalderen som jeg kaller den. Her begynte vi å få skikkelig verktøy som var litt mer sofistikert. Det som kjennetegner Web 2.0 er AJAX-mulighetene og større grad av interaktivitet med grensesnittet og dynamisk oppbygging.

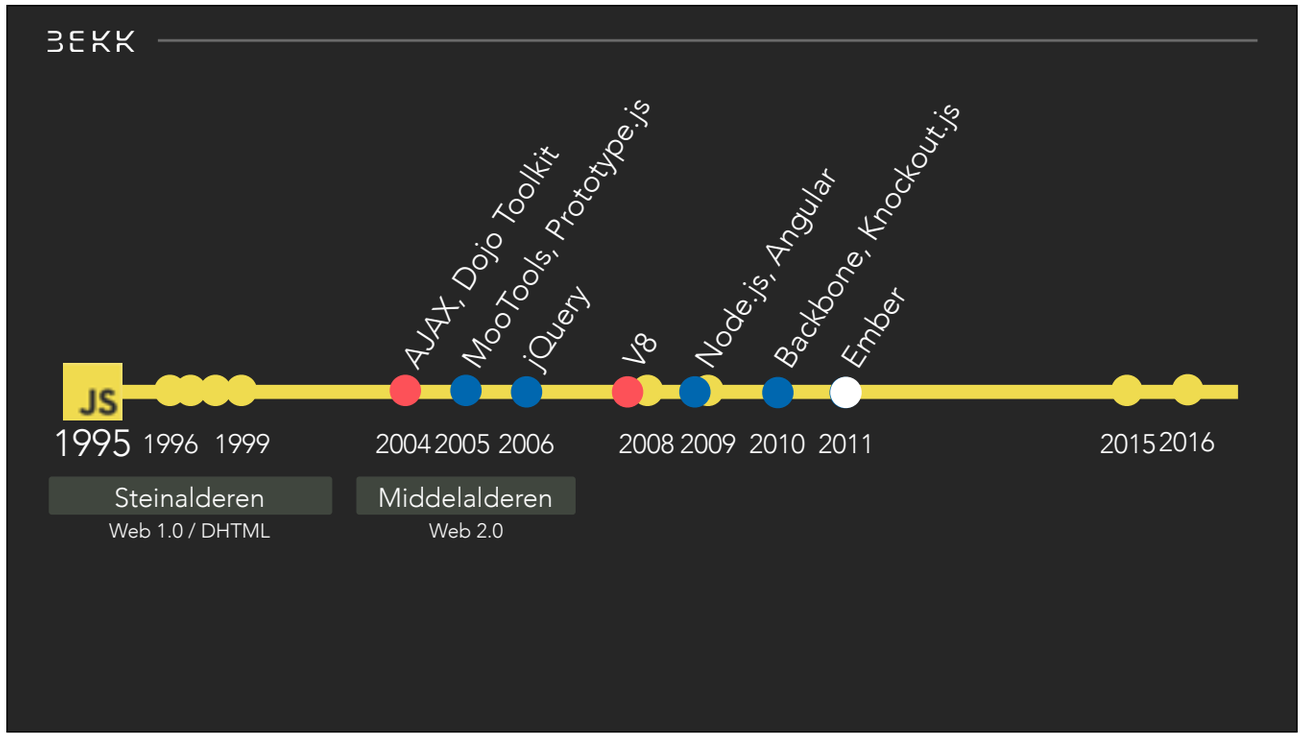
Dette var og perioden jeg gikk fra å være dødskul i solbriller til å være en av de første trendsettere med duck-face.



Neste store milepæl var nye JavaScript motorer. I 2008 lanserte Google V8 som plutselig tillot mye bedre performance for grensesnitt-programmering.



En annen ting det tillot var Node.js. Node.js tok V8-motoren og kombinerte det med et API skrevet dels i JavaScript og dels i C++ for å kunne kjøre JavaScript på serversiden. En ting som var interessant med Node.js var at det ble rapportert gode tall for håndtering av asynkrone IO oppgaver, grunnet JavaScript natur med eventloop osv. Men det som er like så interessant er økosystemet som har blitt bygd rundt Node.js i senere tid. Både for serverside men og klientside-kode og ikke minst verktøy for web-en.



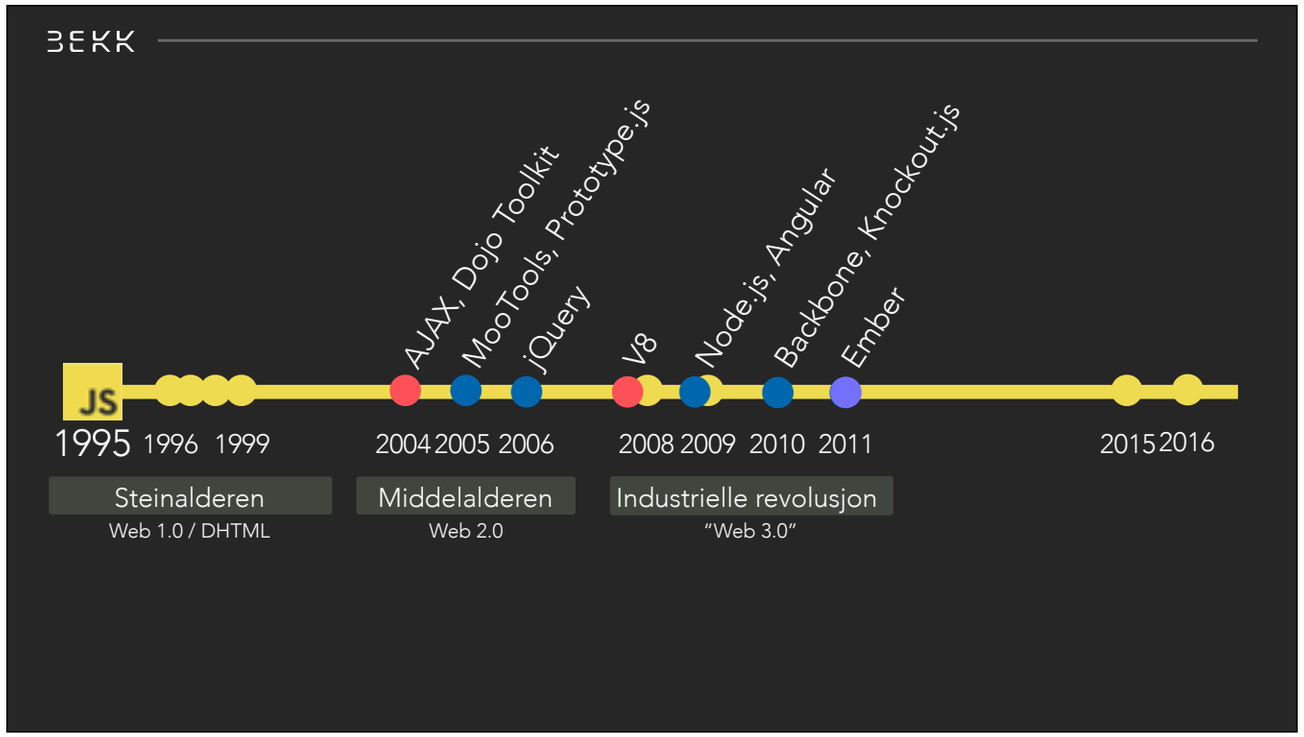
Økende popularitet og forventninger til klientkode og interaktivitet gjorde at mer og mer kode ble flyttet over til klientsiden. Dette gjør at det ble krav til større grad av struktur og arkitektur i løsningene for å kunne være vedlikeholdbar. På lik linje som man trenger arkitektur i koden på serversiden trenger man det og i klientsiden. Dette er da single page applications virkelig begynner å bli en ting, og det kommer mange løsninger som skal gjøre det på en god måte.

I 2009 og til 2011 var det veldig mange som skulle løse dette problemet. Det startet i 2009 med Angular. Angular var egentlig tiltenkt som en løsning for designere/ux-folk med erfaring i HTML/CSS til å kunne lage single page applications. Det utviklet seg til å bli et stort rammeverk med løsninger for services, controllere for dependency injection, og mange andre arkitekturiske patterns man vanligvis ser i server-side-verden.

En annen bauta var Backbone som var et angrep mer fra Ruby og REST-verden. Det var fokus rundt MV*, med modeller, collections og views. Sentralt står koblingen opp mot ressurser i RESTfulle tjenester.

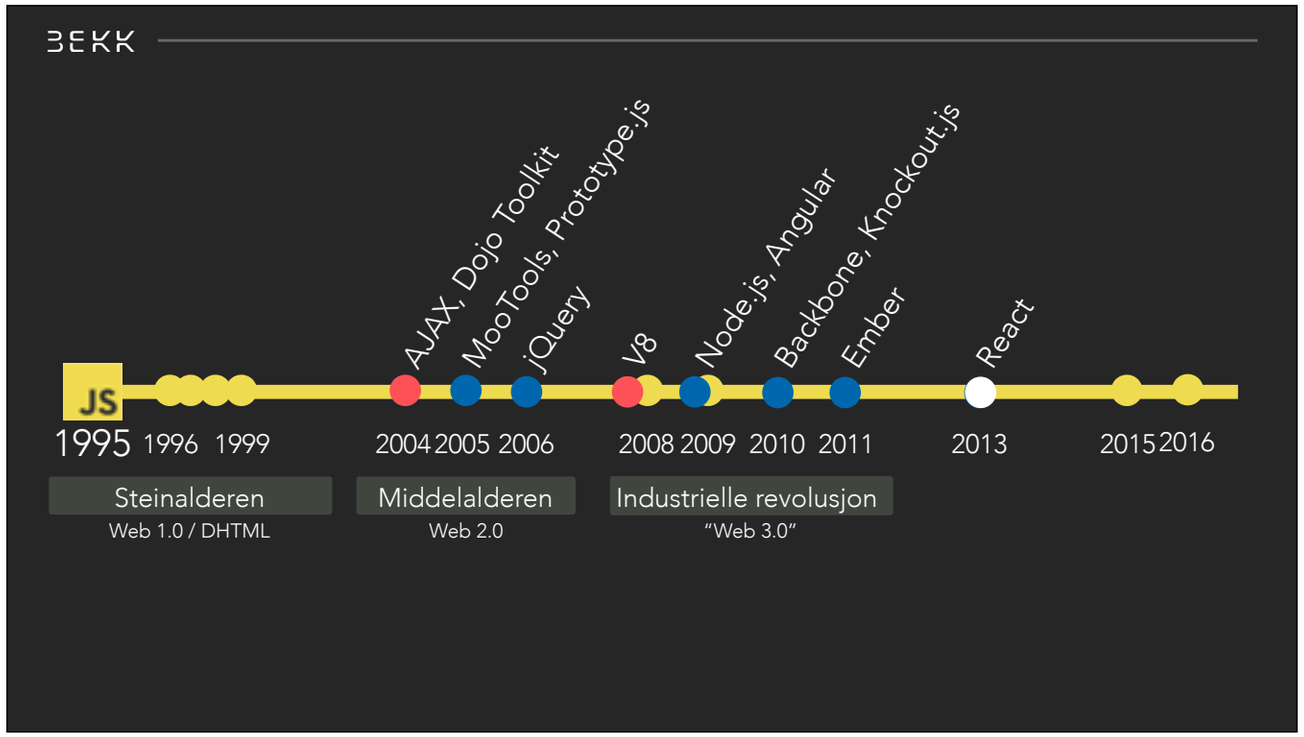
Microsofts løsning, Knockout.js, har en litt annen tilnærmingsmåte. De fokuserte på to-vegs-bindinger via observables mellom modeller og views. Knockout.js fokuserer mer på MVVM.

Ember var en av de siste store som ble introdusert i den perioden. De satser på MVC-pattern og har og inspirasjon fra Ruby-on-rails-verden.



I utgangspunktet, da Tim Bernes-Lee ble spurt om hva Web 3.0 er, var svaret bevegelsen om Semantisk Web som bevegde seg ut fra 2006.

