170 praktických ukázek

CSS

MODERNÍ LAYOUT

Grid, flexbox a nové metody rozvržení webů v příkladech

Martin Michálek

Tento e-book je zabezpečen

systémem SimpleShop.cz

Licence pro: petr Ondříšek, petr.ondrisek@seznam.cz
Adresa: karla čapka 119 35709 Habartov, ID prodeje: 9dL3NMZ1WnDXjp62Z5T38az1AvqYR5GbmQoaygrx

CSS

MODERNÍ LAYOUT

Martin Michálek

© Martin Michálek – Vzhůru dolů, 2022
ISBN: 978-80-88253-08-2

Obsah

Předmluva **8**

Autor: Martin Michálek **9**

Pro koho knížka je a jak ji číst? **9**
Jak se v knížce stavím k podpoře v prohlížečích? 12

Kapitola 1: Začínáme s layouty v CSS **14**
Seznamka 15
Grid vs. flex (vs. multicol) 25
Velmi stručná historie layoutů v CSS 29
Moderní a nemoderní layouty 35
Shrnutí první kapitoly 40

Kapitola 2: Layoutové dotazy **41**
Media Queries 42
Container Queries 52
No Queries 63
Shrnutí druhé kapitoly 74

Kapitola 3: Příklady použití **75**
Seznam ukázek 76
Boční panel 77
Hromada lívanců 80
Svatý grál mezi layouty 83
Responzivní fotogalerie 92
Mřížka jako v Bootstrapu 98
Centrování boxu 103
Formulář pro vyhledávání 109
Galerie bez Media Queries 116
Krkavčí technika 121
Shrnutí třetí kapitoly 124

Kapitola 4: Flexbox **125**
Všechny vlastnosti flexboxu 126
Co je dobré o flexboxu vědět 132
Flex-wrap: zalamování položek 133
Flex-direction: směr vyskládání položek 135
Flex-flow: zkratka pro určení směru a zalamování 138

Flex-grow: faktor zvětšování položky 140
Flex-shrink: faktor smršťování položky 143
Flex-basis: velikost položky 146
Flex: nastavení pružnosti položky 153
Shrnutí čtvrté kapitoly 156

Kapitola 5: Grid 157
Úvod a rozcestník 158
Vychytávky gridu 165
Linka, buňka, oblast a další pojmy 170
Mřížka explicitní a implicitní 180
Formátovací kontext mřížky 185
Grid v Internet Exploreru 188
Grid-template-rows/columns: explicitní mřížka 197
Grid-template-areas: pojmenované oblasti 203
Grid-template: zkratka pro mřížku a oblasti 208
Grid-auto-rows/columns: implicitní mřížka 211
Grid-auto-flow: způsob umísťování do mřížky 215
Grid: zkratka pro definici mřížky 219
Grid-row/column: umístění položky do mřížky 222
Grid-area: umístění položky do oblasti 227
Jednotka fr 229
Repeat(), auto-fill, auto-fit 233
Minmax(), fit-content(), max/min-content 236
Subgrid 243
Masonry, zděný layout 247
Shrnutí páté kapitoly 255

Kapitola 6: Vícesloupcový layout 256
Úvod a rozcestník 257
Column-width, column-count a columns: počet sloupců a jejich šířka 262
Column-gap a column-rule: mezery a oddělovače 266
Column-fill: vyvažování výšky sloupců 270
Break-before, break-after a break-inside: zalamování 273
Column-span: překlenutí 277
Shrnutí šesté kapitoly 279

Kapitola 7: Zarovnání boxů 280
Úvod a rozcestník 281
Nutná teorie 286
Justify-items: zarovnání položek na hlavní ose 296

Align-items: zarovnání položek na příčné ose 301
Place-items: zarovnání položek na obou osách 305
Justify-self: zarovnání položky na hlavní ose 307
Align-self: zarovnání položky na příčné ose 312
Place-self: zarovnání položky na obou osách 318
Gap: mezera mezi položkami 320
Justify-content: rozdělení prostoru na hlavní ose 327
Align-content: rozdělení prostoru na příčné ose 332
Place-content: rozdělení prostoru na obou osách 337
Order: změna pořadí položek 340
Shrnutí sedmé kapitoly 344

