Univerza v Mariboru

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko

Spletne tehnologije

Pregled front-end spletnih rešitev

Niko Lukač

Uvod: 1. generacija web front-end-a

- Uporaba statičnih HTML/CSS datotek
- Veliko **dinamike na strežniški strani** s skritpnimi jeziki in CGI
- Prva generacija spletnih ogrodij:
  - **PHP**: CakePHP (2000)
  - **ASP.net** (2002)
  - **JSP** in Java servlets

Uvod: 1. generacija web front-end-a

- Uporaba statičnih HTML/CSS datotek
- Veliko **dinamike na strežniški strani** s skritpnimi jeziki in CGI
- Prva generacija spletnih ogrodij:
  - **PHP**: CakePHP (2000)
  - **ASP.net** (2002)
  - **JSP** in Java servlets

- Uporaba šablon (**templating**) za separacijo logike od videza (view)
  - Npr. v PHP:

```
$vars = array();

$vars['title'] = 'Products';
$vars['products'] = $productModal->getAllProducts();

$this->renderHtml('all-products.html', $vars);
```

```
<div class="products">
    {% for product in products %}
        <h2>{{ product.title }}</h2>
        <p>{{ product.description }}</p>
    {% else %}
        No products exist.
    {% endfor %}
</div>
```

- Odjemalčeva stran: **imperativni** način manipulacije z elementi (jQuery)

Uvod: 1. generacija web front-end-a

- Odjemalčeva stran:Imperativni način manipulacije z elementi (npr. z jQuery)

```
$("#textbox").kendoNumericTextBox({
    value: 10,
    min: -10,
    max: 100,
    step: 0.75,
    format: "n",
    decimals: 3
});
```

- Slabosti ? Počasno izvajanje, saj moramo najprej poiskat v DOM hierarhiji dani element

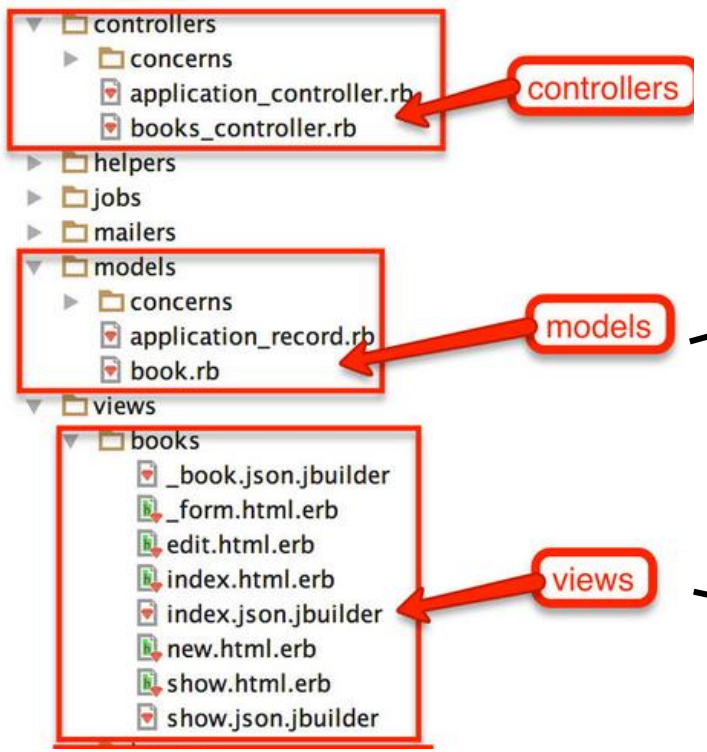
Uvod: 2. generacija web front-end-a

- **Ruby on Rails** (2005) in **Django** (Python) (2005)
- Še danes v obsežni uporabi (npr. **GitHub**)
- Močna utilizacija **MVC** (model-view-controller) stila dekompozicije spletne aplikacije
- Objektno-relacijsko mapiranje (**ORM** – object-relational mapping) – omogoča hitro preslikavo podatkov iz podatkovnih baz v objekte

+

Uvod: 2. generacija web front-end-a

- Primer Ruby MVC:




```
class StoriesController < ActionController::Base
  def index
    @variable = 'Value being passed to a view'
  end
end
```

```
CREATE TABLE stories (
  "id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  "name" varchar(255) DEFAULT NULL,
  "link" varchar(255) DEFAULT NULL,
  "created_at" datetime DEFAULT NULL,
  "updated_at" datetime DEFAULT NULL
);
```

```
<p>The instance variable @variable contains: <%= @variable %></p>
```

Uvod: 3. generacija web front-end-a

- Breme prenesemo na odjemalčevo stran
  - Zmanjšamo **prenos podatkov**
  - Končni programski jezik: **JavaScript**
  - **Deklerativni** način spreminjanja elementov
  - Komunikacija s strežnikom preko **REST API** izključno za posodobitev podatkov (običajno v JSON)
  - Močni temelji na **virtualnem DOM-u**
- Obstoječe rešitve 1. in 2. generacije se ohranijo:
  - Uporaba **šablonskega naslavljanja**
  - Uporaba **komponent** s katerimi je možno sestavit: V, MVC in **MVVM (view,viewmodel,model)**
- Različna spletna ogrodja in knjižnice:
  - **Angular** (2010), **React** (2013), **Vue**, Ember, in mnogo drugih

Uvod: 3. generacija web front-end-a

- Šablonski način in deklarativno definiranje videza (npr. v Angular):
- Prednosti?
  - Elementa ne rabimo iskat po DOM-u brskalnika
  - Ne zanima nas **kako spremenit** element (imperativni način) ampak **kaj bomo** spremenili (deklerativni način)
  - Večja fleksibilnost pri spremembi videza

```
$("#textbox").kendoNumericTextBox({
    value: 10,
    min: -10,
    max: 100,
    step: 0.75,
    format: "n",
    decimals: 3
});
```

```
<input kendo-numerictextbox k-min="-10" k-max="100" k-step="0.75"
  k-format="n" k-decimals="3"/>
```

Uvod: 3. generacija web front-end-a

- Uporaba komponent in šablon preko JS
- Primer JSX (JavaScript XML) v Reactu:

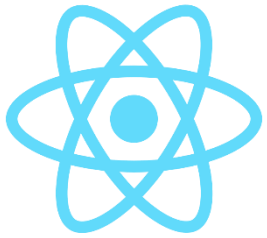
```
const subject = "World";
const header = (
  <header>
    <h1>Hello, {subject}!</h1>
  </header>
);
```

```
const subject = "World";
const header = React.createElement(
  "header",
  null,
  React.createElement("h1", null, "Hello, ", subject, "!"),
);
```

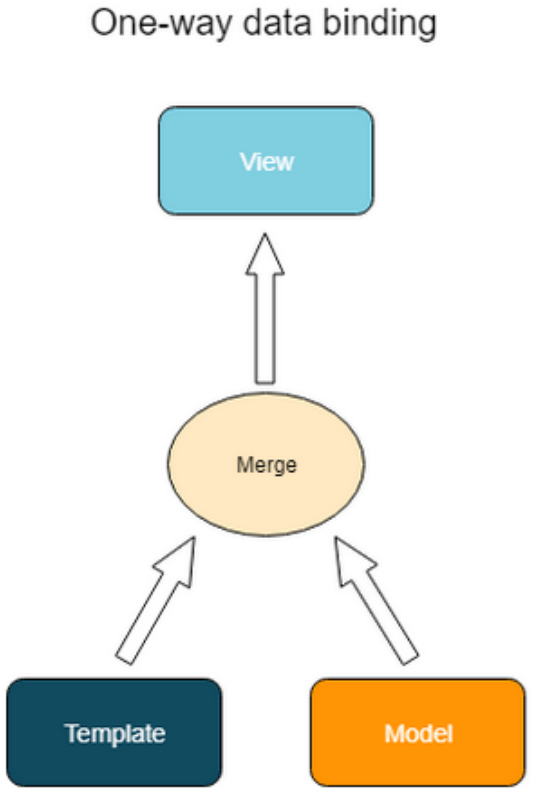
```
<header>
  <h1>Hello, World!</h1>
</header>
```

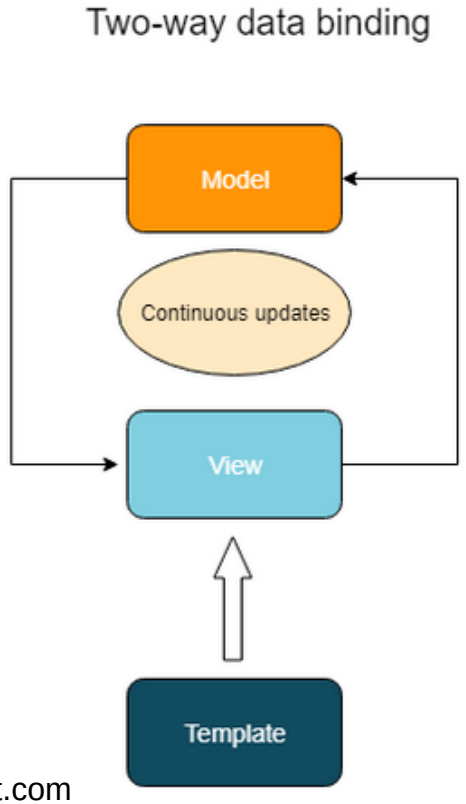
Uvod: 3. generacija web front-end-a

- Enosmerna ali dvosermna izmenjava podatkov iz pogleda v model ?
  - **Enosmerno**: bolj stabilna koda, **potrebna dodatna logika** za obratno smer posodobitve
  - **Dvosmerno**: lažja uporaba, saj je potrebne manj logike na strani kontrolerja, **ni primerno za večje količine podatkov**