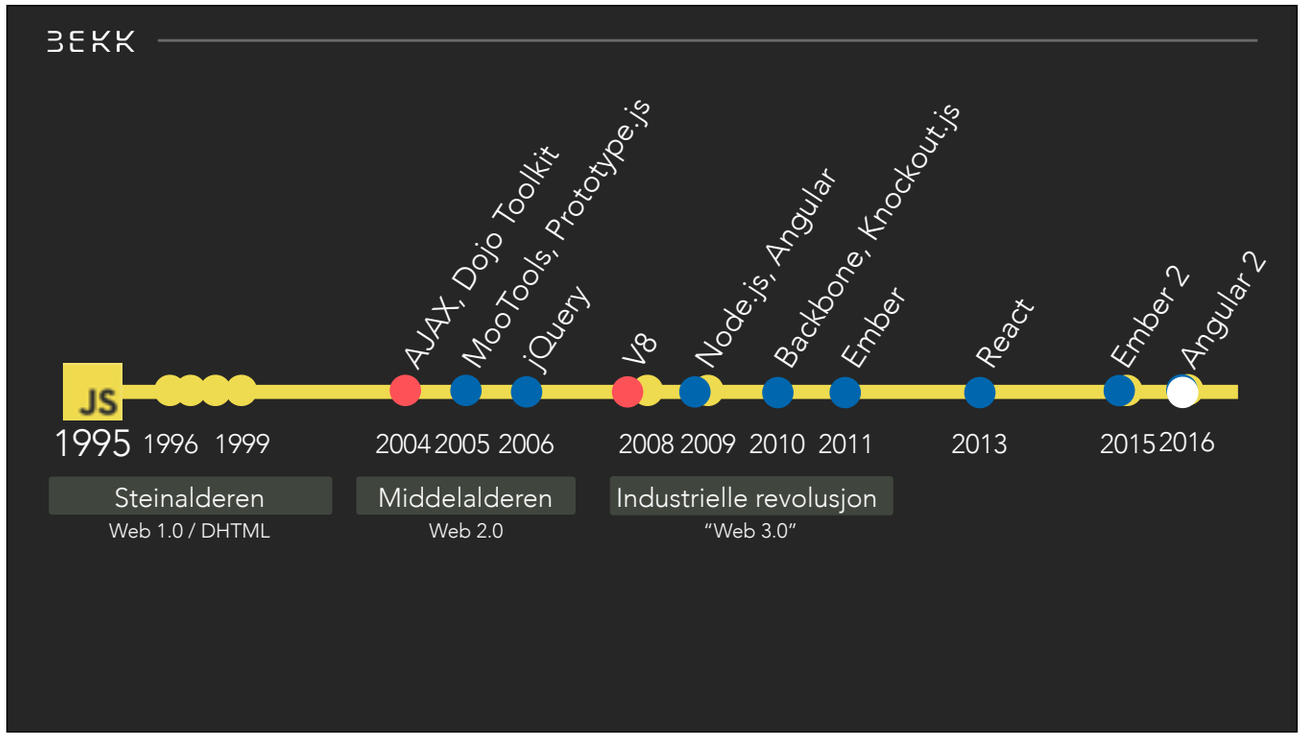
Denne perioden har jeg valgt å kalle den industrielle revolusjon. Ikke så mye for at industrien revolusjonerte, men at da både V8 og ting som Node.js ble industrien revolusjonert.



Samtidig som gigantene Angular, Backbone, Knockout, Ember herjet var det mange små biblioteker som utforsket, jobbet parallelt, testet ut med forskjellige måter å programmere grensesnitt på. I 2013 kom en viktig aktør på banen. Facebook kom med React.

Med en talk med navnet “Rethink the Web”, lanserte Pete Hunt et bibliotek som tenkte på en helt annen måte enn de eksisterende store. Det var disruptivt og nyskapende med noen fancy algoritmer som som skulle optimalisere de kostnadsfulle operasjonene opp mot DOM-en. Endre på modellen vi programmerte grensesnitt. I tillegg var det introdusert en ny syntaks på toppen av JavaScript som de kalte JSX. Som vi vet nå, er ikke dette egentlig noe nytt, men er bygd på idéer som Mozilla hadde tilbake da de snakket om ECMAScript 4 og ES4Xml. Og jeg *hatet* det. En latterliggjøring av web-en. Backbone med HTML. La det tvert fra meg som en tåpelig kurriositet som aldri kom til å ta av.

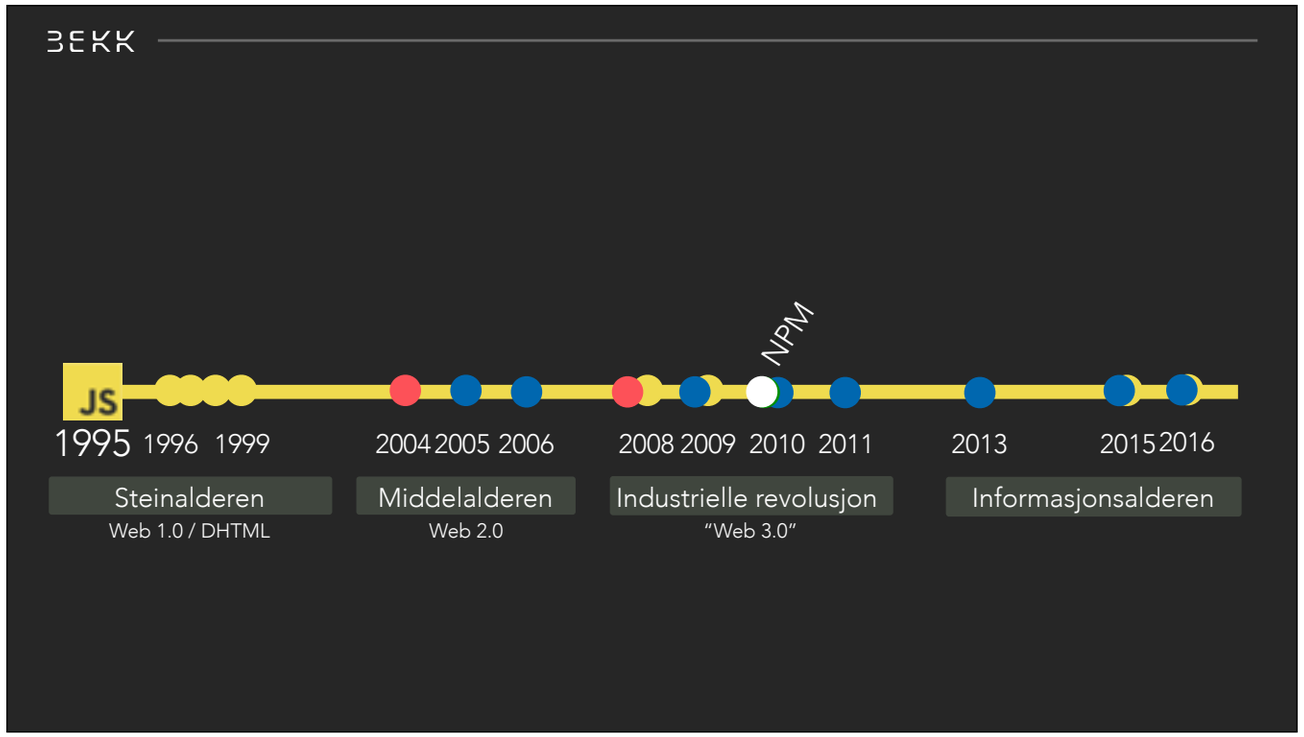
Og det gikk egentlig et års tid før jeg endelig innså at de kanskje var inne på noe. Ikke med JSX. Det er fremdeles unødvendig. Men å flytte den deklerative representasjonen av grensesnittet inn i JavaScript.



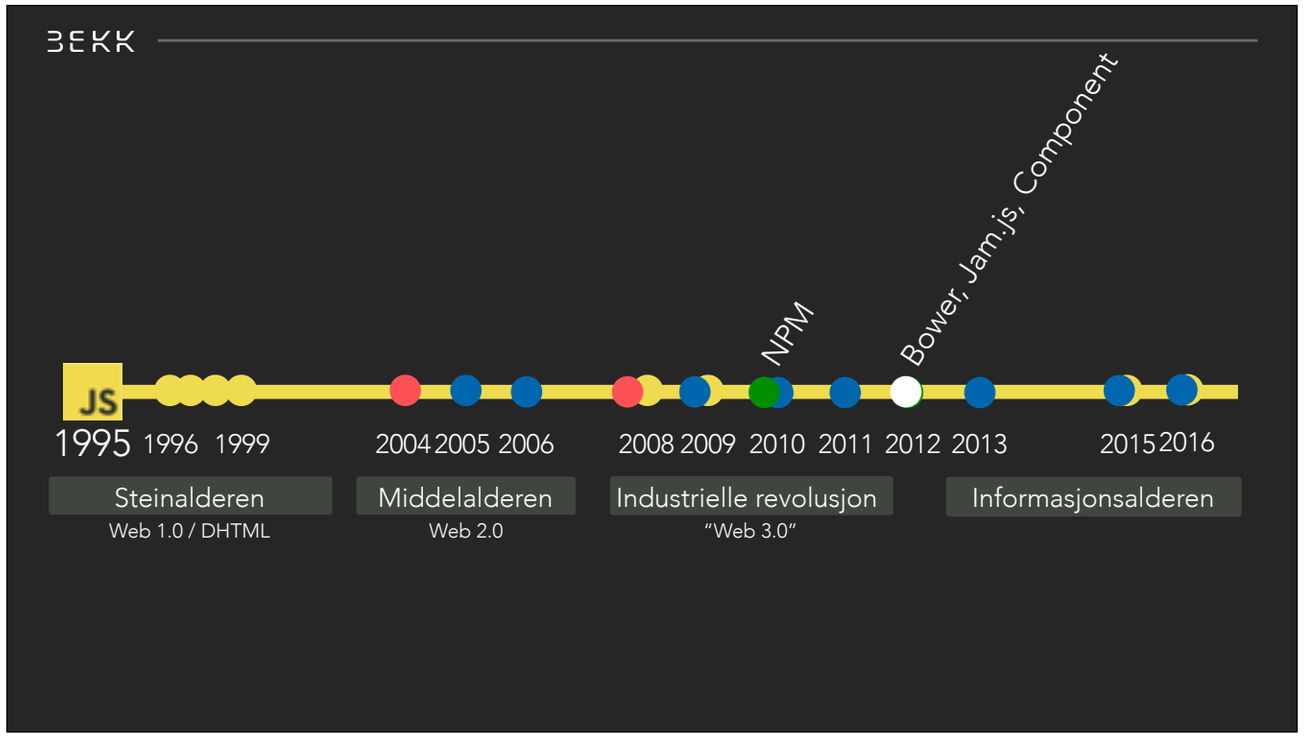
Andre som og ser at de var inne på noe med arkitekturen sin var resten av bransjen. Ember 2 kom i 2015 med en revidert render-modell kalt Glimmer. Ikke en virtuell DOM på samme måte som React, men med en inkrementell DOM og noen fancy algoritmer som jeg ikke skjønner. Angular 2 har ikke Virtuell DOM de heller, men kjører samme arkitekturen som React med envegsflyt og komponenter, fremfor tovegsflyt og direktiver fra Angular 1.

Dette er på mange måter andre iterasjon og andre generasjon av de store rammeverkene. Det er og mange andre konsepter som foregår rundt disse, med små rammeverk som tester ut og prøver. F.eks for tilstandshåndtering etc. Dette går og tusler til det til slutt blir mainstream.

Jeg har kalt denne perioden for informasjonsalderen. Ikke nødvendigvis for at vi er så informerte og kan mye, men at vi har så mye informasjon å gå på. Mange forskjellige retninger som blir utforsket.



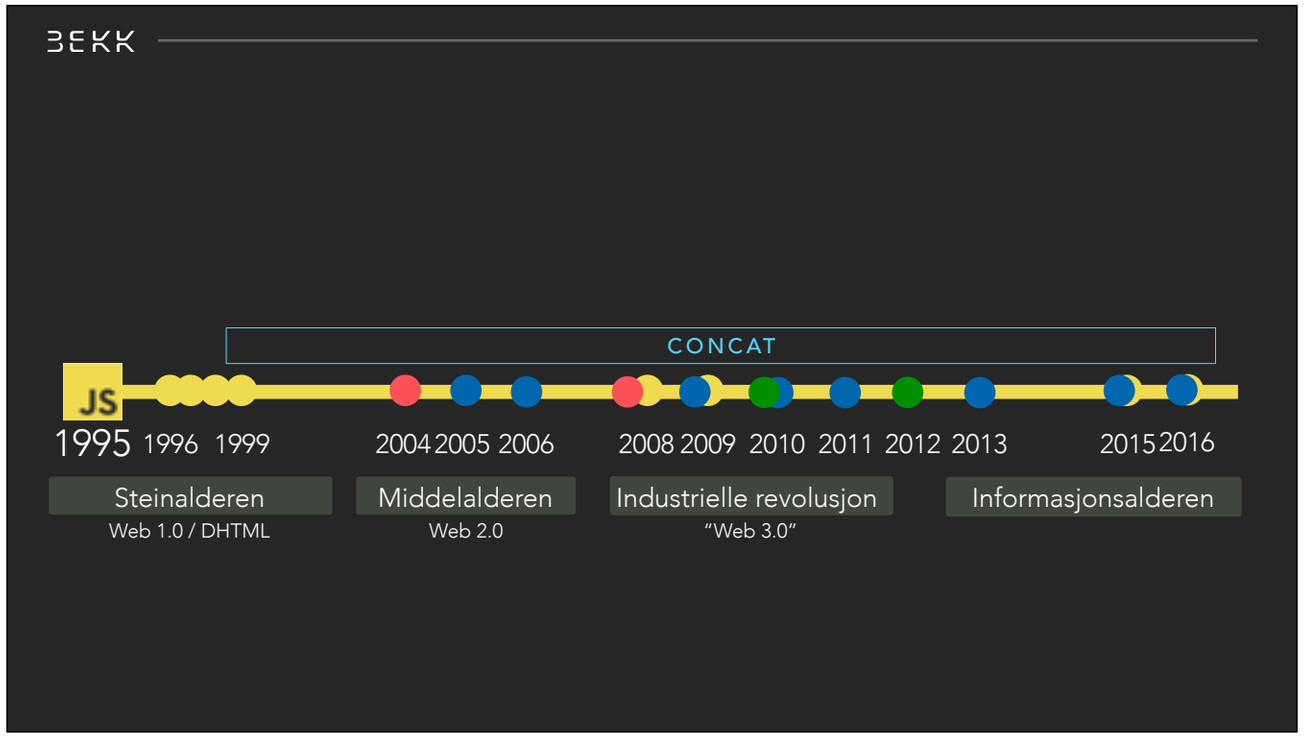
Inne på verktøy siden er det noen viktige hendelser som jeg syns er verdt å nevne. Med Node.js så fikk vi en package manager for JavaScript som kunne distribuere, ikke bare Node-pakker men og klientkode.