Kapitola 8: Co byste ještě měli vědět? 345
Internet Explorer (je už mrtvý?) 346
Podpora layoutů v prohlížečích 351
Přístupnost: pozor na pořadí 361
Rychlost: pozor na složitá rozvržení 366
DevTools a CSS layout 372
Shrnutí osmé kapitoly 376

Kapitola 9: Související nástroje a vlastnosti 377
Vlastnost display 378
Logické vlastnosti 386
Jednotky v CSS 395
Autorské vlastnosti (proměnné) 403
Funkce calc() 411
Funkce min(), max() a clamp() 414
Containment a vlastnost contain 420
@supports 425
Autoprefixer 433
Shrnutí deváté kapitoly 437

Na závěr 438
Poděkování 441

Rejstřík 442

Předmluva

Milé čtenářky, milí čtenáři,

kaskádové styly už nejsou tím, čím bývaly. Během těch více než 20 let, kdy CSS sleduji, píšu o nich a přemýšlím nad nimi, se změnily z triviálního jazyka pro popis barevnosti dokumentu k robustnímu systému schopnému deklarovat vzhled dokumentů, aplikací, ale třeba i knížek.

Kdysi bylo možné „céeseska“ umět jakoby mimochodem vedle hlavní odbornosti vývojářů nebo designérů. Dnes tvrdím, že není v silách sebelepších specialistů znát všechny možnosti stylů. Myslím si, že je na čase, aby vznikaly profese zaměřené jen na určitou část CSS, a je tudíž paradoxem, že odborníků na CSS jako celek je jako šafránu.

S tím ale nic nenaděláme. Ostatně ani já na tom nejsem jinak. V době psaní předchozí knížky „Vzhůru do (responzivního) designu“ jsem seděl na jedné židli jako CSS kodér a na druhé jako designér. Dnes už sedím jen na malém růžku obou těchto židlí, až hrozí, že z nich spadnu. Mojí hlavní odborností je totiž zkoumat a zlepšovat rychlost webů.

Jenže CSS mám pořád opravdu hodně rád. Deklarace dnes častěji čtu než píšu, a tudíž se častěji chytám za hlavu při pohledu na cizí než vlastní kód. Tak nevím, jestli se mi ulevilo. Rozhodně ale vidím, že najít správné řešení je v tom dnešním obrovském sortimentu stylů pro nově příchozí někdy sakramentsky těžké.

I proto vznikla tato kniha o nových layoutech v CSS. Aby vám ve složitém světě ukázala cestu, naučila vás triky, které vás možná nenapadly. Je tu proto, abyste se do ní mohli kdykoliv vrátit. Věřím, že pro vás bude jako mapa. A doufám, že vás tahle mapa bude zároveň bavit.

Martin Michálek
v Praze, březen 2022

Autor: Martin Michálek

Martin je webový expert na volné noze s více než dvacetiletou praxí, který se zaměřuje na rychlost webů a CSS. Píše Vzhůru dolů, blog určený webovým vývojářkám a vývojářům. Vede spolek Frontendisti.cz, který se zaměřuje na vzdělávání a propojování komunity. Se ženou a dvěma syny žije v Praze. vzhurudolu.cz/martin

Pro koho knížka je a jak ji číst?

V knížce jsem cílil na webové vývojáře všech úrovní – od začínajících a orientujících se v problematice až po profesionální specialisty na CSS.

Principy CSS layoutů by ideálně mohly být uložené v obecném povědomí také našich netechnických kolegyň a kolegů – z oblasti designu, grafiky, marketingu, produktového managementu a tak dále. Tentokrát je kniha vyloženě technická, ale věřím, že například v prvních dvou kapitolách si i tato skupina najde svoje.

Co musíte umět?

Pro pochopení technických částí budete potřebovat alespoň základní znalost HTML a CSS. Text také vyžaduje hrubý přehled o moderních webařských technologiích a postupech. Pokud je to možné, vždy odkazuji na bližší vysvětlení ve zdrojích na webu nebo přímo v knížce.

Jak knížku číst?

Ideální odpověď by mohla znít: „Slovo od slova, řádek po řádku. Od začátku do konce." Jenže takhle se technické knížky nečtou. A nebudu si hrát na to, že tahle by měla být výjimkou.

V prvních dvou kapitolách čtenářům postupně představuji téma, abychom se ve třetí dostali k vrcholu, který představují příklady použití CSS layoutů.