React + JSX





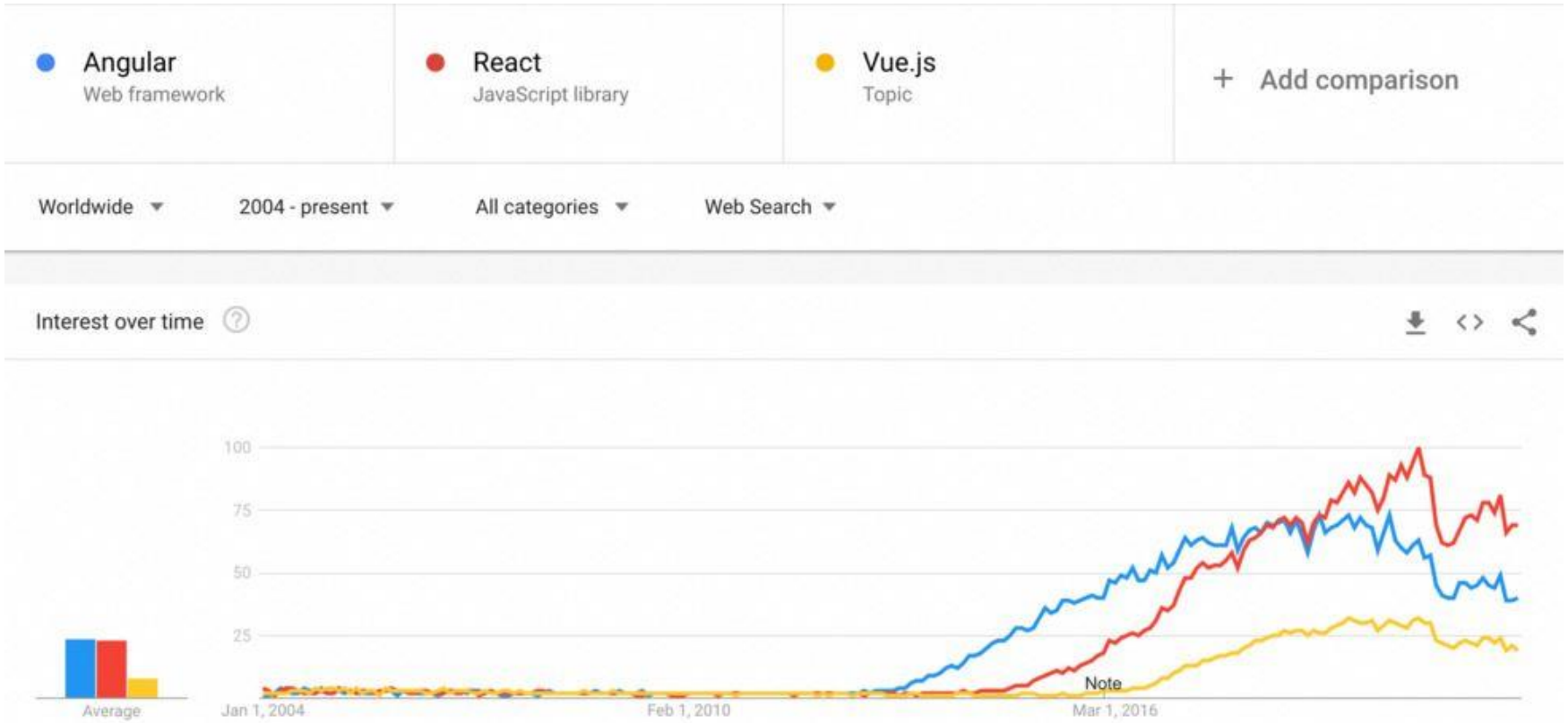
Angular

Vue

Vir slike: altexsoft.com

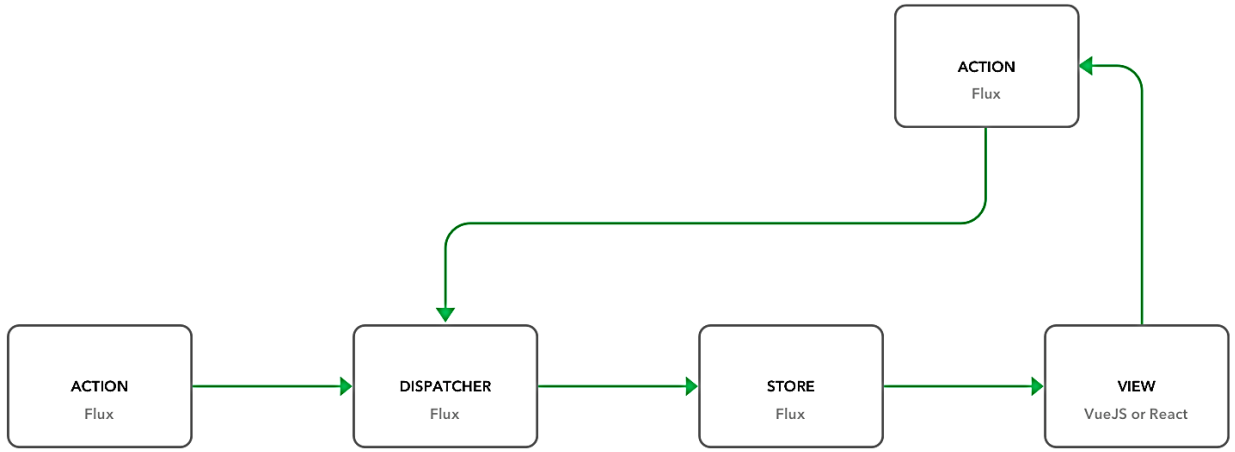
Uvod: 3. generacija web front-end-a

- Tri najbolj uporabljene front-end rešitve



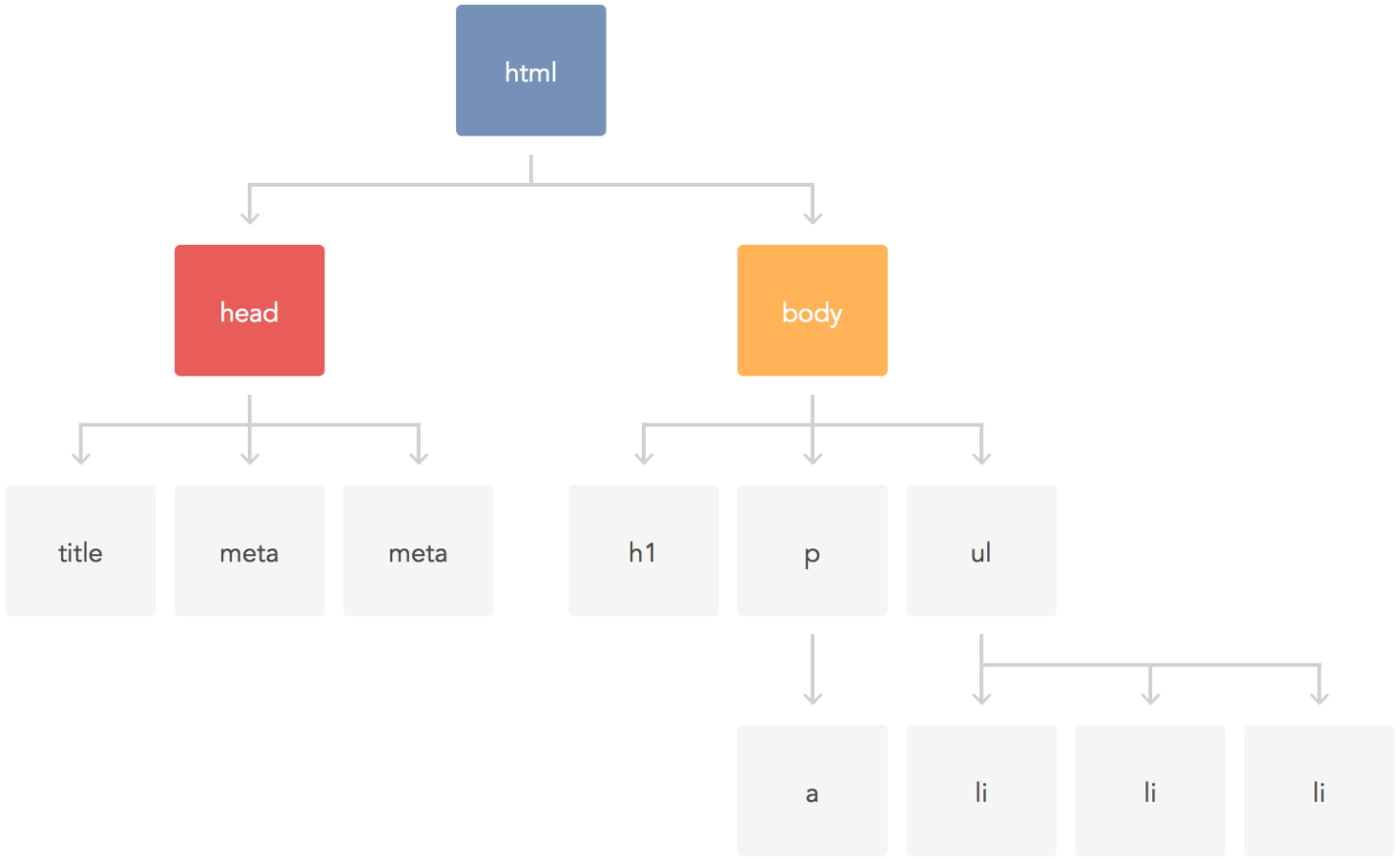
FLUX

- Razvil Meta, enostavnost proti drugim vzorcem kot MVC
  - Uporabljata React in Vue.js
- Spremenljivke so predstavljene kot **stanja**
  - Action (spremenimo podatek) -> dispatcher (logika) -> store (hramba stanja) -> view (prikaz)
- **Reaktivnost** (spremljamo spremembe stanja, npr. vue.$watch)