Men det var ikke helt tydelig i bransjen at NPM egnet seg til klientkode i starten. Så det var en del eksperimenter som foregikk for å prøve å løse det på en annen måte. Bower, Jam.js og Component er de mest nevneverdige - hvorav Bower var den største utfordreren.

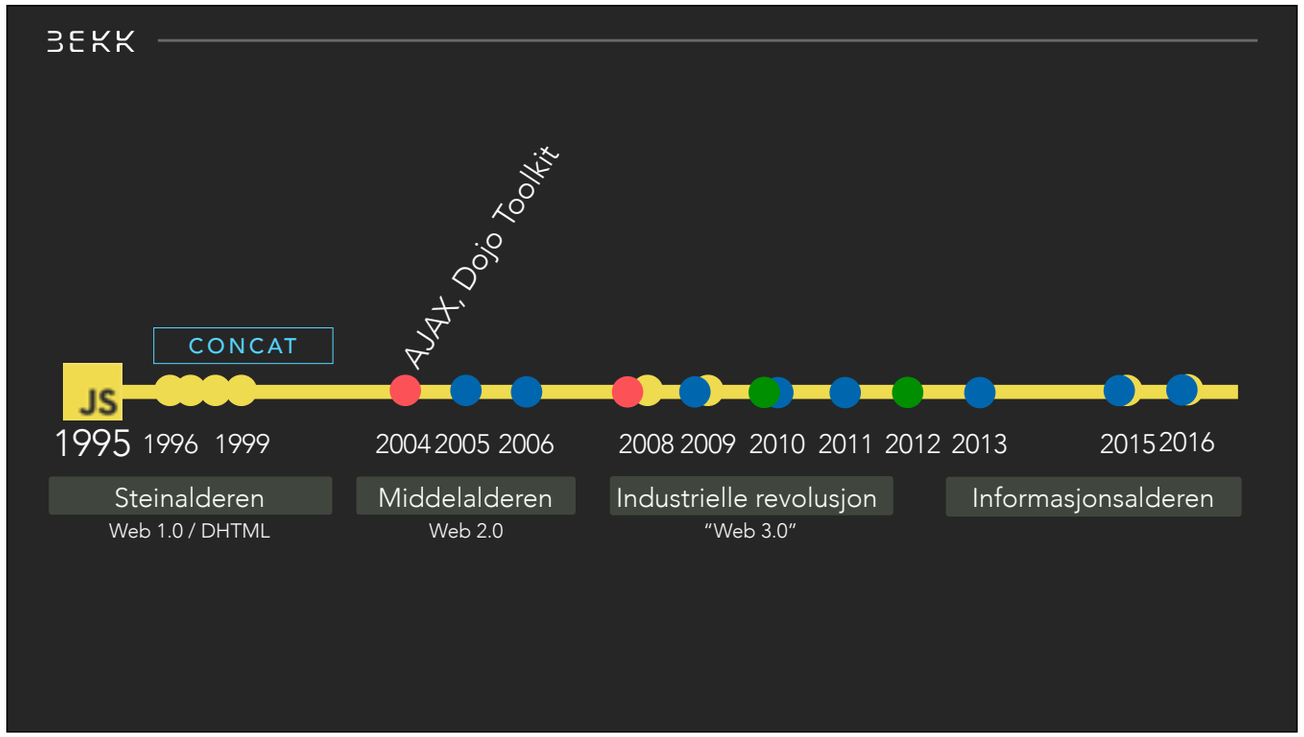
Men etterhvert ble det tydelig at NPM utfylte alle kravene vi hadde for pakke-register for både Node.js og klientkode.

Nå til dags er det ingen grunn til å bruke noe annet enn NPM i de tilfellene man trenger eksterne avhengigheter.

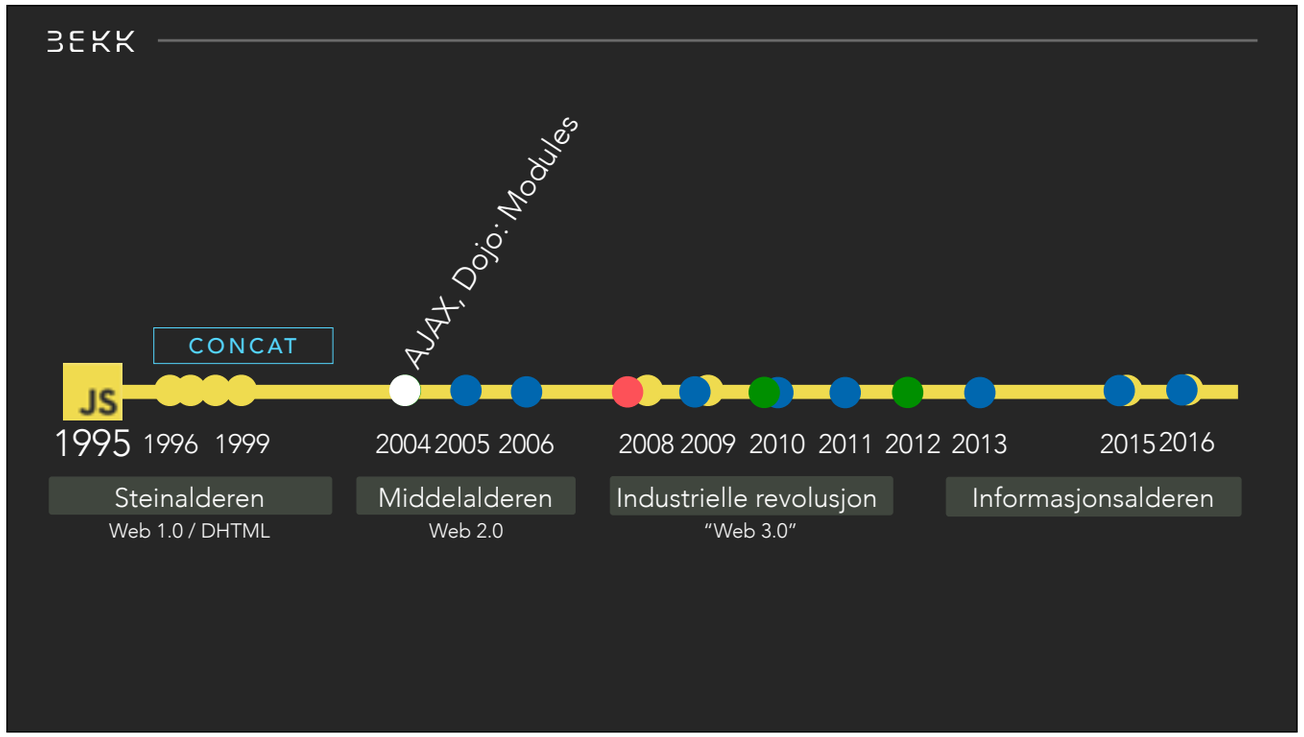


I større prosjekter, for både eksterne og interne avhengigheter, er man avhengig av å kunne bundle de. Å ha mange script tags skalerer opp til et punkt og fungerte fint så lenge man har et fåtall filer som man vil inkludere, men dersom man lager apps måtte man til med optimaliseringer på antall HTTP requests. Det ble og blir mye gjort enda med å bare smelte sammen alle filer.

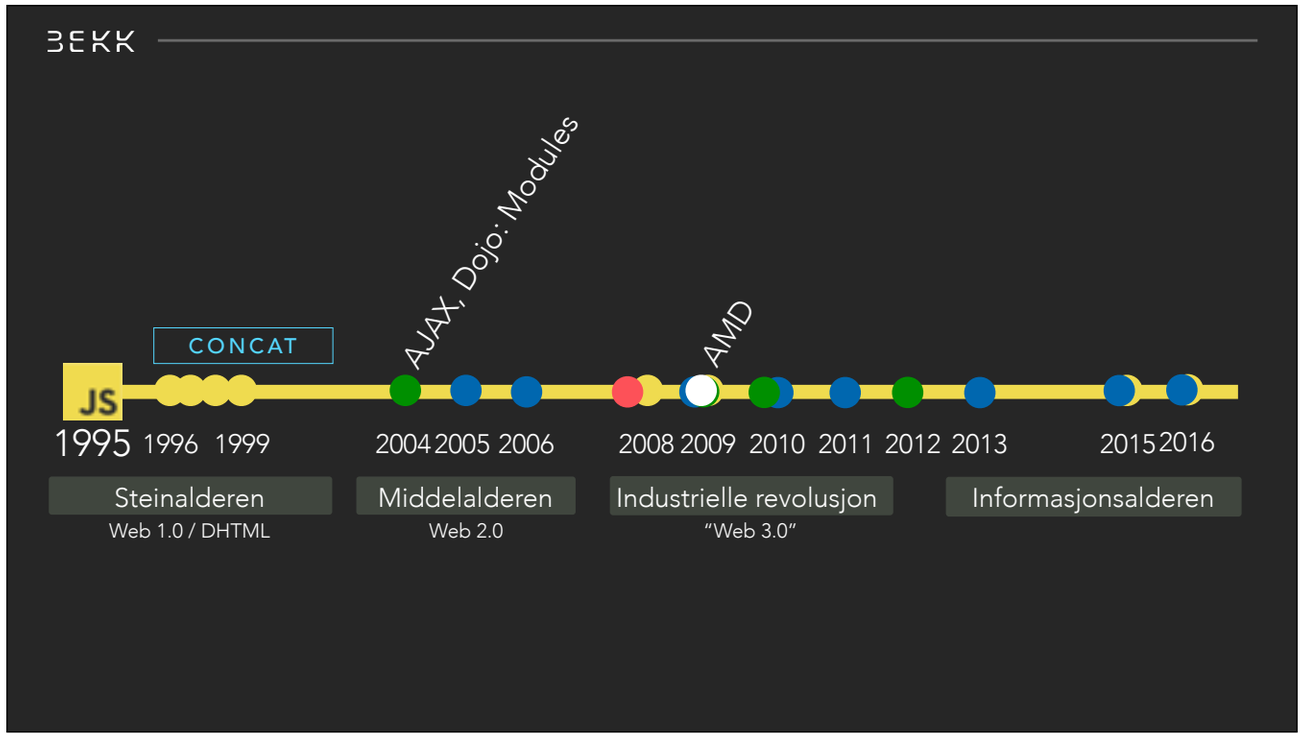
Det og selv om man har hatt skikkelig modulhåndtering ganske lenge.