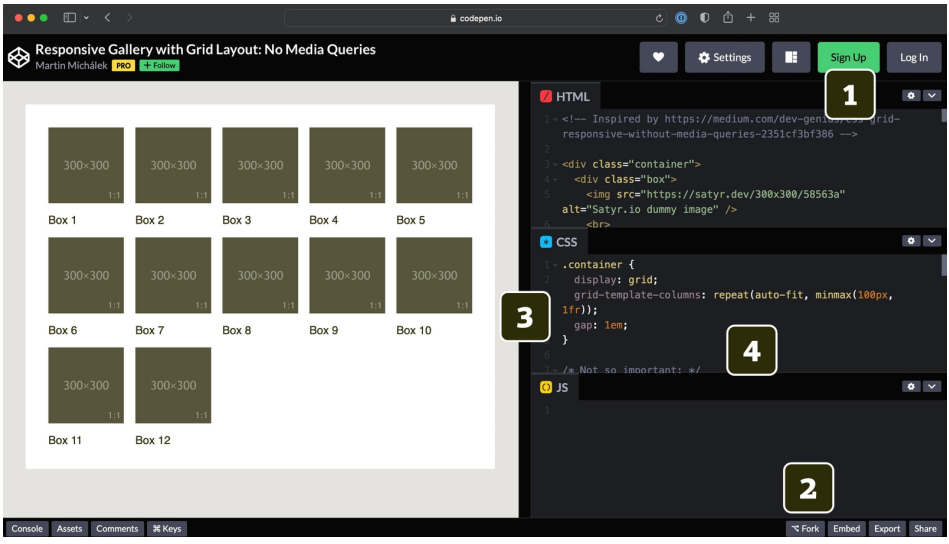
Pátá až devátá kapitola jsou už čistě referenční příručky. Budu moc rád, když si je pročtete – hlavně vy začínající. Ale myslím si, že do nich spíše budete nahlížet, když to budete potřebovat – až narazíte na vlastnost, kterou byste chtěli více pochopit.

Jedno ale doporučím všem – z každé kapitoly si poctivě přečtěte úvod a závěrečný kvíz. V tom druhém se zhruba dozvíte, jak jste na tom se znalostí problematiky. Nakonec třeba uznáte, že jsem netrávil čas zbytečně, když jsem se snažil, aby i referenční kapitoly měly při čtení hlavu a patu.

CodePen: „Talk is cheap, show me the code"

Vy, kteří čtete moje články na Vzhůru dolů, víte, že ctím heslo, že ukázka je víc než tisíc slov. K tomuto účelu používám skvělý nástroj Chrise Coyiera a jeho týmu CodePen.

Mně samotnému spadla čelist, když jsem spočítal ukázky na CodePenech v téhle knize. Je jich na 170.



To je on, našich srdcí šampión. CodePen.

Pokud chcete „peny“ zkoušet, což vám velmi doporučuji, udělejte prosím následující:

1. Zřiďte si na CodePenu účet. Je to zdarma a velmi jednoduché.
2. Moje CodePeny si „forkněte“, čili uložte kopii do vlastního účtu.
3. Responzivitu si emulujte v tomto rozložení a pohybem šoupátka ze strany na stranu.
4. Experimentujte s kódem dle libosti. Snažte se ho vylepšit nebo rozbít.

Takhle se naučíte nejvíc.

Jak se v knížce stavím k podpoře v prohlížečích?

Pokud v následujících řádcích píšu o moderních prohlížečích, mám skoro vždy na mysli tyto tři a jejich poslední verze:

- Chrome od Googlu, což je nyní dominující prohlížeč na mobilech i desktopu.
- Safari od Applu, které se drží hlavně díky uzavřenosti ekosystému, ale s implementací podpory některých modernějších layoutů nespěchá.
- Firefox od Mozilly, což je výborný prohlížeč, ale dnes už bohužel jen s velmi malým tržním podílem v řádu jednotek procent.

Existuje ale ještě minimálně jeden. Pokud jste teď dlouhá léta neleželi v kómatu, nejspíš jej znáte. Až padne jeho jméno, pravděpodobně byste se do kómatu nejraději odebrali.

Internet Explorer, (téměř) mrtvý prohlížeč

Knížku dopisuji v době, kdy už bych skoro nikomu nedoporučil podporovat při startu nového projektu prohlížeč Internet Explorer (MSIE) – jeho tržní podíl je v ČR kolem 1 % a strmě klesá.