Vir slike: Mark Darlymple

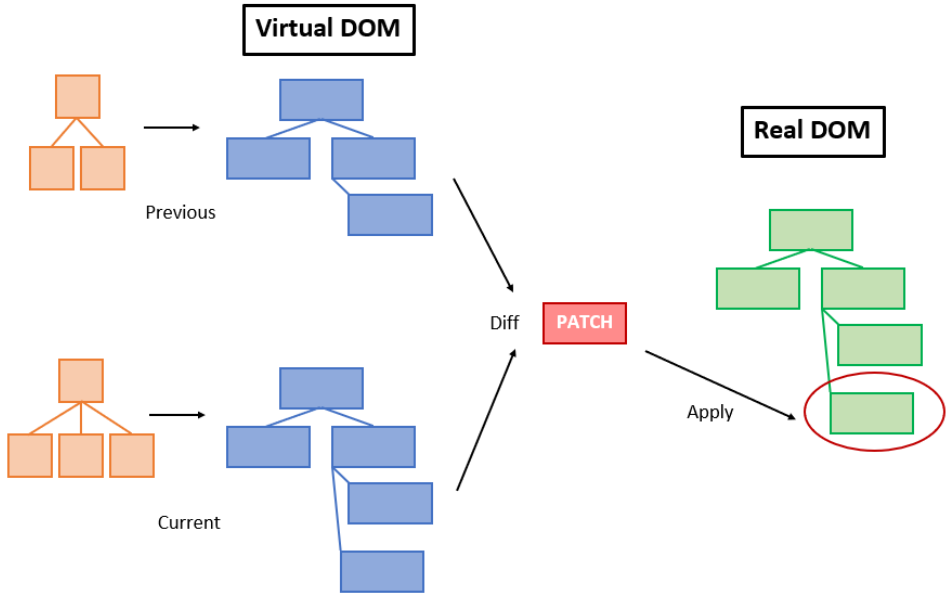
Uvod: Document Object Model (DOM)



Vir slike: Sebastian Peyrott, auth0.com

Virtualni DOM

- Klasika pri večini spletnih ogrodjih (React itd)
- Omogoča **“hitrejše” iskanje elementov** (komponent) in njihovo **manipulacijo**
- **vDOM je praviloma implementiran v JS**
- Uporaba začasnih komponent (npr. **virtual scrolling**)
- vDOM se primerja s prejšnjo verzijo vDOM, nato se popravi DOM v brskalniku
- Slabosti?
  - Dodatna potreba po **pomnilniku**
  - Primerjava dveh virtualnih DOM
  - **Polimorifzem** JS objektov (bomo še spoznali..)
- Ali je so operacije vDOM hitrejše od DOM-a brskalnika ?

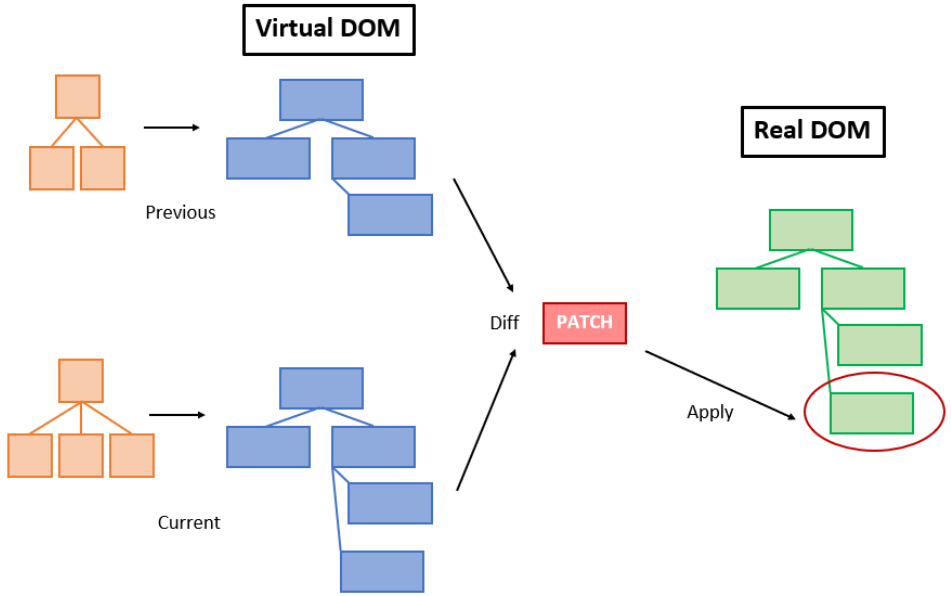


Vir slike: Sebastian Peyrott, auth0.com

Virtualni DOM

- Slabosti?
  - Dodatna potreba po **pomnilniku**
  - Primerjava dveh virtualnih DOM
  - **Polimorifzem**
- Ali je so operacije vDOM hitrejše od DOM-a brskalnika ?

  Ne, vendar se velikokrat celotne komponente zamenjujejo, kar doprinese do **preveč operacij nad DOM** brskalnika in **nepotrebnih ponovnih izrisov celotne komponente**. Z vDOM minimiziramo število operacij nad pravim DOM in št. izrisov.



Vir slike: Sebastian Peyrott, auth0.com

Virtualni DOM - šolski primer

- Ustvarimo lastni vDOM:

```
<div id="app">
  Hello world
  <img src="https://media.gi
</div>
```

```
import createElement from './vdom/createElement';
import render from './vdom/render';

const vApp = createElement('div', {
  attrs: {
    id: 'app',
  },
  children: [
    'Hello world',
    createElement('img', {
      attrs: {
        src: 'https://media.giphy.com/media/cuPm4p4pClZVC/giphy.gif',
      },
    }),
  ],
});

const $app = render(vApp);
console.log($app);
```

- Vir: https://dev.to/ycmjason/building-a-simple-virtual-dom-from-scratch-3d05

Virtualni DOM - šolski primer

- Ustvarimo svoj createElement in render funkciji
- Lastni createElement naredi reprezentacijo DOM vozlišča (node-a) kot JS objekt
- Render kliče ukaze direktno nad DOM brskalnika (document objekt)

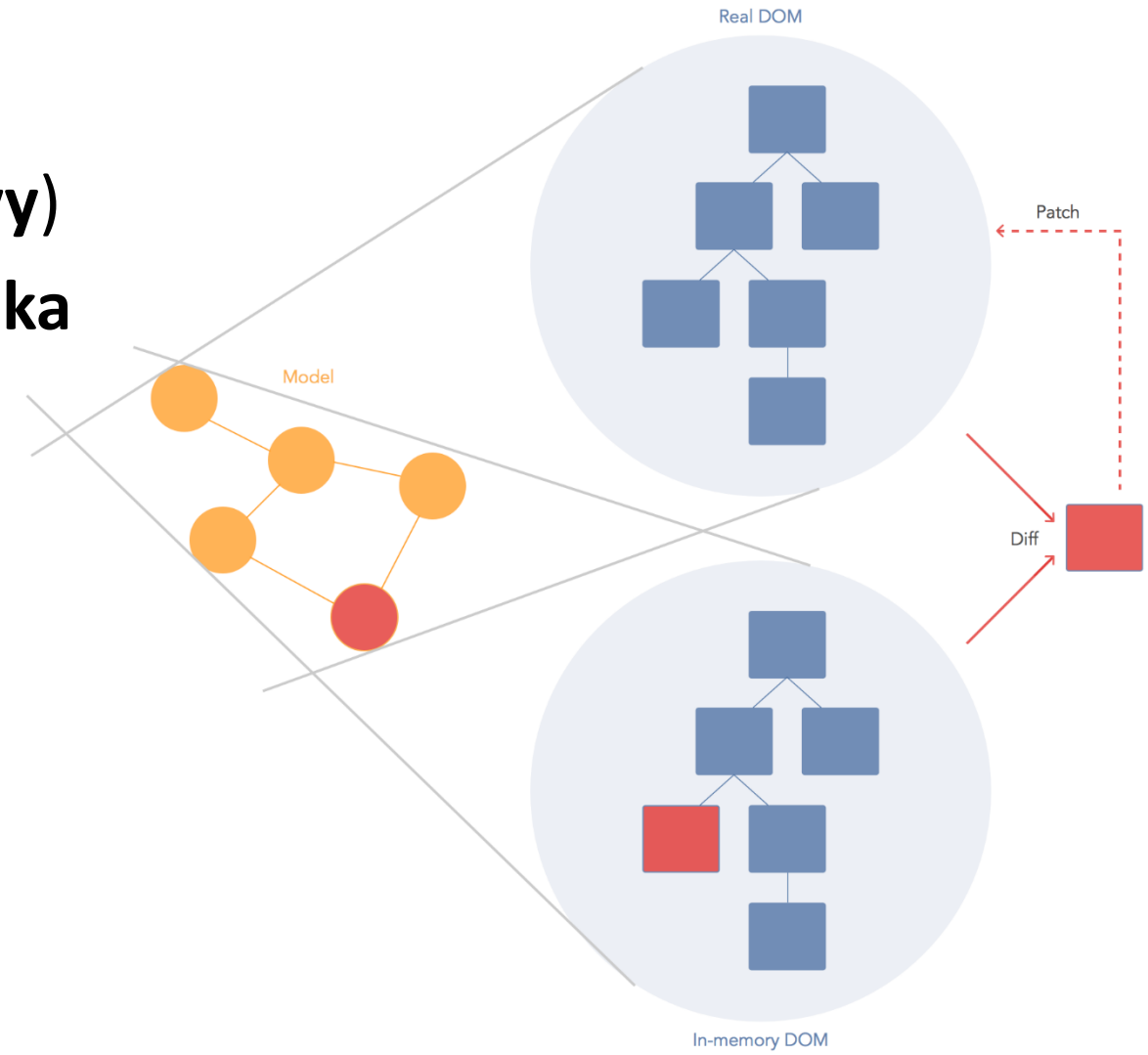
```
export default (tagName, { attrs, children }) => {
  const vElem = Object.create(null);

  Object.assign(vElem, {
    tagName,
    attrs,
    children,
  });

  return vElem;
};
```

```
const renderElem = ({ tagName, attrs, children}) => {
  // create the element
  //   e.g. <div></div>
  const $el = document.createElement(tagName);

  // add all attributs as specified in vNode.attrs
  //   e.g. <div id="app"></div>
  for (const [k, v] of Object.entries(attrs)) {
    $el.setAttribute(k, v);
  }

  // append all children as specified in vNode.children
  //   e.g. <div id="app"><img></div>
  for (const child of children) {
    $el.appendChild(render(child));
  }

  return $el;
};

export default render;
```