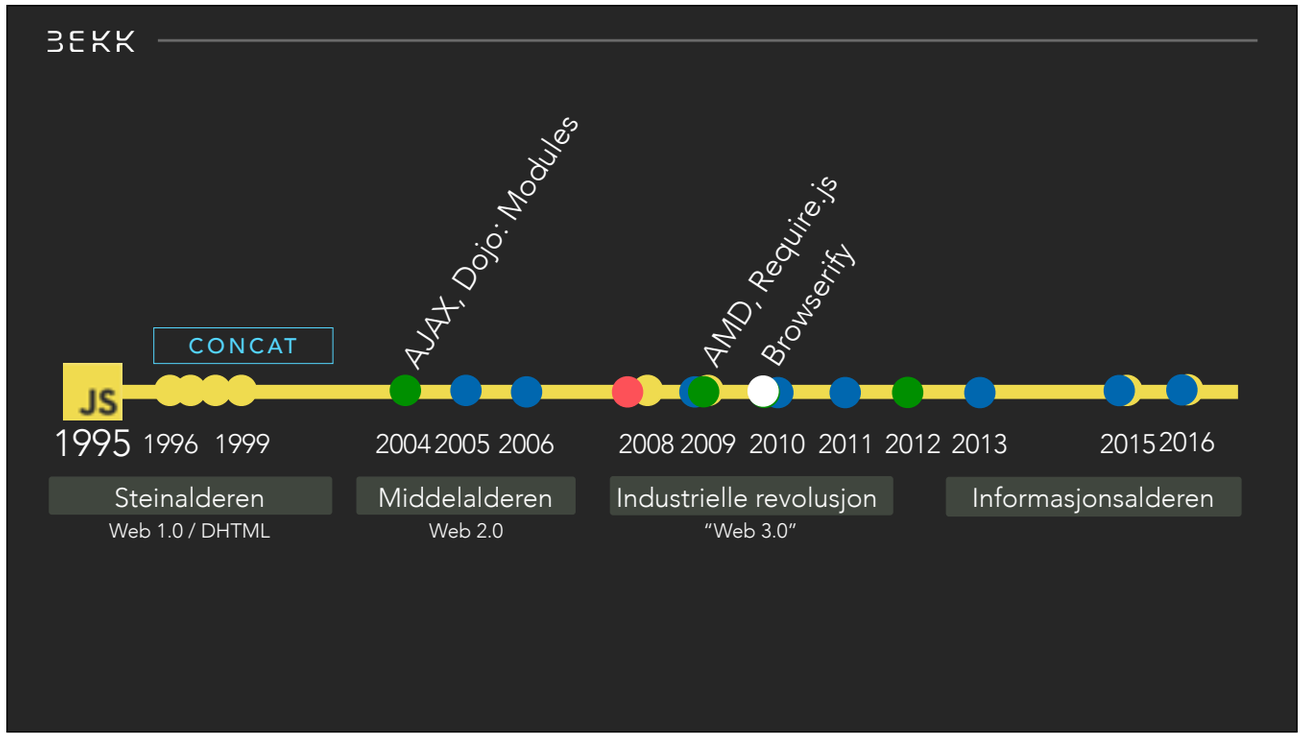
Alt da AJAX ble utbredt og Dojo Toolkit kom, fantest det moduler.



Der kunne man resolve javascript filer i runtime via AJAX. Det kunne man gjøre, selv om det ikke ble så veldig utbredt utenfor Dojo helt



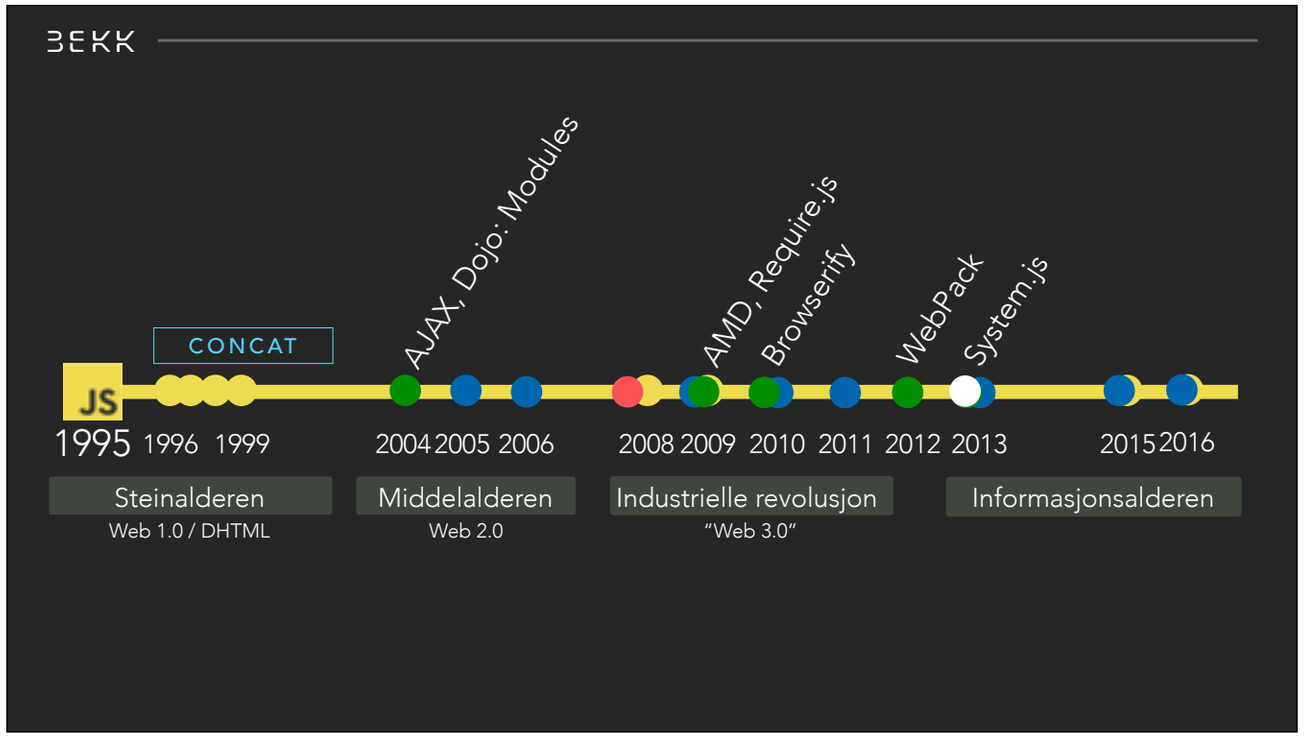
frem til da Asynchronous module definition (AMD) ble en ting i 2009. Da støttet Dojo AMD. En person som jobbet på det så nytten av å gjøre det som en ekstern pakke og



lagde da Require.js. Dette ble straks mer populært. Og her kunne man bygge opp et skikkelig avhengighets-tre og definere hvilke andre moduler en gitt modul forutsetter. Dette gjorde og muligheten for testing enklere. Da man kunne separere ut funksjoner til egne test-bare moduler.

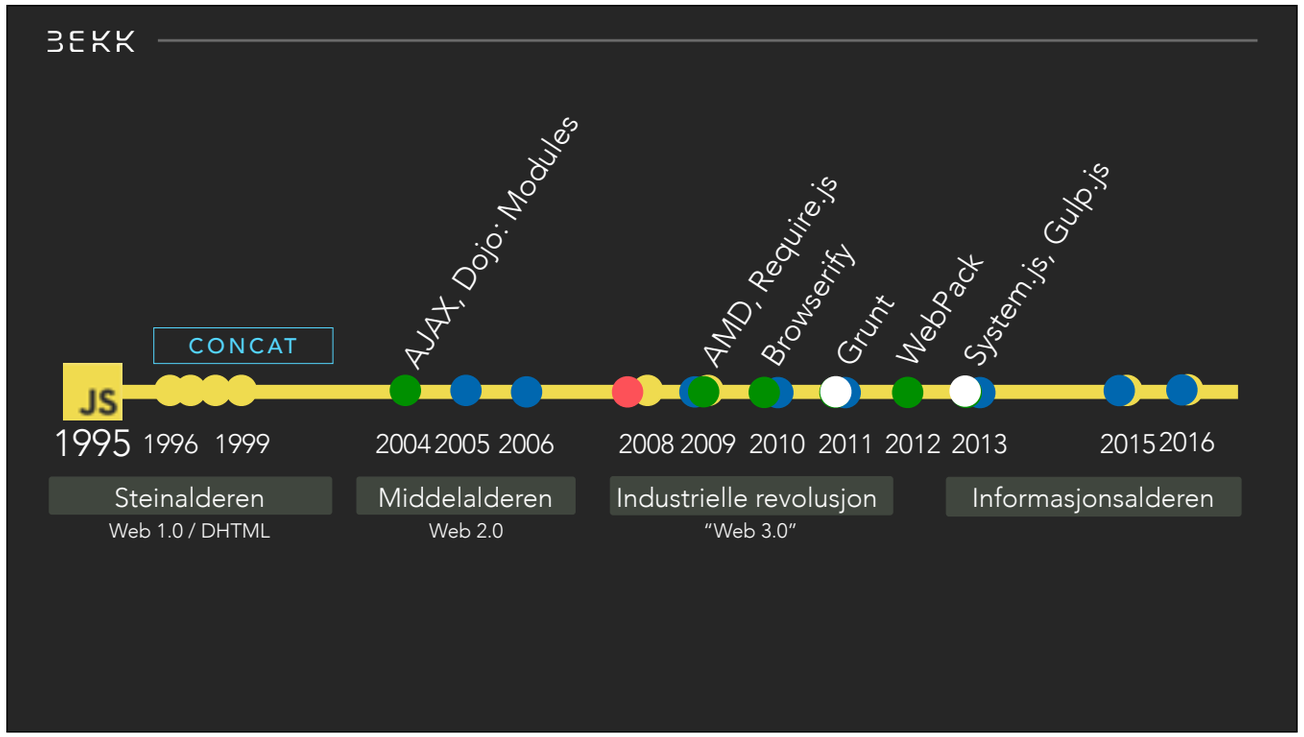
Det var en annen viktig spiller i rundt denne tiden. Det var Node.js og dens måte å håndtere avhengigheter på via en variant av CommonJS-iniativet - som var en måte å referere til avhengigheter på server-side javascript. Browserify ble laget som en enkelt måte å autoamtisk bygge et avhengighetstre og bundle alle filer til å kunne bruke i nettleseren.

Disse to levde side om side ganske lenge, men



Det skjedde mange andre initiativ etterhvert som teknologien ble mer og mer utbredt. Mest aktuelle er WebPack og System.js. Nå til dags har AMD dødd mer og mer ut, men Browserify, WebPack og System.js er i utvikling og er utbredt.

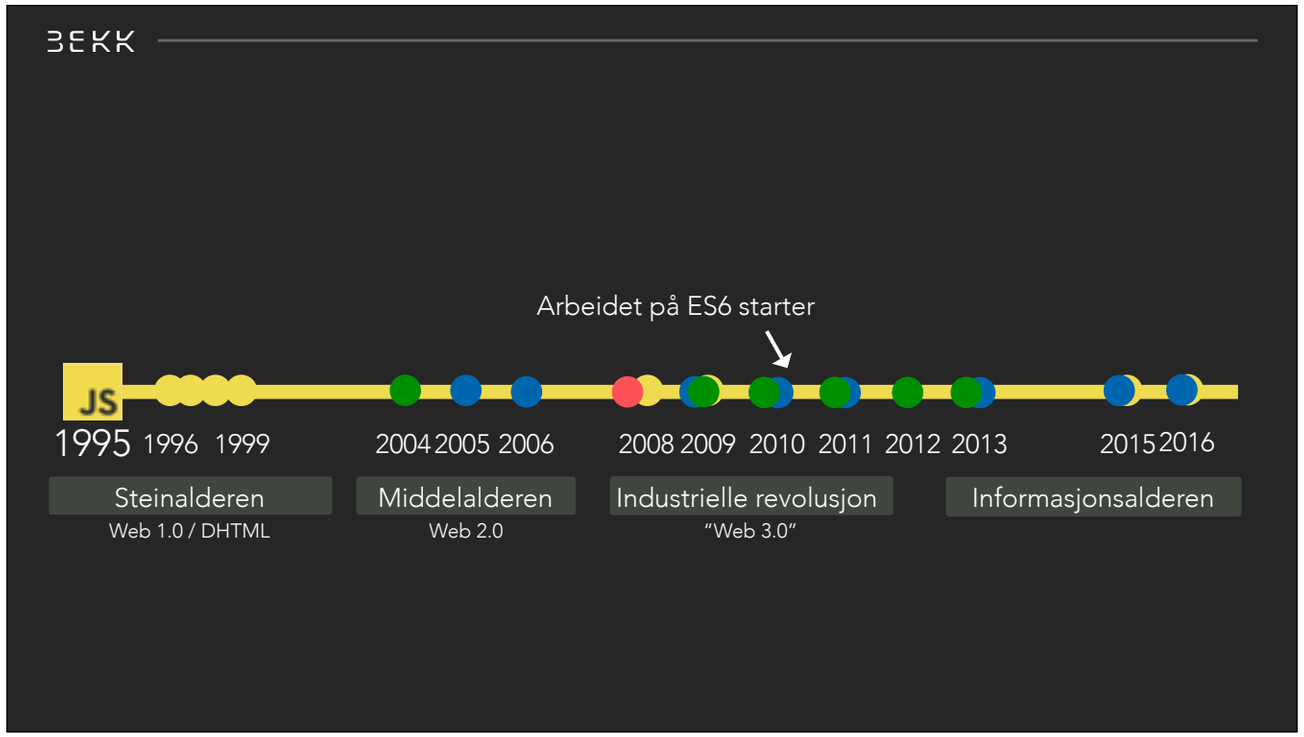
Det skjer og mange andre små eksperimenter rundt bygging og er en del av økosystemet som har vært mye i utvikling i løpet av “informasjonsalderen”.