Pak tady ale jsou už starší a existující projekty. Většina příkladů v knize je stavěná tak, aby v Exploreru přímo fungovala. U těch ostatních můžete použít techniky, které obecně zmiňuji. MSIE a podpoře ve starších prohlížečích se věnuji i v podkapitolách o jednotlivých vlastnostech. O podpoře gridu v MSIE píšu v páté kapitole, o samotném Exploreru pak v kapitole osmé. Raději se ale zabývám novinkami než zombie prohlížeči.

Spekulativní sázky na novinky

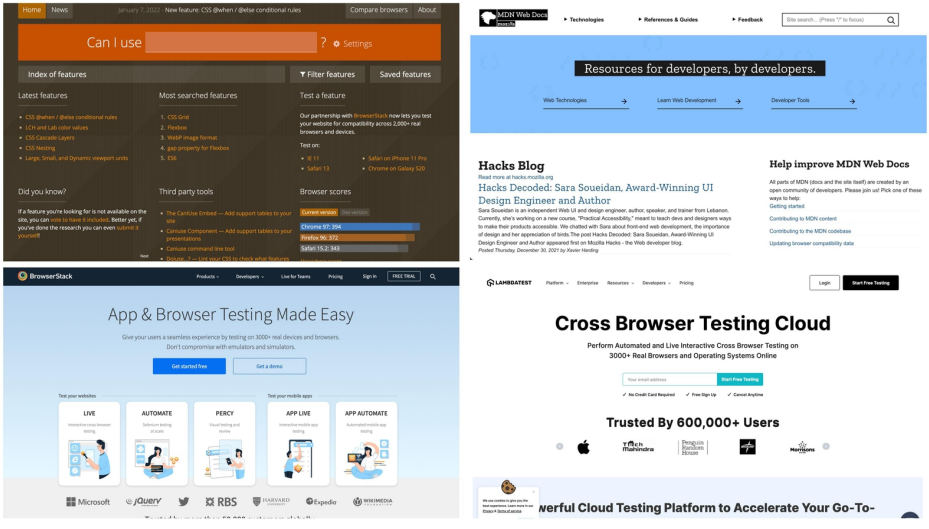
Během psaní knížky se objevila řada novinek, které ještě nemají plnou podporu v moderních prohlížečích, ale já to nevydržel a píšu tady o nich. Vždy jsem docela dlouho zvažoval, zda je zařadit, protože strašně nerad píšu o technologiích, které jsou sice cool, ale v praxi je použít nemůžete.

Takovým Container Queries sice zatím chybí podpora ve Firefoxu, ale pokud to klapne, zcela to změní způsob tvorby webových uživatelských rozhraní.

Dále jsem neodolal například u subgridu, který má zatím implementaci jen ve Firefoxu. Opět se ale zdá, že zájem o implementaci jeví všechny prohlížeče a že i tohle výrazně změní tvorbu layoutů.

Jak hledat podporu a testovat ji

Prakticky každou vlastnost jsem v nějaké době testoval ve všech prohlížečích, ale znáte to – časy se mění a prohlížeče s nimi. Je tedy takřka jisté, že některé informace zde zestárnou.



Tyhle weby mějte po ruce. Kdyby něco.

Doporučuji vám brát podporu v prohlížečích zmíněnou v knize jako orientační a doplnit ji ještě dalšími zdroji:

1. Vždy se dívejte na aktuální stav podpory na webech CanIUse.com nebo MDN (developer.mozilla.org).
2. Testujte to. Buď v prohlížečích, které máte u sebe, nebo v cloudových službách, jako je Browserstack.com nebo LambdaTest.com.

A jsme na konci úvodu. Teď už vám jen přeji, ať se vám knížka líbí a naučíte se mnoho nového!

01

ZAČÍNÁME S LAYOUTY V CSS

V první kapitole se s jednotlivými typy layoutů nejprve seznámíme na jednoduchých příkladech a pak si je představíme v širším kontextu – nejprve historickém a potom ještě v souvislostech s dřívějšími metodami návrhu stránky a nakonec porovnáme mezi sebou.

A teď už: Vzhůru do layoutů v CSS!