- Vir: https://dev.to/ycmjason/building-a-simple-virtual-dom-from-scratch-3d05

Inkrementalni DOM

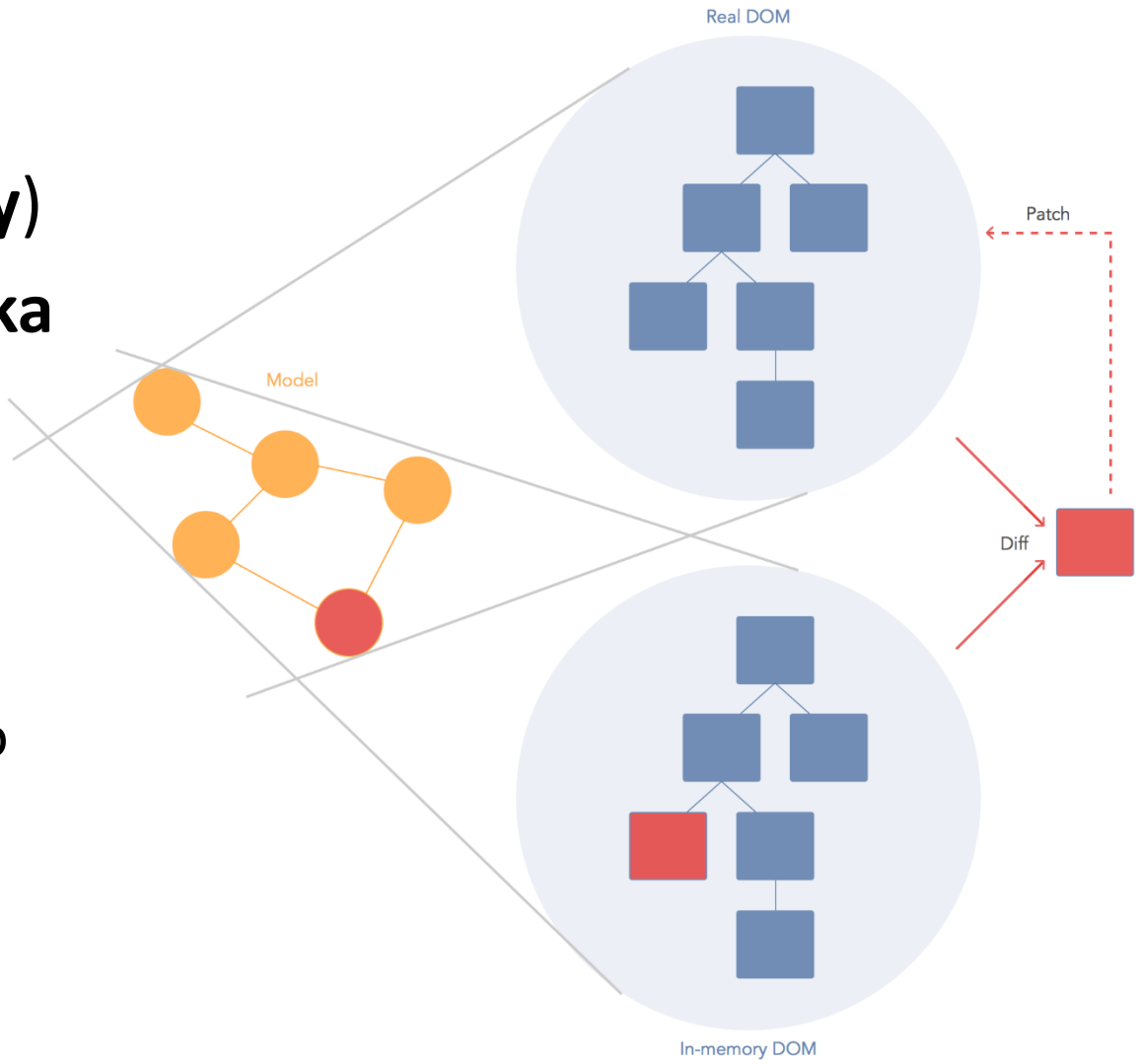
- Drugačni pristop (npr. uporabljata Ember, **Angular Ivy**)
- **Zmanjšana uporaba pomnilnika** (dobro za mobilne naprave)
- DOM primerjamo s zadnjo verzijo vDOM, popravljamo nato original DOM
- Slabost?



Vir slike: Sebastian Peyrott, auth0.com

Inkrementalni DOM

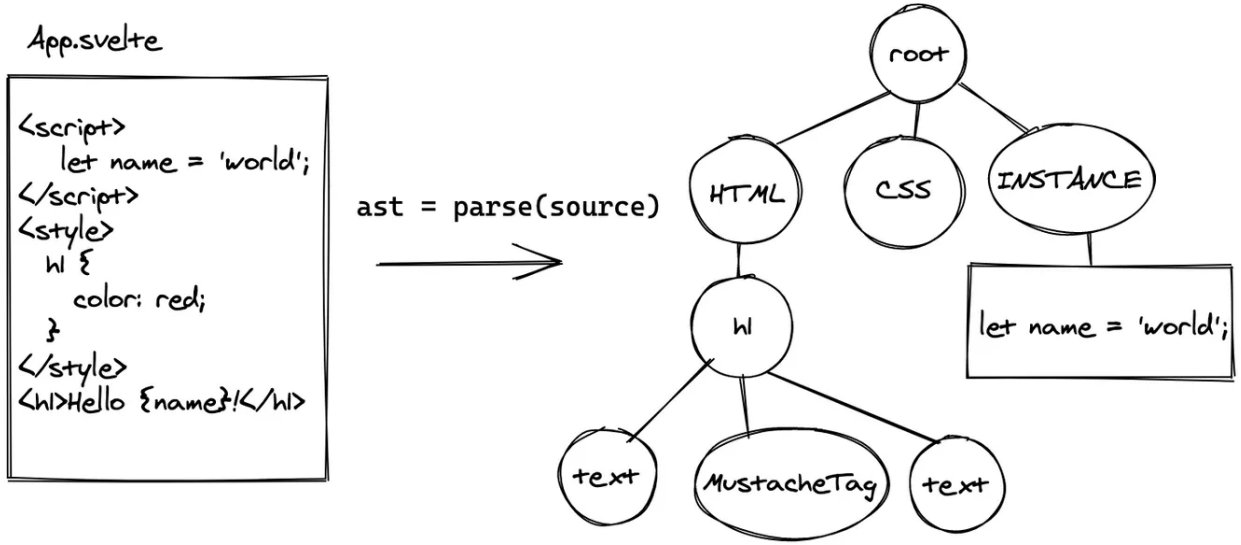
- Drugačni pristop (npr. uporabljata Ember, **Angular Ivy**)
- **Zmanjšana uporaba pomnilnika** (dobro za mobilne naprave)
- DOM primerjamo s zadnjo verzijo vDOM, popravljamo nato original DOM
- Slabost ? Pri večji manipulaciji lahko počasnejše kot klasični vDOM



Vir slike: Sebastian Peyrott, auth0.com

Alternativa vDOM-u: SVELTE

- Vsa koda se prevede v končno obliko, kjer uporablja vanilla JS
- Prebiranje preko AST (abstraktno sintaktično drevo)
- Operacije direktno nad DOM, prednost: hitrost + manj pomnilnika



Vir slike: https://lihautan.com/the-svelte-compiler-handbook/

Primerjava 2022 (svelte, react, vue, angular)

Hitrost:

Name Duration for...	vanillajs-1	vue-v3.2.26	svelte-v3.46.2	react-hooks-v18.0.0	angular-v13.0.0
Implementation notes	772				
create rows creating 1,000 rows	81.7 ±6.2 (1.00)	103.5 ±4.0 (1.27)	120.8 ±6.4 (1.48)	112.6 ±3.8 (1.38)	113.7 ±3.5 (1.39)
replace all rows updating all 1,000 rows (5 warmup runs).	77.5 ±0.5 (1.00)	92.2 ±1.3 (1.19)	97.9 ±1.3 (1.26)	94.7 ±2.0 (1.22)	104.8 ±2.4 (1.35)
select row highlighting a selected row. (no warmup runs). 16x CPU slowdown.	20.2 ±1.4 (1.00)	33.1 ±1.2 (1.64)	29.9 ±0.9 (1.48)	60.9 ±2.1 (3.02)	52.8 ±1.3 (2.62)
swap rows swap 2 rows for table with 1,000 rows. (5 warmup runs). 4x CPU slowdown.	46.1 ±0.4 (1.00)	48.9 ±0.6 (1.06)	49.1 ±0.7 (1.07)	328.0 ±3.2 (7.11)	352.4 ±5.8 (7.64)
create many rows creating 10,000 rows	811.6 ± 39.7 (1.00)	1,029.1 ± 14.5 (1.27)	975.8 ± 24.9 (1.20)	1,293.4 ± 39.2 (1.59)	1,112.5 ± 10.4 (1.37)
clear rows clearing a table with 1,000 rows. 8x CPU slowdown.	46.3 ±1.8 (1.00)	62.7 ±0.7 (1.35)	69.8 ±1.6 (1.51)	63.9 ±1.8 (1.38)	139.3 ±1.2 (3.01)
geometric mean of all factors in the table	1.00	1.28	1.32	2.07	2.32

Pomnilnik:

Name	vanillajs-1	vue-v3.2.26	svelte-v3.46.2	react-hooks-v18.0.0	angular-v13.0.0
ready memory Memory usage after page load.	1.5 (1.00)	1.7 (1.14)	1.5 (1.01)	1.8 (1.23)	2.2 (1.52)
run memory Memory usage after adding 1000 rows.	1.4 (1.00)	3.4 (2.36)	2.4 (1.70)	4.1 (2.87)	4.3 (2.96)
update every 10th row for 1k rows (5 cycles) Memory usage after clicking update every 10th row 5 times	1.4 (1.00)	3.7 (2.53)	2.4 (1.66)	4.6 (3.18)	4.6 (3.15)
replace 1k rows (5 cycles) Memory usage after clicking create 1000 rows 5 times	1.7 (1.00)	3.7 (2.20)	2.6 (1.55)	4.8 (2.87)	4.9 (2.89)
creating/clearing 1k rows (5 cycles) Memory usage after creating and clearing 1000 rows 5 times	1.2 (1.00)	1.7 (1.39)	1.6 (1.26)	2.3 (1.85)	2.8 (2.31)
geometric mean of all factors in the table	1.00	1.83	1.41	2.26	2.48