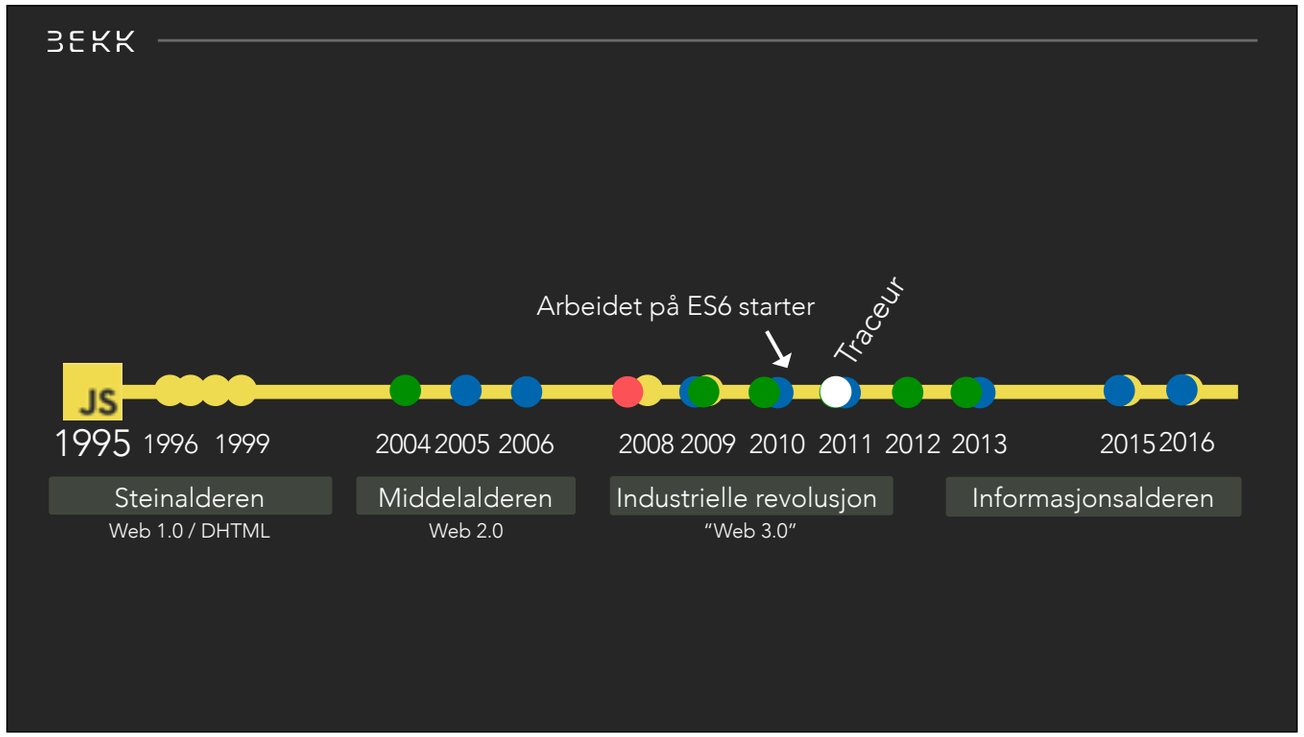
En annen del som har vært mye omtalt er task-runners. Tidlig ute var Grunt som viste hva man kunne gjøre. Det utløste en fanfare av forskjellige andre måter å gjøre det på og i 2013 var det en som utmerket seg ved å spille mer på Kode enn på konfigurasjon som var Grunts greie. Gulp.js viste og til en del benchmarking om at de var raskere ettersom de baserer seg på streams, fremfor mye IO-aktivitet.

Det er mange andre som kan nevnes her som har mange andre gode idéer, men disse er de som er mest prominent. Grunt har, etter min mening dødd mer og mer ut, og på mange måter har man sett at taskrunners ofte er unødvendig da man har andre verktøy som å bruke type browserify og f.eks Webpack direkte.

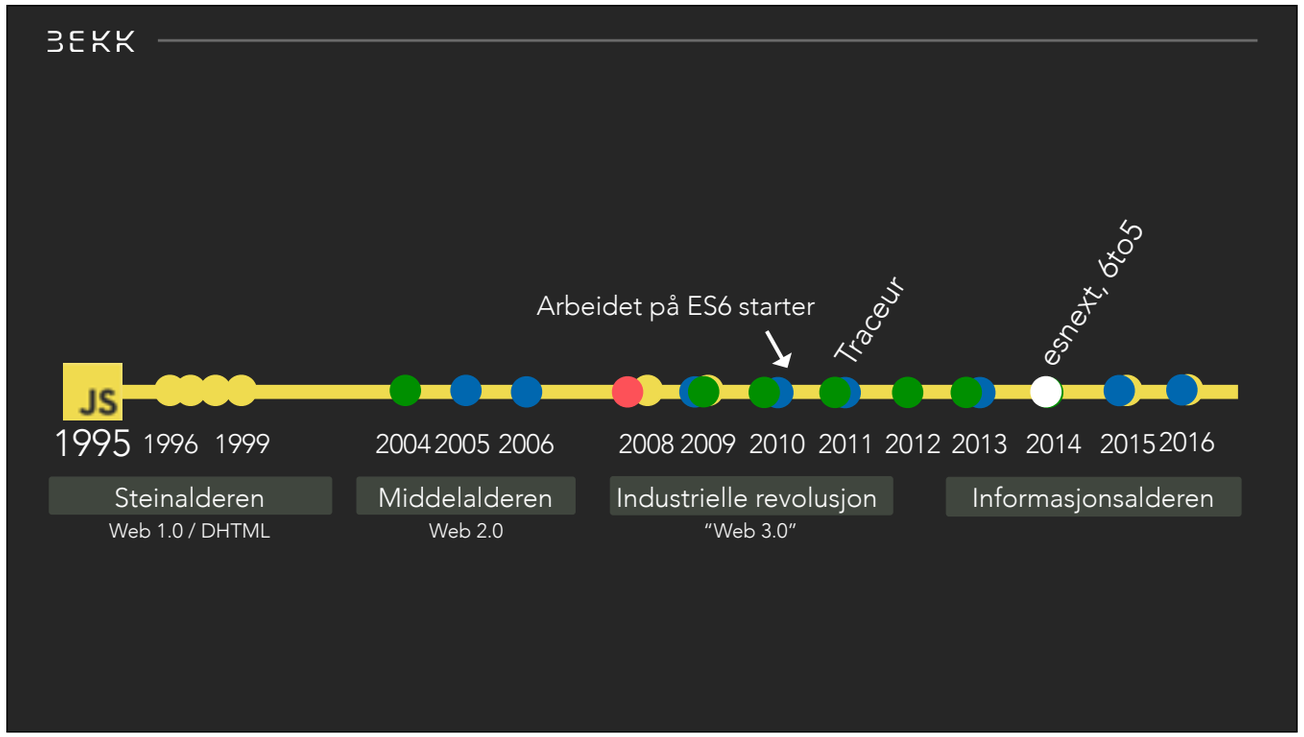
Men disse implementasjonene hadde ikke vært mulig uten forskjellige eksperimenter rundt hvordan man kan gjøre task-runners.



Noe før 2011 startet arbeidet på ES6 (nå kjent som ECMAScript 2015), men det skulle bli lenge til det kom til nettleserene implementert.

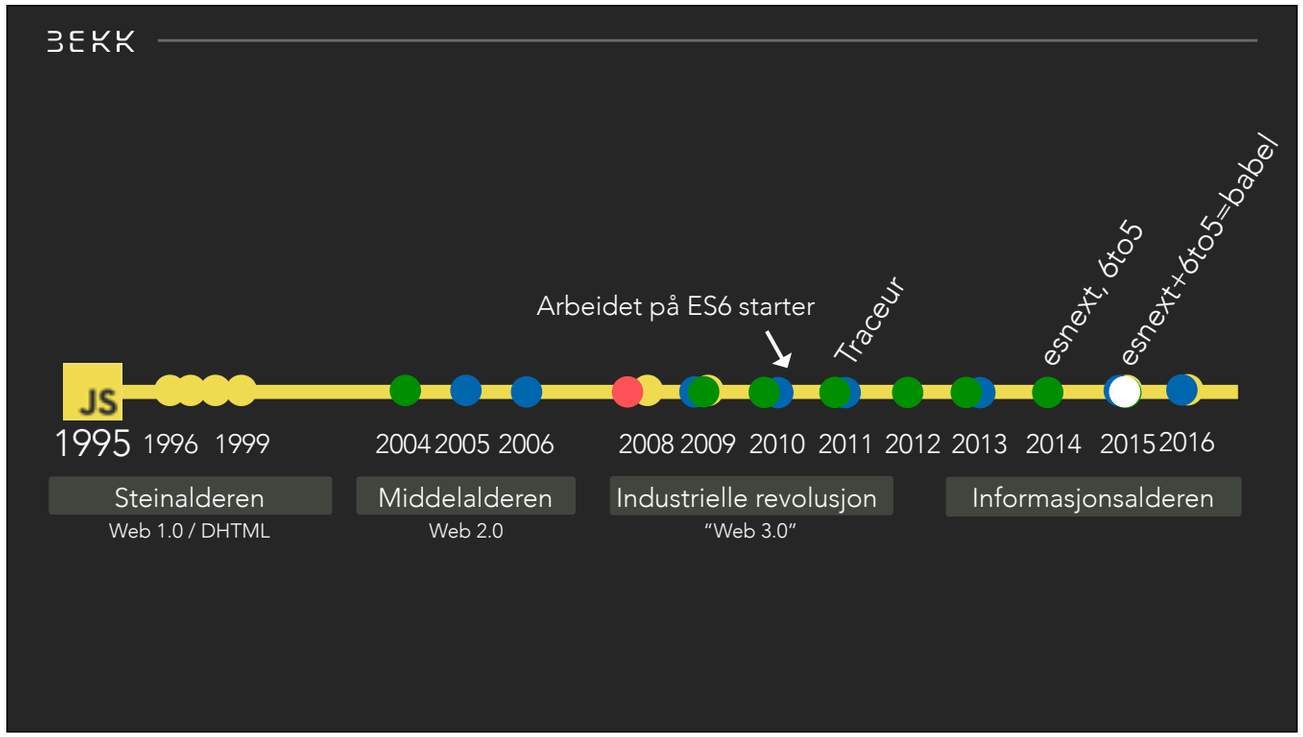


Mange utviklere var utolmodig og ville gjerne ta i bruk de nye språk-featurene. Kanskje for i entusiasme pga at språket ikke hadde vært i særlig stor aktivitet frem til da, men nå endelig skjedde det noe. Google holdt på med en transpileringsmotor traceur som oversatt den nye koden ned til kjørbar kode i nettleseren.



Men som første versjoner av ting var den langt i fra perfekt. Det gikk veldig sakte for mange, og det var et massivt beist av en kodebase med en runtime med alle slags mulig polyfills, osv. I tillegg var det mer bygd på Java-idiomatisk kodestil fremfor JavaScript-vennlig API-er.

Etterhvert begynte det å komme mange flere verktøy for å transpillere fra de nye språkutvidelsene som kom. Størst var kanskje es.next og 6to5. 6to5, var et initativ som først kun tok for seg språklig syntaktisk sukker og ikke de store nye implementasjonene.



Men i januar i fjor gikk esnex og 6to5 sammen for å lage en plattform for Javascript-til-javascript transpilleringsmotor med navn Babel. Ikke bare for å gå fra ES2015 til ES5, men og alle nye versjoner fremover. Og ikke bare ting som er tatt inn i standarden, men og andre ting basert på plugins til Babel.

Det kan virke som det har vært en massiv utviklingshastighet i frontend. Og det som er gått igjennom nå er bare de grove trekkene for å se hvordan utviklingen av økosystemet har kommet dit det er i dag.

En ting man hører ofte er at for Javascript så kommer det et “nytt rammeverk hver uke”. Og det har skjedd mye. Det er mye eksperimentering. Det er en plattform som plutselig så en stor opptur. Mer og mer kode kom over fra serversiden. Man kunne plutselig gjøre helt nye ting som AJAX og V8 gjorde at store ting kunne skje og mye kode kunne kjøre raskt.

Det blandet med at mange kommer fra andre plattformer og prøver å løse sine problemer med sin bagasje har gjort at mye har skjedd. Men er det en negativ ting? Trenger man å følge med på alle detaljene som skjer, eller kan man slappe av, bruke etablerte måter å gjøre det på og vente til endringer skjer i bransjen.

For som vi har snakket litt om flere ganger, fungerer mange av de bibliotekeker og rammeverkene som kommer ut eksperimentelle, utforskende tiltak som ser hvilken retning økosystemet skal ta. På mange måter som en flaggermus bruker echo lokasjon for å finne frem til retningen å fly.

Ta for eksempel oversikten til forskjellige rammeverker og biblioteker som har kommet. Denne oversikten inneholder ikke veldig mange og i realiteten er det jo en drøss av andre ting som kom ut mellom her. Og det var mange forskjellige utforskende rammeverker som kom ut. For mange til å nevne. Men for den jevne konsument, var det viktig å vite hva det var? Nei. Men resultatet fra de og og eksperimentene de kjører er viktig. Gode idéer blir tatt vare på og iterert på.