Seznamka

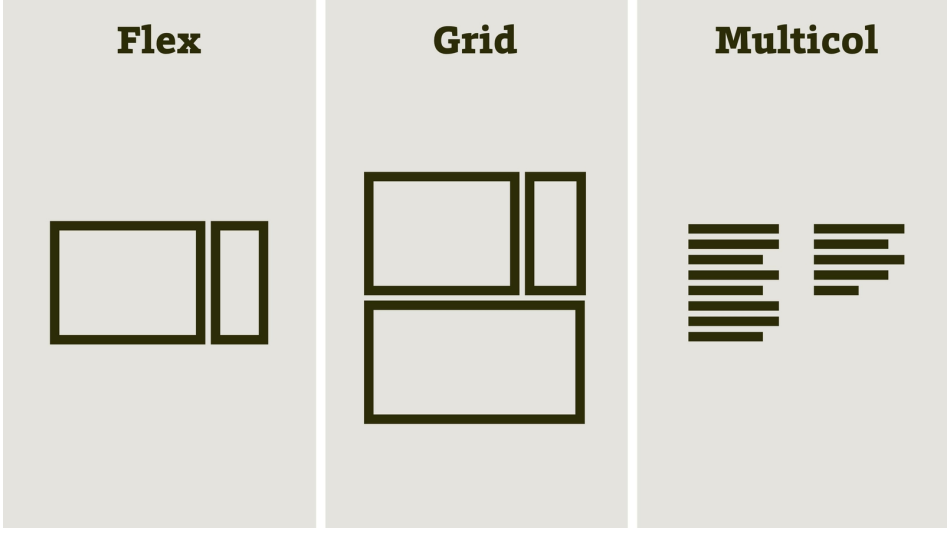
Doba je rychlá a inovace zvýšily tempo řešení situací, které lidem dříve zabíraly velké množství času.

Vezměme třeba seznamování. Když jsem byl v pubertě, bylo potřeba obléct sváteční kostkovanou košili, co nejpodobnější té, jakou nosil Kurt Cobain, vyrazit na vesnickou zábavu, investovat do vstupného a do tekutin, které zajistí uvolněnost a zábavnost, předvést se v tanci na 2 Unlimited a celou dobu hlídat, až budou hrát Nothing Else Matters, a v tu chvíli sprintovat pro dívku snů… Bylo to složité a v mém případě s nejistým výsledkem.

Dnešní seznamovací aplikace jako Tinder jsou s efektivitou úplně jinde, člověk párkrát klikne a je to.

V první kapitole potřebujeme s tématem nějak začít, vytvořit si první náčrt mapy, kterou se vám pokusím během čtení knížky vytvářet v hlavě. A bylo by nejlepší začít svižně. Tohle tedy bude rychlé seznámení s gridem, flexem a vícesloupcovým layoutem, třemi hlavními hrdiny této knížky.

Tohle bude Tinder pro systémy layoutu v CSS.



To jsou oni. Seznamte se.

Zatímco na skutečném Tinderu máte k ruce vystajlované fotky možných partnerek nebo partnerů, CSS layouty jsou sexy asi jako návod na sestavení nábytku z IKEA.

Něco vizuálně poutavého na nich je, ale flex, grid a multicol jsou intelektuálové, kteří na vás mluví řečí symbolů. Žádné obrázky z posiloven, žádné „duckfaces“, žádné výstřihy, a dokonce ani žádné instagramové filtry…

Jsou to od pohledu relativně nudní patroni, takže vám o nich budu muset něco říct:

- Flexbox je skvělý pro jednosměrná rozložení prvků na stránce. Ze všech tří je to největší parťák, kamarádi mu říkají krátce „flex“. Nezkazí žádnou legraci a všichni jej mají rádi. (Specifikace o flexboxu mluví jako o „CSS Flexible Box Layout“ a já pak více ve čtvrté kapitole.)
- Grid výborně zvládá obousměrné layouty. Je mladší než flex a trošku arogantní, protože toho zvládá o dost více. (Gridu se ve specifikaci říká „CSS Grid Layout“, píšu o něm v páté kapitole.)
- Multicol, neboli vícesloupcový layout, je nenápadný introvert. Mnozí si jej ani nevšimnou, protože pasuje jen do několika životních situací týkajících se zalomení textu do sloupců. Když ale takovou řešíte, za vztah s ním budete opravdu vděční. (Plný název podle specifikace je „CSS Multi-column Layout“ a můj detailní rozbor je v šesté kapitole.)