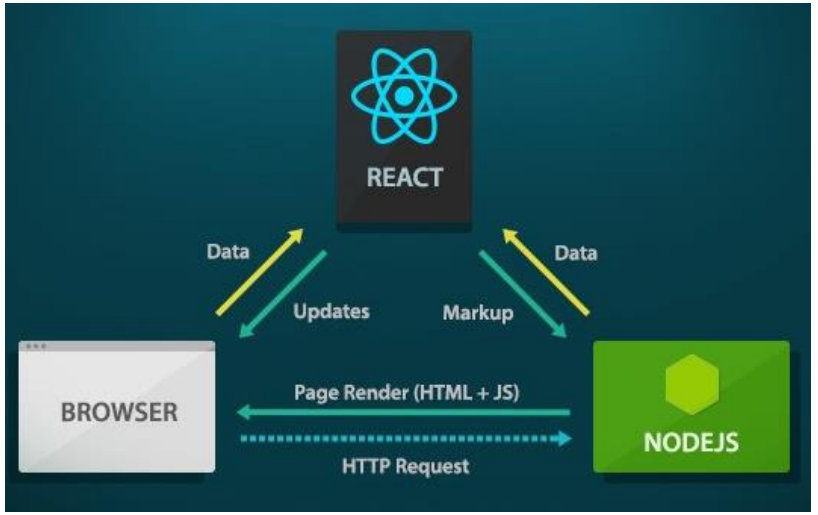
Zagon:

Name	vanillajs-1	vue-v3.2.26	svelte-v3.46.2	react-hooks-v18.0.0	angular-v13.0.0
consistently interactive a pessimistic TTI - when the CPU and network are both definitely very idle. (no more CPU tasks over 50ms)	1,955.7 ± 1.2 (1.00)	2,105.9 ± 0.8 (1.08)	1,955.5 ± 0.3 (1.00)	2,555.7 ± 0.8 (1.31)	2,817.4 ± 22.4 (1.44)
main thread work cost total amount of time spent doing work on the main thread. includes style/layout/etc.	160.0 ±4.8 (1.00)	180.0 ± 13.6 (1.13)	170.8 ±1.3 (1.07)	197.7 ± 20.3 (1.24)	320.9 ±9.6 (2.01)
total kilobyte weight network transfer cost (post-compression) of all the resources loaded into the page.	147.3 ±0.0 (1.01)	195.3 ±0.0 (1.34)	146.1 ±0.0 (1.00)	260.1 ±0.0 (1.78)	294.5 ±0.0 (2.02)
geometric mean of all factors in the table	1.00	1.17	1.02	1.42	1.80

Vir slike: https://www.kodaps.dev/en/blog/angular-vs-vue-js-vs-react-vs-svelte-in-2022-by-the-numbers

SSR, pre-rendering, prefetching

- Server-side-rendering (SSR)
  - Nasprotje od upodabljanja na strani odjemalca (client-side-rendering, **CSR**)
  - Na strežniškem delu (npr. v node) zgeneriramo HTML+JS dinamično ob interakciji uporabnika
  - Prednost: **hitrejši FCP** (first contentful paint)
  - Prednost: **SEO** (search engine optimization)
  - Prednost: **manj obremenjajoče** za odjemalce (v primeru, da strežnik ni preobremenjen)
  - Slabost: **višja latenca**, vsaj moramo počakat, da se stran zgenerira na strežniški strani pred prejetjem



Vir slike: Neetu Das, Hirelra

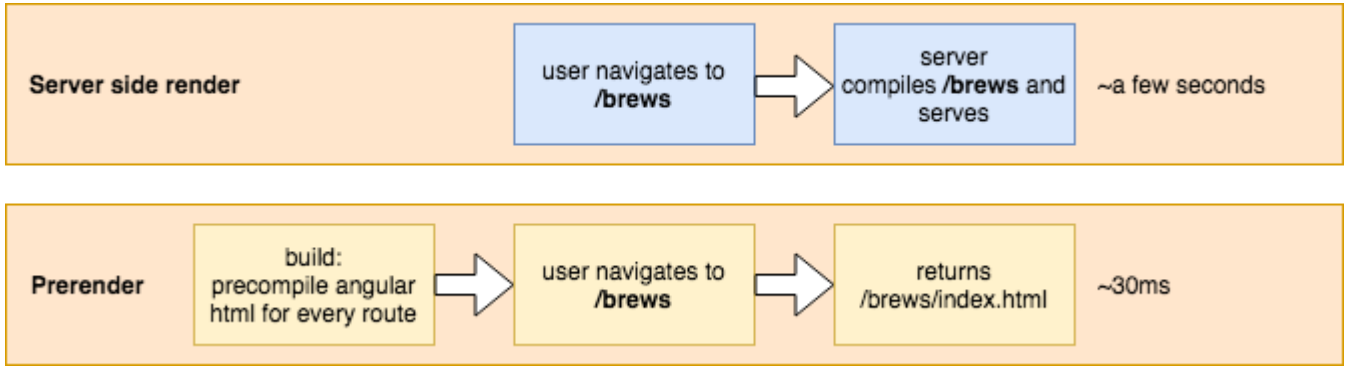
SSR, pre-rendering, prefetching

- SSR + rehidracija (rehydration)
  - Za **prvo upodabljanje** strani uporabimo SSR -> razbremenimo odjemalca in **hitrejši FCP**
  - Nadaljna dinamika in upodabljanje preko odjemalca

```
<h1>To Do's</h1>
<ul>
  <li><input type="checkbox"> Wash dishes</li>
  <li><input type="checkbox" checked> Mop floors</li>
  <li><input type="checkbox"> Fold laundry</li>
</ul>
<footer><input placeholder="Add To Do..."></footer>
<script>
  var DATA = {"todos":[
    {"text":"Wash dishes","checked":false,"created":1546464530049},
    {"text":"Mop floors","checked":true,"created":1546464571013},
    {"text":"Fold laundry","checked":false,"created":1546424241610}
  ]}
</script>
<script src="/bundle.js"></script>
```

SSR, pre-rendering, prefetching

- Prerendering
  - Za **vse možne poti** aplikacije se zgenerirajo HTML+JS datoteke
  - Podpora SEO (search engine optimization)
  - Nalaganje **statičnih komponent ogrodja** je veliko hitrejše na odjemalčevi strani
  - Podpora v večini spletnih ogrodjih (Angular), ali uporaba npr. **Prerender.io** modula
  - Slabost ? Ni možno uporabit za **dinamične komponente**



Vir slike: Brecht Billiet, blog.strongbew.io

SSR, pre-rendering, prefetching

- Prefetching
  - Podpora v HTML5, razširitev na večino spletnih ogrodij
    - Še v razvoju za večino brskalnikov: https://en.wikipedia.org/wiki/Link_prefetching
  - Različne strategije:
    - **DNS-prefetch**: želimo DNS poizvedbe izvesti vnaprej

      <link rel="dns-prefetch" href="//example.com">
    - **Pre-connect**: želimo, da se brskalnik poveže na oddaljeno stran pred klikom

      <link rel="preconnect" href="https://css-tricks.com">
    - **Prefetch**: zunanji vir vnaprej naložimo

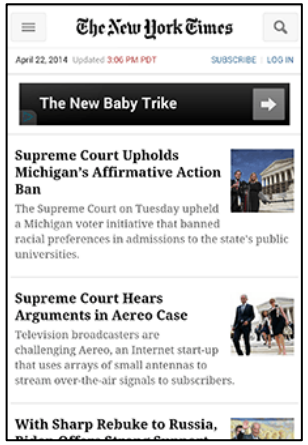
      <link rel="prefetch" href="image.png">
    - **Pre-render**: zunanjo stran/podstran na katero linkamo že vnaprej naložimo, čeprav še je uporabnik ni obiskal

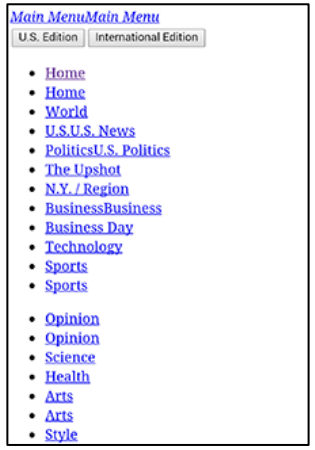
      <link rel="prerender" href="https://css-tricks.com">

  - Preverimo v Chrome brskalniku: chrome://predictors/

Blokiranje

- Novejši CSS omogoča media atribut pri definiranju stilov, na podlagi katerega se brskalnik odloči, če bo počakal (l**azy-loading**) na nalaganje CSS pred izrisom (**blokiran način**)