Akkurat samme strategi vi kjører når vi bygger software.
Definer - Realiserer - Valider - Start på ny med forhøyet kunnskap.

Men dette skjer på et kollektivt plan. Det er mye Open Source arbeid, så det er ikke nødvendigvis de enkelte implementasjonene eller sentrale personer som tester ut, men det skjer i bransjen og gevinsten blir delt med alle.

Gode idéer blir tatt vare på og brukt til å forbedre måten vi skriver software på.

Photo by Tony Webster, flickr@tonywebster.com

For det er lett å tenke at alt var bedre før. Vi klarte oss uten noen rammeverk/bibliotek. Klarte oss uten noe arkitektur i frontend. Eller, jQuery plugins fungerte helt fint. Og i noen tilfeller er det sant. Men man må og huske på at kravene endrer seg. Sluttbrukere får høyere krav til funksjonalitet, stabilitet og bedre løsninger.

Det som er med problemer er at vi glemmer ofte smerten vi har hatt med andre teknologier før, og sitter bare igjen med de smertene vi har nå. Akkurat som når vi er syk osv osv.

Det man ofte hører er destruktive unyanserte anit-argumenter med større grad av hytting enn objektivitet. Som spøkete sleivspark om hipster-teknologi. Kritisk tenking er viktig, men vær kritisk av de rette grunnen.

Litt av rykte-problemene JavaScript har hatt, virker for meg som et resultat av at det tradisjonelt har vært en sekundær prioritet. JavaScript har tradisjonelt sett vært et suppleringsmål. Men etterhvert som klientene har blitt tykkere, og mer kode blir flyttet over, må flere forholde seg i større grad til JavaScript. Da er det en vanlig fremgangsmåte å angripe det som man har angrepet sin primær-plattform som fører til at forventningene blir feil og språket og plattformen kan virke irrasjonell.

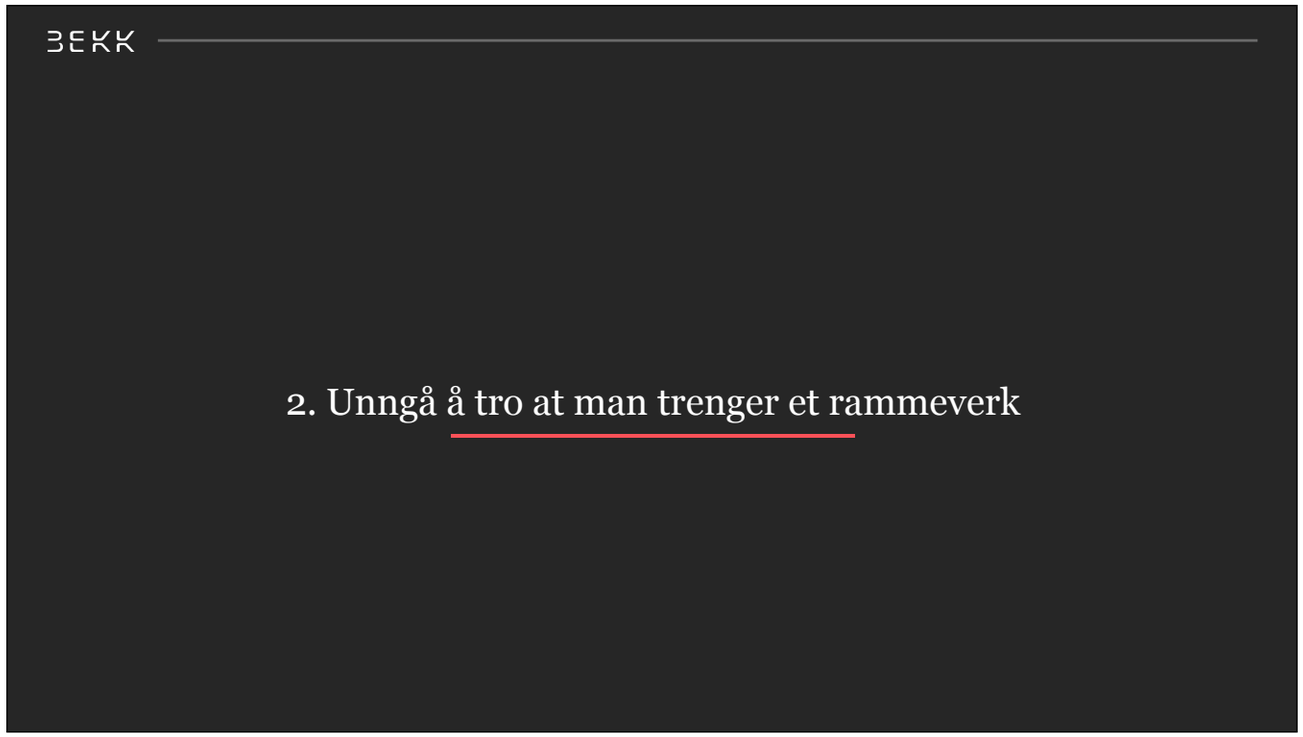
Men nå tar jeg på meg Conchita-skjegget mitt, bli en slags ekstra ung og kjekk Nicholai Cleve Broch,

og se litt på hva et moderne frontend prosjekt trenger nå. Det er lett å bare gå bananas og tro at man trenger alt som er av features og shiny toys. Men her kommer en litt mer nøktern tilnærming til det.

Først er det viktig å tenke på at man faktisk ikke trenger en stor klientside for alle prosjekter. Noen ganger kan man ta den helt tilbake til DHTML-dagene og bare endre litt på DOM-noder med vanilla JS. Da trenger man ingen infrastruktur, ingen rammer. Det er lett å tenkte at alt må ha klientkode, men slik er det altså ikke.

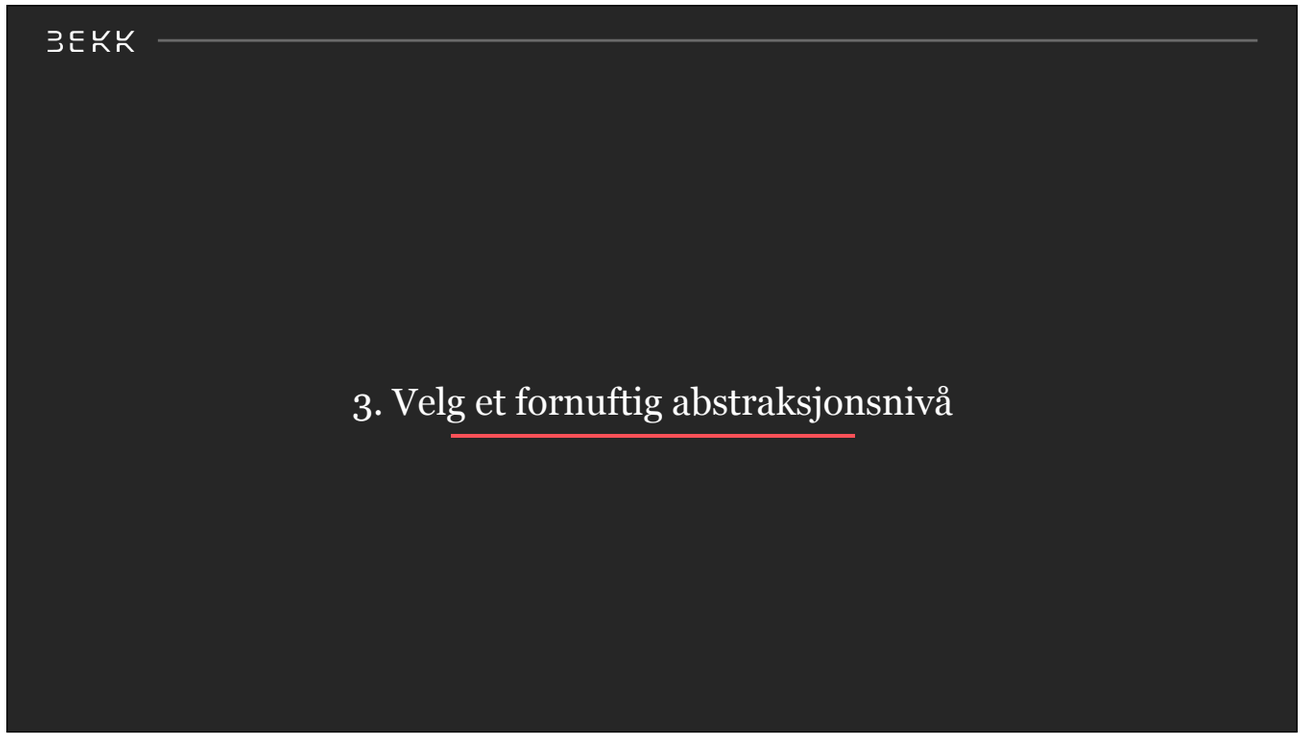
En viktig regel, syns jeg, er for all del unngå å kopiere blindt andres stack. Å starte med en boilerplate er å skyte seg selv i foten. Man starter med en massiv kodebase som gjør ting en egentlig ikke vet hva er, og egentlig ikke trenger. Starterpacks er OK å se på for å se hvordan ting kan gjøres, men i det man starter et nytt prosjekt med starterpacks er man på tynn is fra første stund.

Det er mye enklere å legge til kode enn å fjerne det. Starter man med mye er det et dårlig utgangspunkt.

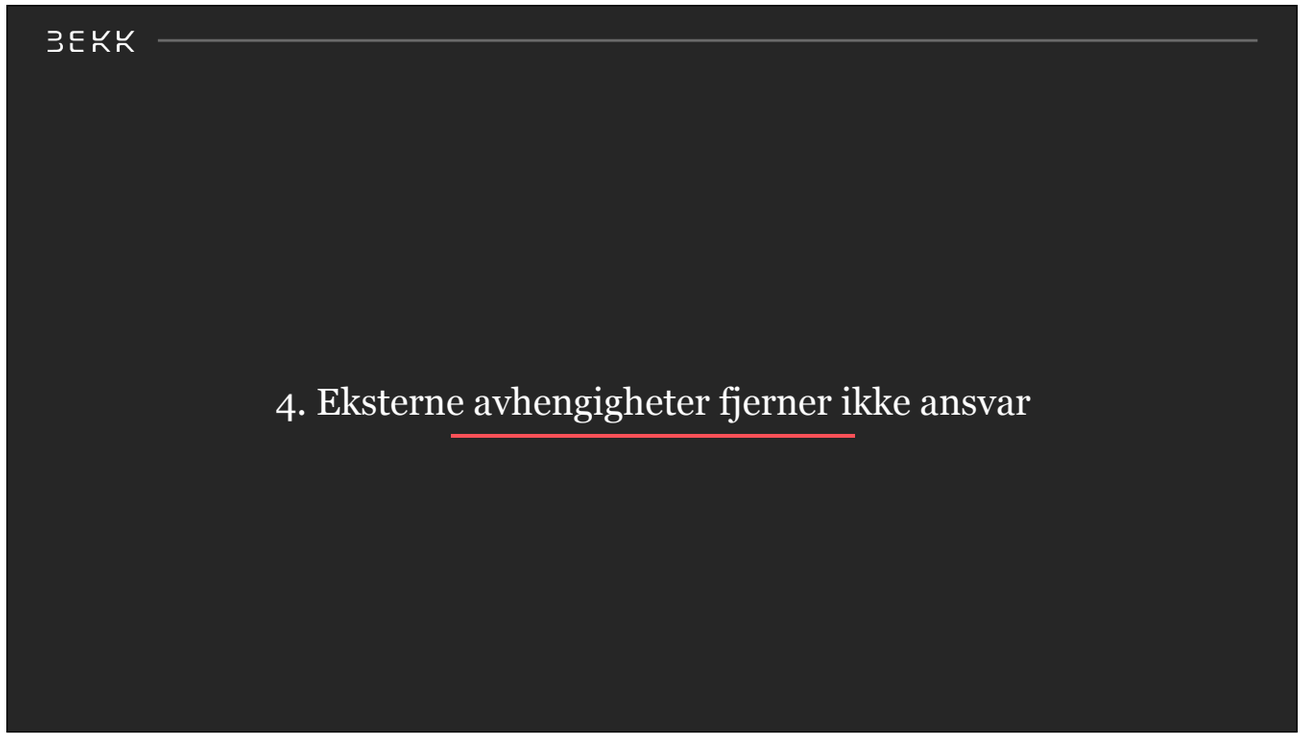


Dette henger også sammen med at man ofte tror at man må til med et rammeverk – eller forsåvidt et bibliotek. Noen ganger kan man fint klare seg uten et rammeverk.

Det er vanskelig å tenke enkelt. Senest for noen uker siden, skulle noen starte på et dashboard-prosjekt og spurte meg om hva de skulle velge av rammeverk. Med en gang begynte jeg å tenke på hva som passet inn med det problemet de hadde. Før det plutselig slo meg: Hvorfor i alle dager trengs det et rammeverk i det hele tatt? Her er det jo egentlig bare en del av en side som oppdaterer seg på en knapp. Dette har vi gjort siden “middelalderen” og Web 2.0. Løsningen blir **mye** simplere og man mister ingenting.