Tak a teď máte nové kamarády, gratuluji vám.

V knížce asi zjistíte, že první pohled někdy klame a to, co vám tady tvrdím, nemusí být vždycky pravda. Od toho jsou ale knížky. Aby vás zbavily povrchního pohledu a ukázaly všechna „ale“.

Box Align, čtvrtý vzadu

Ta paralela s Tinderem samozřejmě trošku skřípe, ale pojďme u ní ještě chvilku zůstat. Znáte to, když jdete s jednou (jedním) na rande a přijdou dvě (dva)? Ona (on) s nejlepší kámoškou (kámošem), pro jistotu?

Přesně tohle by se vám stalo i při seznamování s našimi layouty. Když řekneme flex, grid a multicol, musíme říct také Box Align.

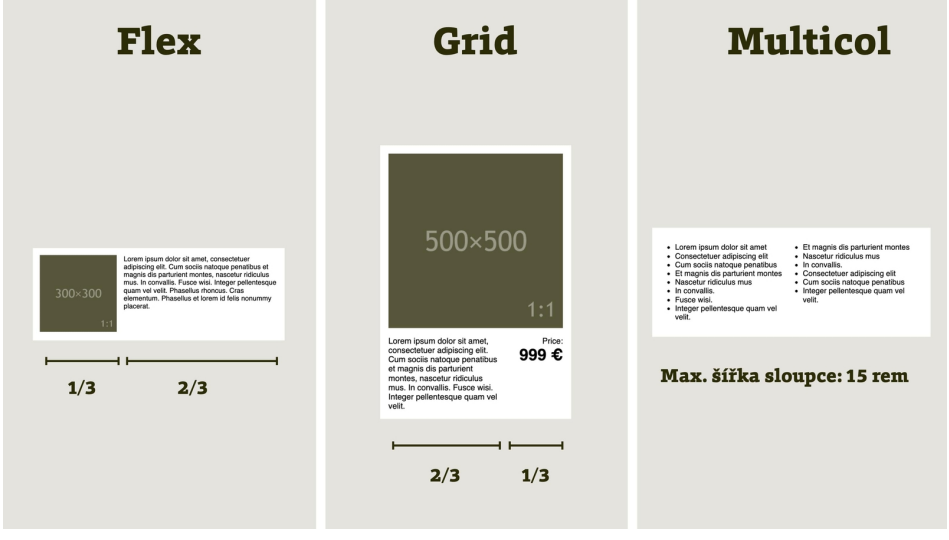
„CSS Box Alignment“, což je plný název podle specifikace, rozebereme v sedmé kapitole. Obstarává celý aparát zarovnávání boxíků v rozvrženích. Bez zarovnání boxů a vlastností, jako je `justify-items` (zarovnání položek na hlavní ose) nebo `align-self` (zarovnání položky na příčné ose), se systémy layoutu (a hlavně flexbox a grid) neobejdou a na rande samy nevyrazí. Ale všeho dočasu. I těmto vlastnostem se budeme detailně věnovat.

Pojďme si je otestovat

Pokud bychom byli při seznamování v reálném životě důslední, chtěli bychom vidět chování partnerky nebo partnera v různých životních situacích. Chtěli bychom co nejrychleji a nejefektivněji zjistit, co je zač.

Tady se proto nabízí spíše paralela s přijímacím pohovorem na pracovní pozici a testy, které zaměstnavatelé svým potenciálním zaměstnancům dávají.

My se teď pokusíme na jednoduchých příkladech ukázat, co je který systém rozvržení zač. Ostatně takhle to budeme dělat v celé knížce.



Tři systémy a tři zadání pro příklad.

Začneme tím nejpopulárnějším parťákem.

Flexbox: příklad s obrázkem a textem

Cílem je dosáhnout rozvržení, kde je na jedné straně obrázek, který zabírá třetinu šířky, a na druhé text. Mezi nimi jsme si ještě vymysleli mezeru o šířce `1rem`, která ale není ve schématu na další stránce vidět.

Začneme asi s HTML, což?

```
<figure class="figure">
  <div class="figure-image">
    <img src="image.jpg" alt="…" width="300" height="300">
  </div>
  <figcaption class="figure-figcaption">
    Lorem ipsum…
  </figcaption>
</figure>
```

Na tom asi nic složitého není, takže pojďme pokračovat.