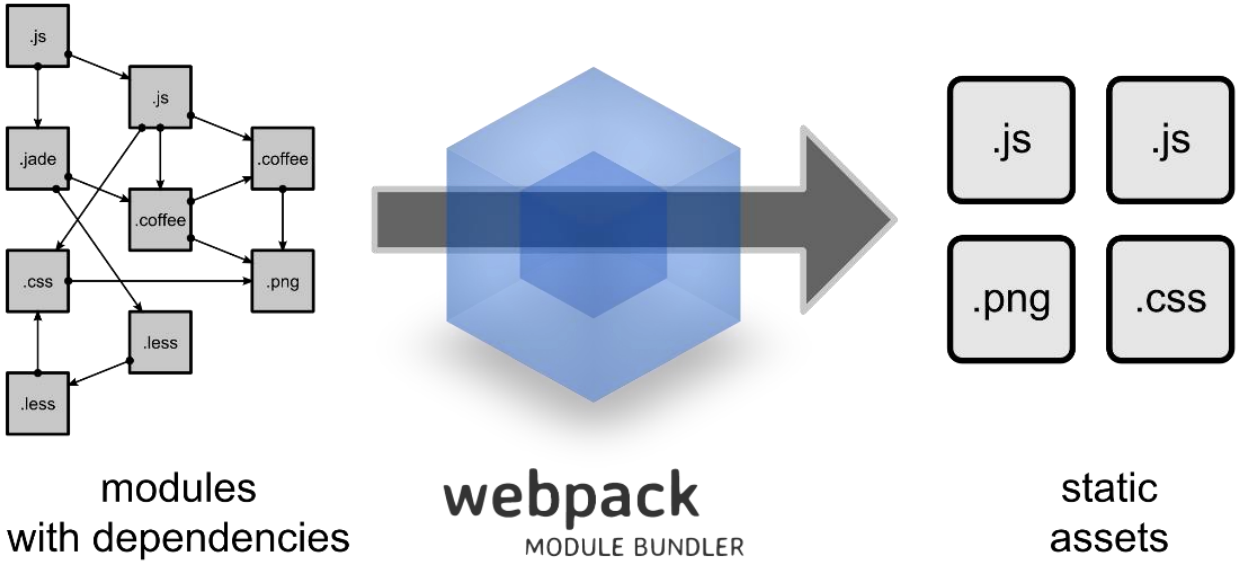

```
<link href="style.css" rel="stylesheet" media="all" />
<link href="portrait.css" rel="stylesheet" media="(orientation:portrait)" />
<link href="print.css" rel="stylesheet" media="print" />
```

- Enako lahko uporabimo pri elementih z **loading** atributom (eager ali lazy)

```
<img src="image.jpg" alt="..." loading="lazy" />
<iframe src="video-player.html" title="..." loading="lazy"></iframe>
```

Tree-shaking

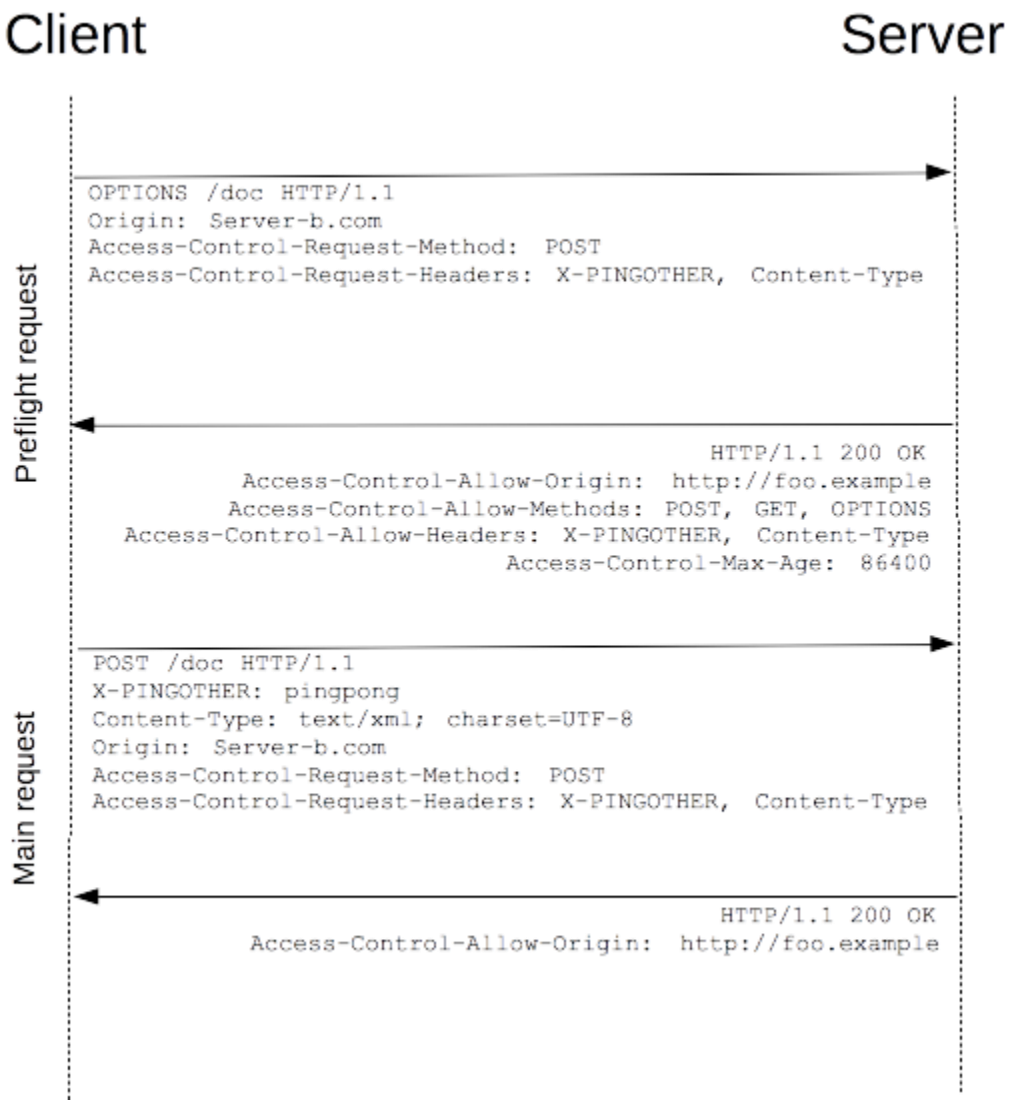
- Od kar ES2015 podpira nalaganje posameznih JS komponent preko **import** funkcionalnosti, je smiselno da ne naložimo celotne JS kode spletnega ogrodja
- Rešitev je preko ti. statične analize kode, znano orodje je ti. **Webpack**
- Statična analiza se izvede samo enkrat nad produkcijsko kodo, in ne v času izvajanja (runtime)
- Npr. v Angular Ivy možno v času prevajanja (Typescript -> JS)



Vir slike: webpack.github.io

Cross-origin resource sharing (CORS)

- Če pri REST API zahtevkih uporabimo kompleksnejše poizvedbe na strežnik iz **druge domene**, je običajno poslan še dodatni zahtevek **OPTIONS**, ki preveri, če strežnik sprejema zahtevke od zunaj
- Enostavna rešitev problema je **uporaba proxy strežnika** na strani domene spletne strani
  - Slabost: dodatni sloj in upočasnitev delovanja
- Boljša rešitev je uporaba **Access-Control-Max-Age** v glavi HTTP paketka, ki jih vrača spletni strežnik:
  - Primer za NGINX: https://enable-cors.org/server_nginx.html
  - V takem primeru se OPTIONS zahtevek pošlje le enkrat, nato se rezultat predpomni v brskalniku za določen čas



Spomnimo se JSONP

- Zaobitev CORS preko **JSONP**, uporabimo **<script>** namesto AJAX

- Ni priporočeno, zakaj?
  - Težava pri napakah (npr. če se zahtevek ni izvedel)
  - Imamo na voljo samo GET zahtevke

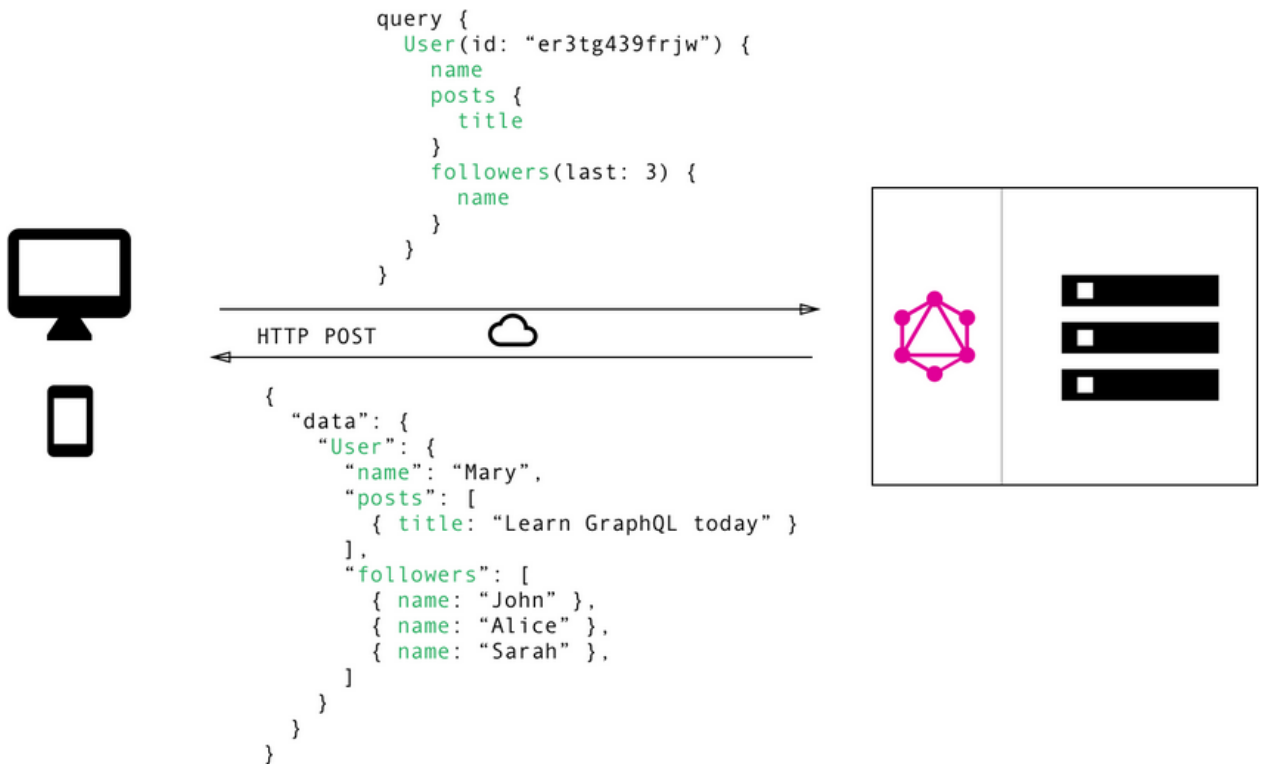
Realnočasovne front-end aplikacije

- Kadar se dinamika strani spreminja konstantno: chat, twitter stream, računalniške igrice itd.
- Potrebna obojestranska (in hitra) komunikacija
- V ta namen se je razvil Websockets protokol (aplikacijski na TCP)
- Lahko uporabimo tudi HTTP/2.0
  - Morebitne težave ? Privzeti Push mehanizem deluje v ozadju in ni dostopa do podatkov v kodi aplikaciji
  - Rešitev je uporaba SSE (server send events)
    - https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events

```
'Cache-Control': 'no-cache',
'Content-Type': 'text/event-stream',
'Connection': 'keep-alive'
```