Dette handler og om å velge et smart abstraksjonsnivå. Før et prosjekt, får jeg noen ganger spørsmålet “hvilket rammeverk skal vi velge”. Men som mange har sagt bedre før meg: Hvorfor skal man sitte på et tidspunkt der man vet minst å ta store valg som man må leve med lenge. Det er alltid enklere å gå fra en lav abstraksjon til en høyere abstraksjon.



En av de tingene jeg liker godt med JavaScript er språk-fleksibilitet og kultur rundt Open Source og valg. Jeg tror som sagt det er et resultat av at språket stod litt stille og at det da kom en kultur der mye skulle skje i brukerland. Men det er ikke nødvendigvis negativt. Men det kan bli dratt til det ekstreme. F.eks med mange små moduler. Noen ganger er det bra, andre ganger er det ikke.

Det som er viktig å huske på er at selv om man delegerer bort handlingen om å kode noe til en ekstern avhengighet, betyr ikke det at man har delegert bort ansvaret. I det du drar inn en avhengighet er det ditt ansvar. Man må alltid tenke over prisen av noe og ta et konkret valg på om prisen er verdt gevinsten.

I informasjonsalderen har det blitt mer og mer vanlig med veldig shiny toys og fine demoer. Igjen kan det være fornuftig å tenke over at alt du tar inn i din kodebase er noe du må vedlikeholde.

Det er en kostnad med å dra inn nye ting, utover vanlig kostnad for eksterne avhengigheter. Noen ganger er det fornuftig å la ting stabiliseres før man tar det i bruk. Andre ganger passer det fint å være en del av den utforskende gruppen. Men ta bevisste valg, ikke tilfeldige.

Man må alltid tenke “Hvorfor” skal dette gjøres fremfor “jaja, hvorfor ikke. Kan sikkert være kjekt”. Det kan godt være at det rette valget er å introdusere et nytt verktøy eller rammeverk, men det må ligge en tanke bak.

Du må ha behov som dikterer valg for å dra inn noe, ikke tilpasse behovene til et ubevisst valg.

Siste punkt av min strenge, kjipe liste er den aller viktigste. Nettleseren og JavaScript er en plattform på lik linje som serversiden. Vi skal skrive skikkelig kode i frontend. Og JavaScript har sine styrker som andre språk.

Det er derfor viktig å ta den plattformen for det den er. Lær deg faktisk hvordan den fungerer. Ikke anta at det fungerer på den måten som du koder Java, C# eller slik du koder Ruby. Da vil det dukke opp mange rariteter som du ikke kan forklare. Les en bok om hvordan språket fungerer. Dagene med DHTML er forbi. Skal vi lage applikasjoner som fungerer og varer i nettleseren er vi avhengig av å vite hva vi gjør.

Photo by cjuneau (https://www.flickr.com/people/15558406@N03) from Ottawa, CANADA

Tror det får være nok av den restriktive kjipe delen. Det er enkelt å nå tenke at alt er “vær kjip”, bruk bare gamle ting, og tro at jeg er bitter.

Men faktumet er at ting begynner faktisk å bli ganske bra i frontend. Og nå skal vi se kjapt på hvordan vi kan sette opp en rett frem moderne infrastruktur for frontend som ikke har så mye fuzz rundt seg.

Vi har til nå sett på the “don’ts”, men nå skal vi se mer på the “dos”.

1. Bruk Node.js og NPM*

Hvorfor

Dersom du trenger eksterne avhengigheter og verktøy til å bygge ting, er Node.js det som burde brukes og NPM burde brukes til å dra inn avhengighetene – både for verktøyene og kode-avhengighetene.

Det er i Node nesten alle moderne verktøy er implementert. Dette for det fungerer overalt. Fra Windows, til Mac, til ARM, what ever. Node.js er kryss plattform. Det gjør og at det egner seg veldig godt til å scripte egen infrastruktur. Drit i make og bash, særlig om man skal kjøre kryssplattform.

BEKK

2. Bruk moduler*

Hvorfor?

```
{
  "name": "without-babel",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack app/main.js dist/bundle.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^2.1.0-beta.5"
  }
}
```

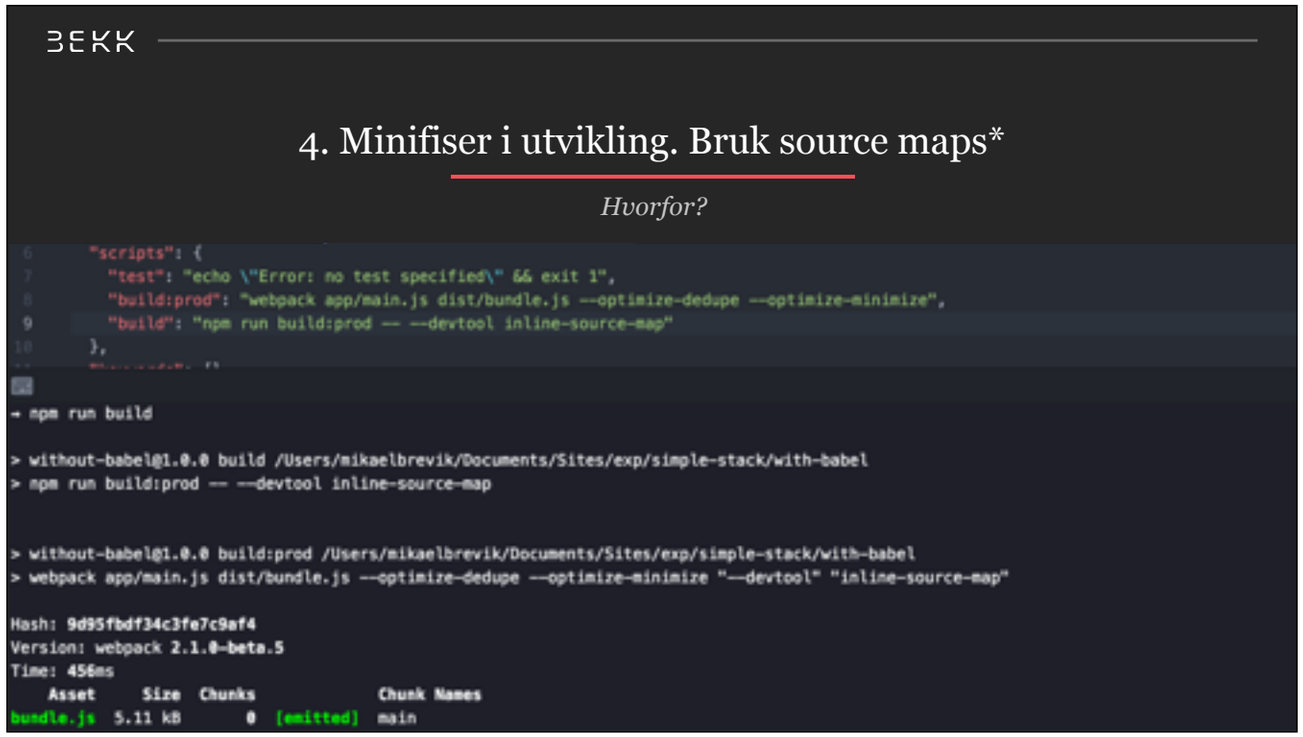
Gitt at man lager en applikasjon av en viss størrelse vil man dele den inn i moduler. Det er ikke alltid man trenger det, for man lager ikke en applikasjon, men om prosjektet beveger seg til en viss størrelse må moduler til. Det kan fungere med concatenering dersom det er få filer uten en tydelig inter-avhengighet, men det skalerer ikke.

For å kunne skrive testbar og gjenbrukbar kode er moduler vegen å gå. En enkel måte å få det opp på er å bruke noe som browserify eller WebPack. Dersom man kun skal bruke noe som CommonJS er det stort sett trivielt hva man velger. WebPack har innebygd mulighet for watching som gjør den kanskje enklere å bruke - dersom man forholder seg til det på en enkel måte.

Nye prosjekt nå blir som regel satt opp med transpillering fra ES2015 til kode som fungerer i alle nettlesere. Trenger man det egentlig? Nei. Gir det en veldig mye og gjør jobben mye enklere? Egentlig ikke.

Men det er litt mer fremtidsrettet og det er mange moderne rammeverker som nå baserer seg på den syntaksen, så det kan være at det gir mening å innføre det.

Det er noen semantiske forskjeller på nye module syntaksen til ES2015 som gjør at det kan være greit å holde seg utelukkende til de. Som f.eks at man eksporterer bindings fremfor values fra moduler og at det er utelukkende statisk analyserbart. Dette gjør f.eks at man kan utelukke ubrukt kode på en bedre måte.



Så langt det lar seg gjøres burde man minifisere koden i utviklingsmiljøet og. Det kan være det gjør at det tar lengre tid å bygge og at det er vanskeligere å lese kildekoden, men det gjør at man jobber med koden slik den kommer til å bli lagt ut på nettet. Så man slipper uheldige hendelser der noe ikke fungerer i prod på grunn av minifisering.

Sourcemaps er blitt såpass gode at de fungerer utmerket i utvikling, selv på minifisert kode med debugging i nettleseren osv.

BEKK

5. Importer avhengigheter direkte

Hvorfor?

```
1   import compose from 'lodash-es/flowRight';
2
3   let a = n => n * 3;
4   let b = n => n + 2;
5   let ab = compose(a, b);
6   console.log(ab(4));
7
```

```
bundle.js  7.74 kB       0  [emitted]  main
    + 42 hidden modules
```

Man kan foretrekke å ta inn større avhengigheter som gir større verdi når man først har tatt det inn, men man burde prøve å alltid være konkret i importeringen av koden. Dette gjør at man ikke drar inn all kode som man ikke trenger og at pakkene blir mindre. Slik blir også byggetiden kortere. Alltid prøv å referere til u-bygd kode og ikke til dist-filer. Slik får man mest mulig ut av algoritmer som fjerner duplikater og ubrukt kode.

Noen tilfeller kan det gi mening å skrive kodene som skal kjøre i headless browser osv. Men i de aller fleste tilfeller trengs ikke det. Fokuser heller på å skrive kode som kan testes som enheter som enkle funksjoner. Og dette kan testes utenfor nettleser. Det er nesten vilkårlig hvilken test-runner du bruker, men det kan være en fordel å velge noe som er enkelt å få til å fungere med transpilering dersom man har tatt i bruk det. Man vil referere direkte til modulene og ikke teste bundlet sluttprodukt.

Headless browsers som PhantomJS er tungvindt å bruke og er ofte sårbart. Fremfor å bruke headless browsers kan man bruke JavaScript-implementerte DOM-er som JSDom som begynner å bli ganske bra. Der kan man simulere eventer.

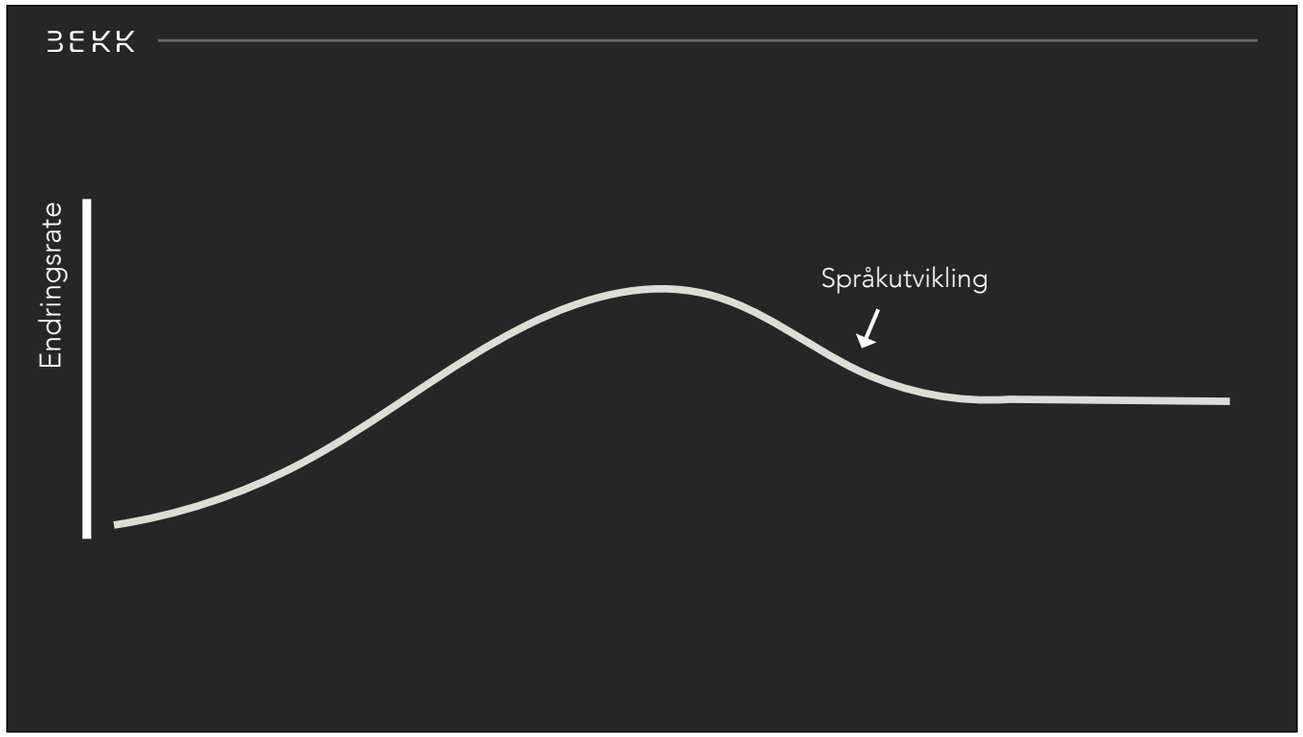
Jeg mener det gir mindre å mindre verdi å teste selve JavaScript koden i forskjellige nettlesere da de begynner å håndtere ting likt, men om det gir stor nytteverdi å teste i faktiske nettlesere på ditt prosjekt, så kan det være lurt å bruke en test-runner som kan kjøre både på serverside og i nettleser. F.eks tape.js eller Mocha.