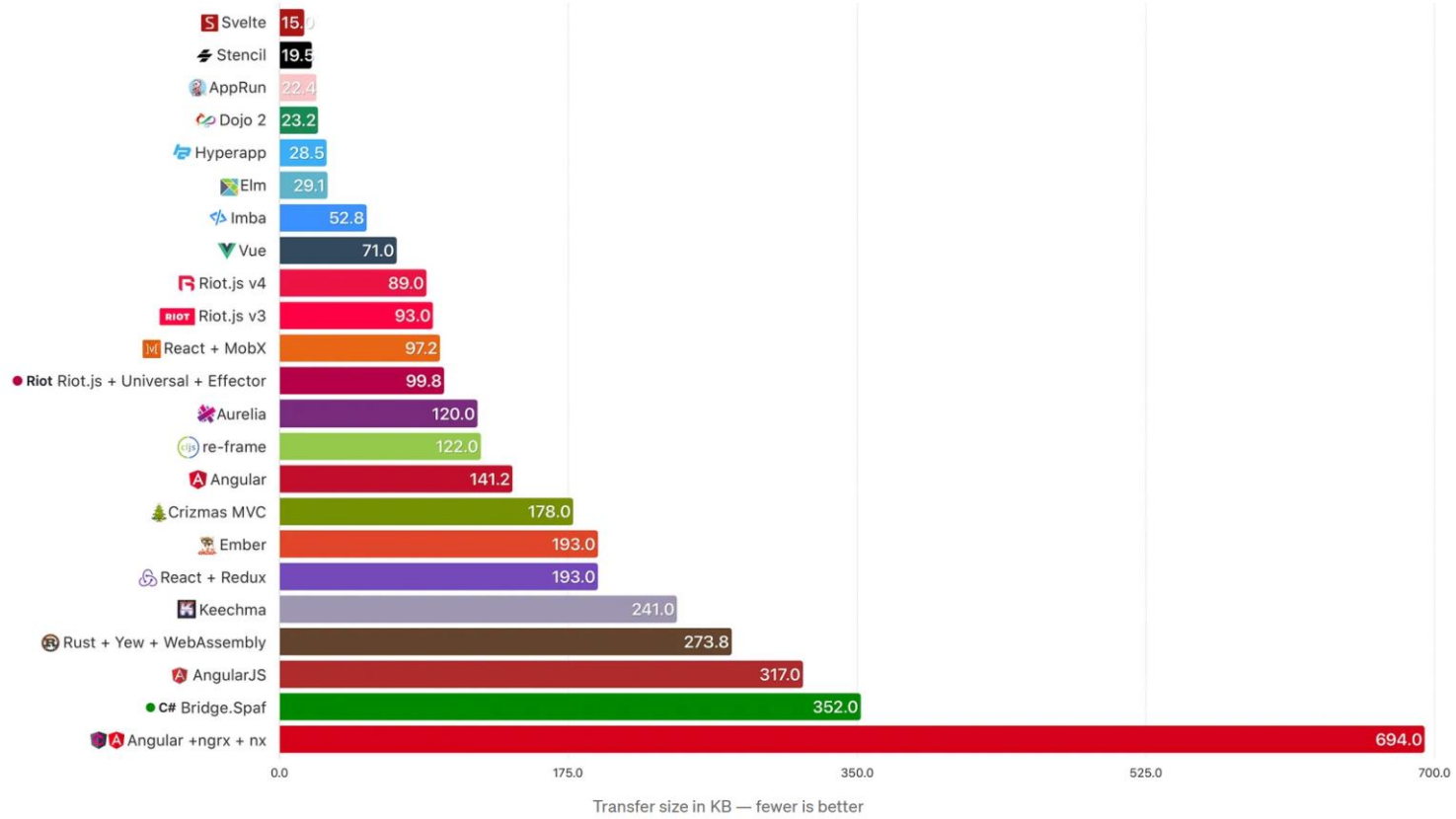
GraphQL vs REST

- Pri GraphQL lahko prejšna dva zahtevka združimo v enega
  - Privzeto uporabimo POST za pošiljanje poizvedbe
  - Pri tem smo poizvedli še za en vir, kateri?
  - Podobna sintaksa kot JSON
  - **Prednosti/slabosti napram REST?**



Zmogljivost ? (2020)

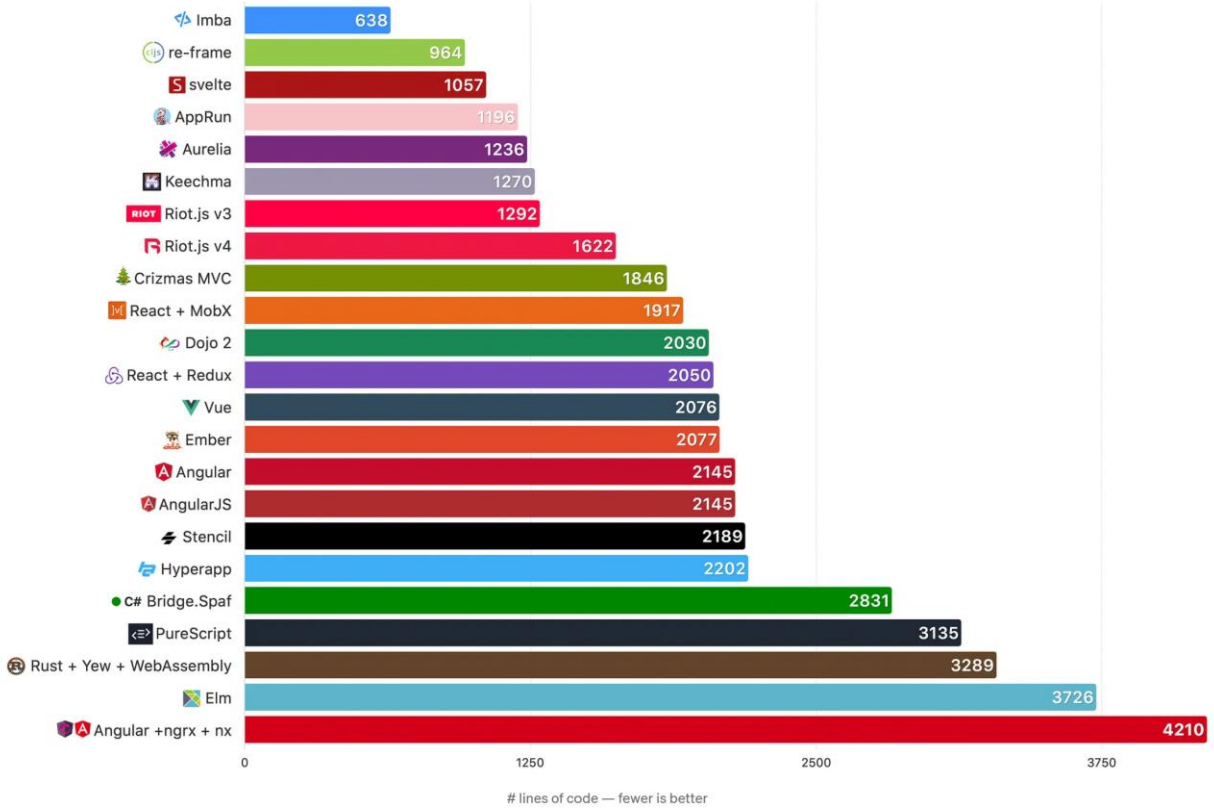
- Pomnilniška poraba (po **gzip-u**):



- Vir: https://medium.com/dailyjs/a-realworld-comparison-of-front-end-frameworks-2020-4e50655fe4c1

Zmogljivost ? (2020)

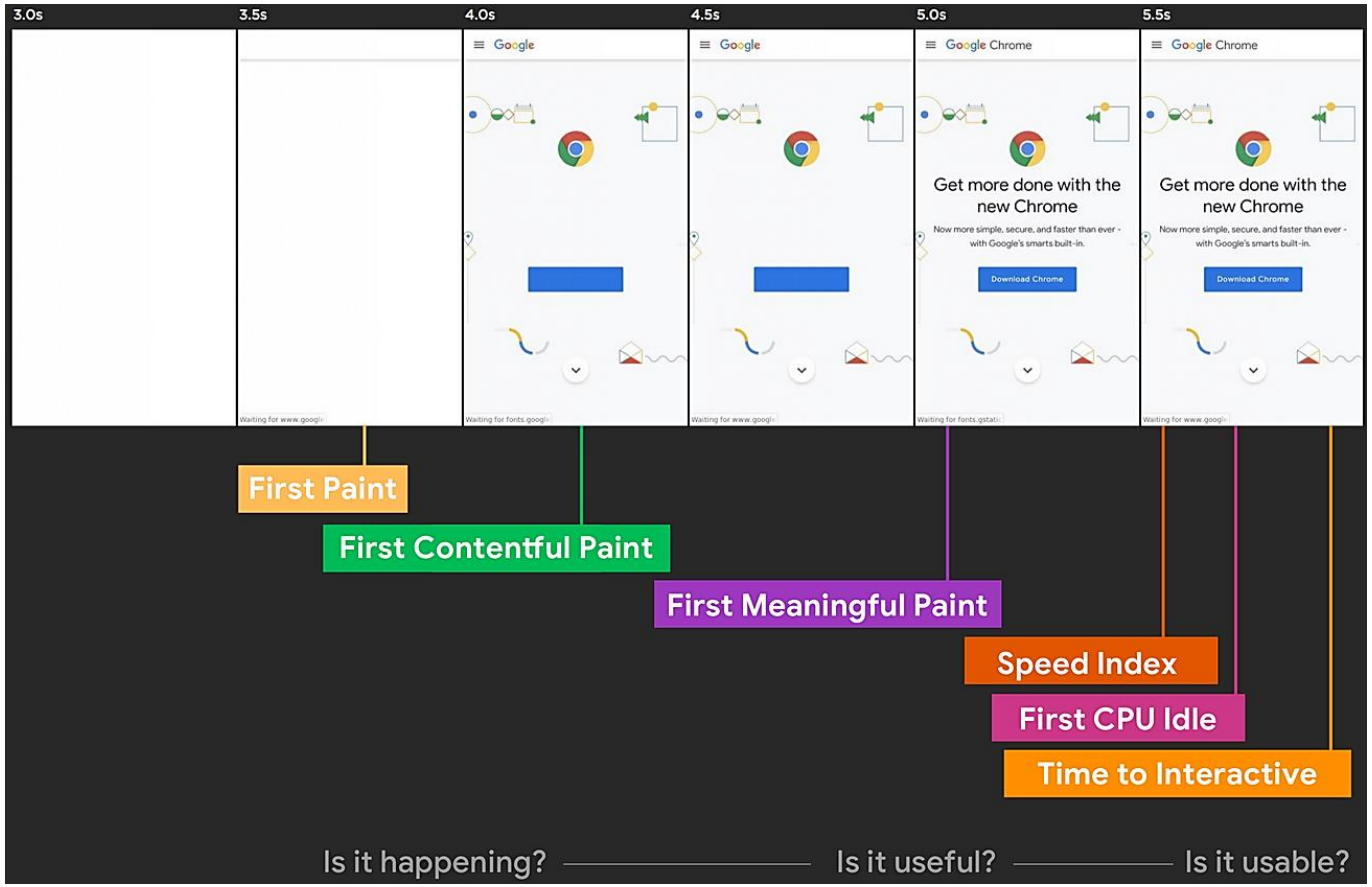
- Število porabljenih vrstic za implementacijo enake aplikacije (**RealWorld APP**):



- Vir: https://medium.com/dailyjs/a-realworld-comparison-of-front-end-frameworks-2020-4e50655fe4c1

Zmogljivost ?

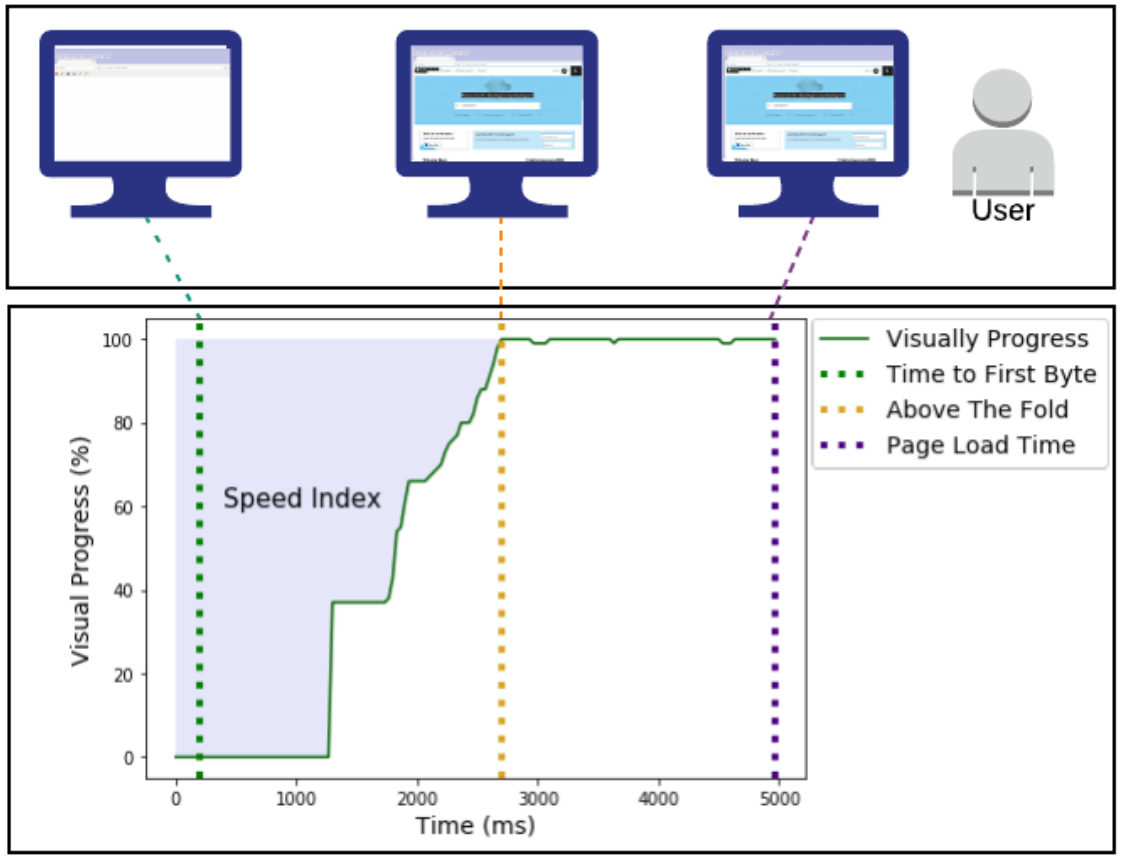
- Kdaj uporabnik lahko uporablja spletno aplikacijo?
- Preverimo Chrome Lighthouse razširitev



Vir slike: Addy Osmani, Google

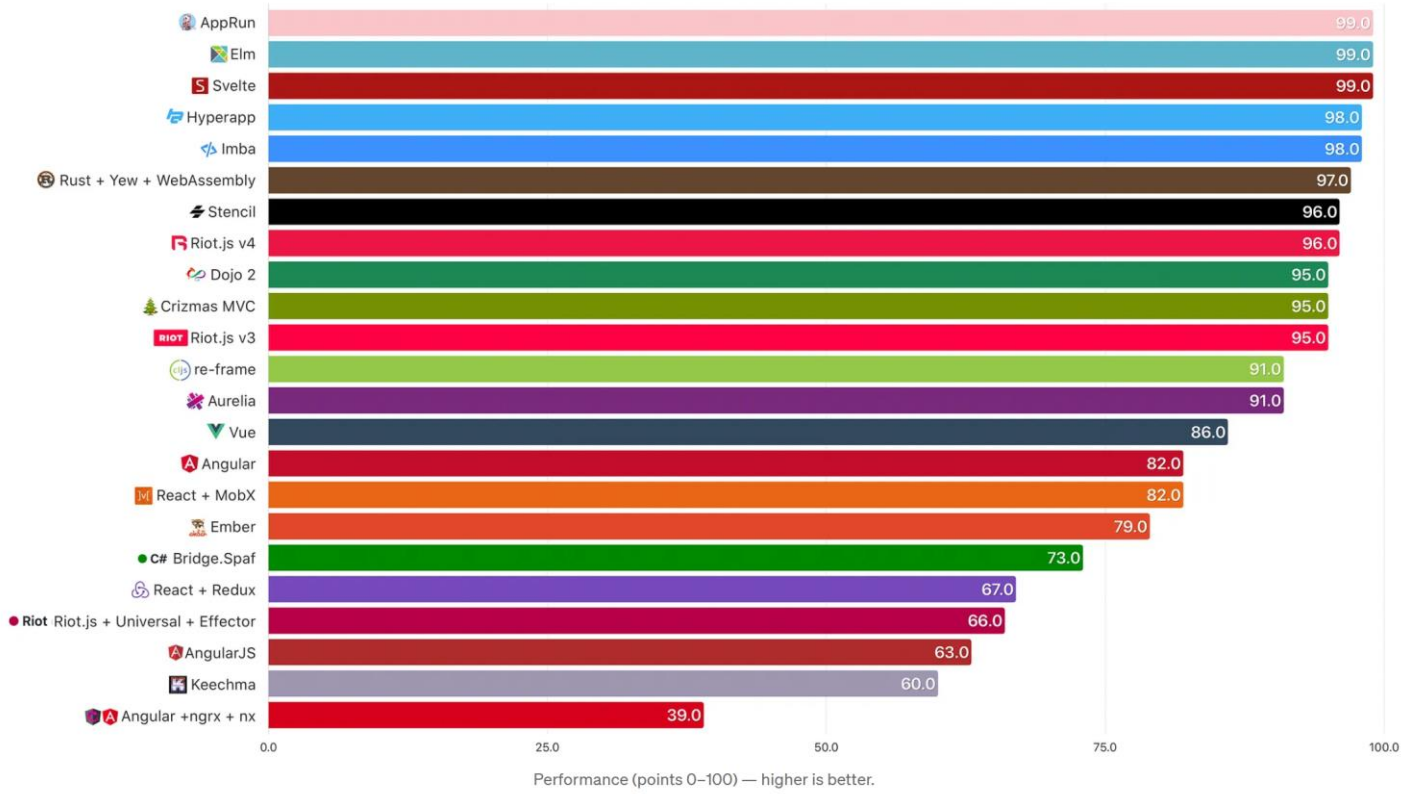
Speed index?

- Kako hitro se populira vidni del spletne strain do uporabnika (viewport).
- Bolj navezana metrika na **UX** kot UI.



Zmogljivost ? (2020)

- Hitrost nalaganja + izrisa komponent (kombinacija različnih metrik):
  - First Contentful Paint
  - First Meaningful Paint
  - Speed Index
  - First CPU Idle
  - Time to Interactive
  - Estimated Input Latency



- Vir: https://medium.com/dailyjs/a-realworld-comparison-of-front-end-frameworks-2020-4e50655fe4c1

Trg front-end programerjev (2023) ?

Vir: talent.com

Zaključek


BRACE YOURSELVES
ANOTHER WEB FRAMEWORK IS COMING...
JS
REQUIRE·JS
npm
TS