7. Bruk sunn fornuft og god dømmekraft

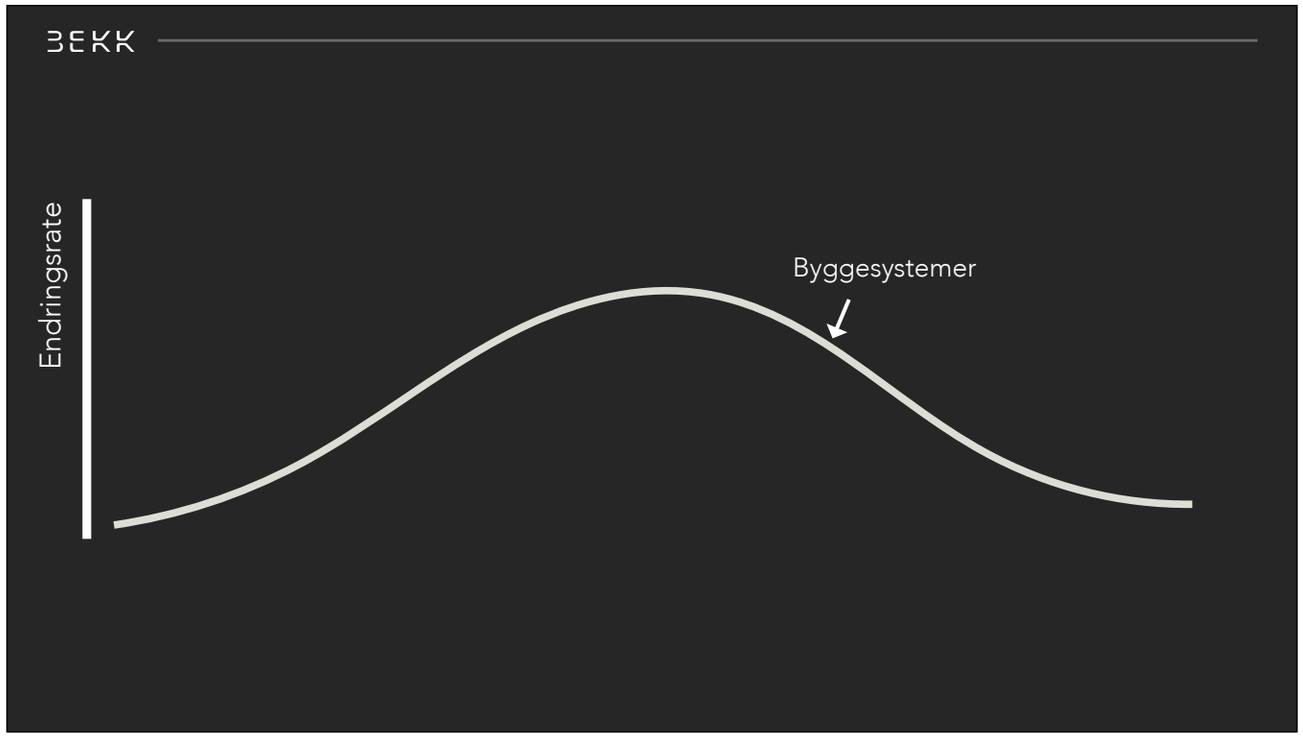
La behov diktere valg, ikke motsatt.

Viktigst av alt er å bruke sunn fornuft og se an behovene sine for hva man trenger og ikke.

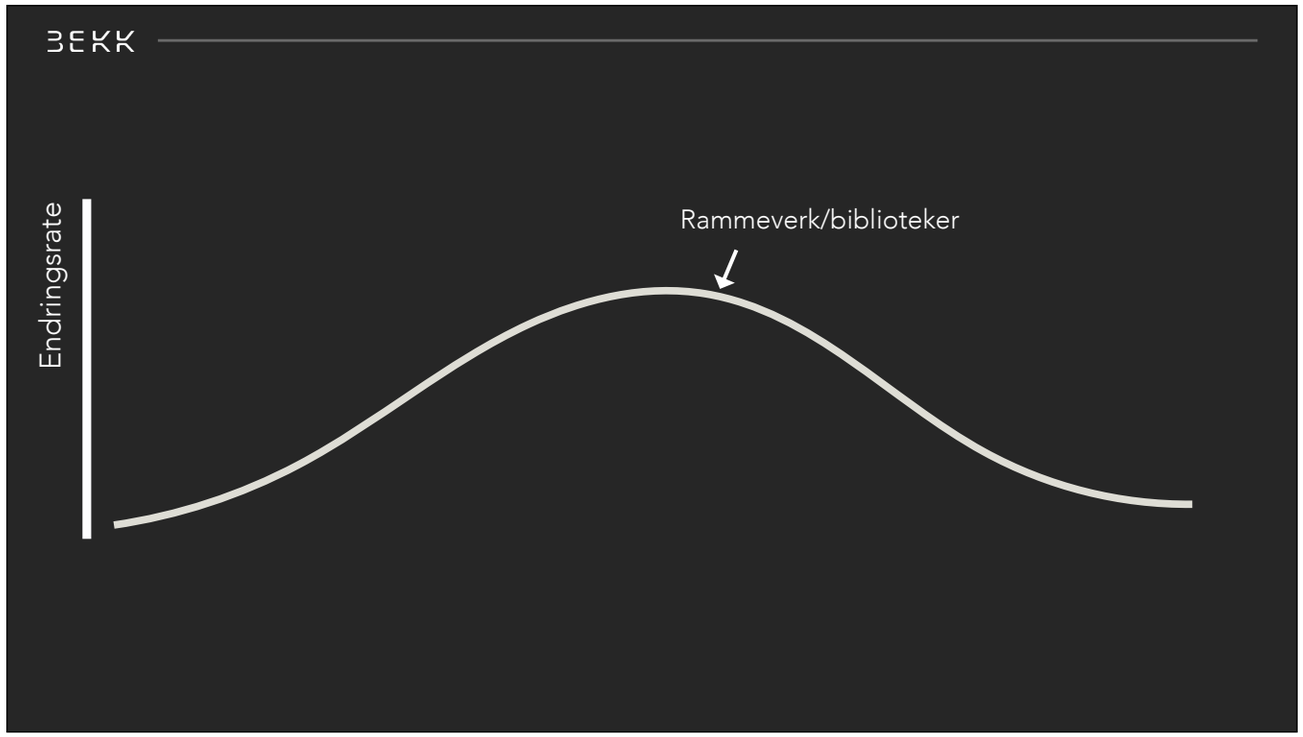
Det har skjedd en del endringer i frontend, og kanskje særlig de siste 3-4 årene. Men jeg tror helt ærlig vi er i ferd med å gå til en mer stable hverdag nå og vi ser litt en backlash av at alt skal være valg. Så det kommer til å komme mer og mer verktøy som bare gjør en opinionated job så man slipper å ta valgene selv.



ES2015 var en stor feature endring og nå kommer det til å bli mye mer jevn årlig flyt for utviklingen av språket. Det betyr at det ikke kommer til å bli så store batcher med features å lære seg fremover.

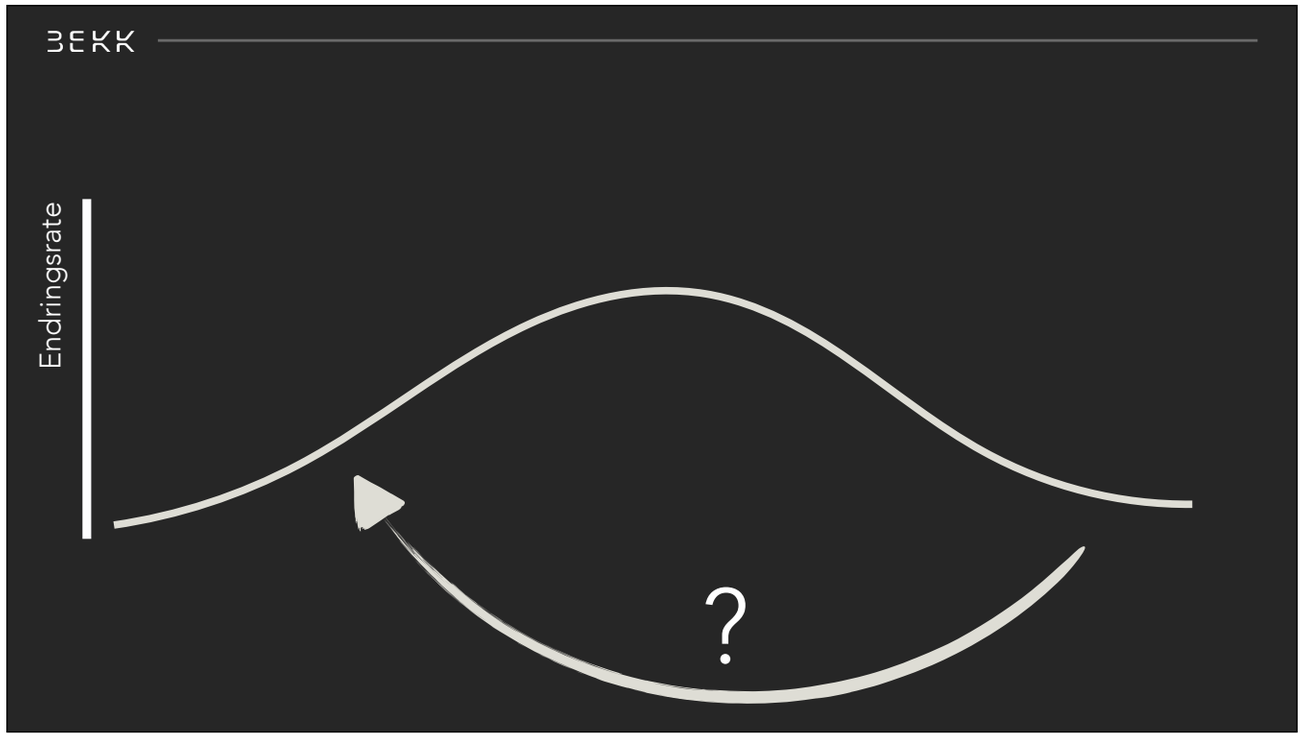


Jeg tror og at vi er over den største kneika av endringer og utforsking i byggesystemer og verktøy. Jeg tror det kommer til å flate ut nå fremover for en liten periode.



Med rammeverk og biblioteker er det kanskje fremdeles en høy grad av utvikling. Vi ser trenden med nye medium å bruke teknologien på. Vi ser at man bruker samme strategi på forskjellige plattformer. Outputer det ikke bare til DOM, men til WebGL, eller til native mobil kode. Så det kommer til å være en del utforsking enda fremover og forskjellige tilnærmingsmåter. Funksjonell stil er i frem-mars og jeg tror det kommer til å kanskje vare en stund og i noen tilfeller bli mer ekstrem / pure.

Men jeg tror kanskje vi er over det “værste”.



Men plutselig kommer det et innoverende prosjekt som revolusjonerer Web-en på ny som har skjedd flere ganger og vi går tilbake til større endringsrate igjen. Det er det som er herlig med Web-en og JavaScript trossalt. Mye skjer og vi forbedrer oss stadig og tørr å utforske for å finne bedre løsninger!

BEKK

TWITTER: @MIKAELBREVIK

For kommentarer eller spørsmål.

Gi tilbakemelding på: http://program.bekk.no

Takk for meg. Om det er noen spørsmål eller du er helt uenig i det jeg har sagt: Ta kontakt eller send inn tilbakemelding på program.bekk.no


BEKK