Používáme zde nejvděčnějšího parťáka z naší trojice, flexbox, a proto nebude složité vytvořit rozvržení:

```
.figure {
  display: flex;
}
```

Už jen tímto zápisem dosáhneme toho, aby se potomci (prvky `figure-image` a `figure-figcaption`) zarovnali vedle sebe.

Pokud je tu někdo, kdo by potřeboval vysvětlit `display` v CSS, nechť zabrousí do poslední kapitoly, kde se jí věnuji do detailu. Ale v tuhle chvíli nám stačí vědět, že zapíná flexboxový layout.

Dále víme, že mezera mezi prvky v rozvržení bude zabírat `1rem`, tedy rozměr základní velikosti písma. Proč zrovna `rem`? O jednotkách něco napíšu ke konci této podkapitoly.

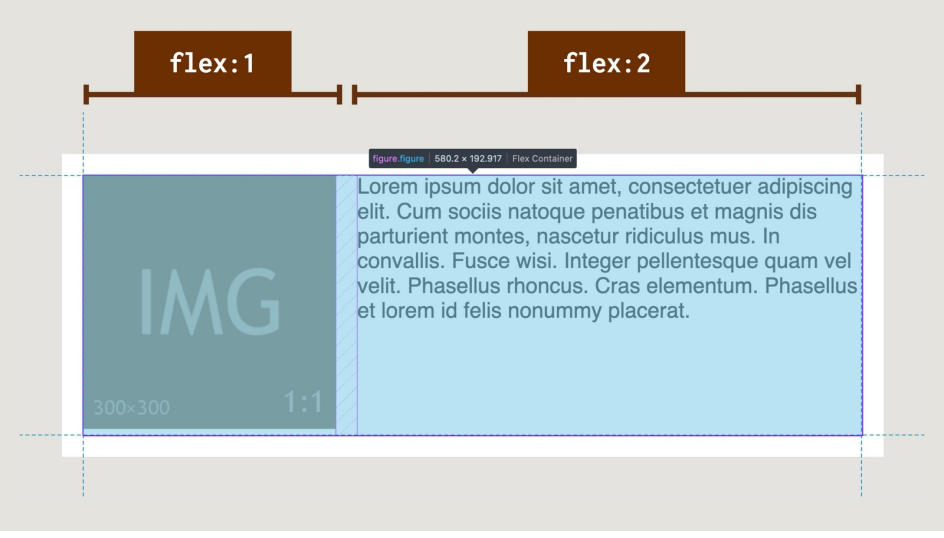
Mezery dosáhneme pomocí vlastnosti `gap`, která v layoutech umí jednoduše udělat mezery:

```
.figure {
  display: flex;
  gap: 1rem;
}
```

Máme rozvržení, ale to ještě nezná své rozměry. Zapíšeme je následovně:

```
.figure-image {
  flex: 1;
}

.figure-figcaption {
  flex: 2;
}
```

Vlastností `flex`, což je zkratka (shorthand) pro nastavení šířky a způsobu rozpínání a smršťování prvku, jsme nastavili podíl na celkové šířce.



Flexbox pod rentgenem. Výsledné rozvržení prohlížené ve vývojářských nástrojích Firefoxu.

Vytvořili jsme rozvržení, kde prvek `figure-image` bude zabírat jednu třetinu šířky (`flex:1`) a prvek `figure-figcaption` vyplní zbylé dvě třetiny (`flex:2`). Tento layout by se dal ještě vylepšovat, například o variantu pro menší displeje. Nicméně jako první ukázka knížky je to myslím dostačující.

Zájemce nyní pošlu na zdrojové kódy ve službě CodePen, což ostatně budu dělat v knížce často.

CodePen: vrdl.in/aedjl

Grid: příklad s kartou produktu

V zadání vidíme, že u gridu jsem vybral rozvržení, které má dva směry – vodorovný i svislý. Obrázek zabírá celou horní polovinu, druhý řádek je rozdělený poměrem 2 : 1. Mezery mezi prvky jsou opět v šířce `1rem`.

Ano, i toto rozlišení bychom mohli udělat flexboxem, respektive dvěma flexboxy. Jenže by to nebylo tak elegantní jako řešení s pomocí gridu.

Pojďme se podívat na zjednodušené HTML:

```
<div class="box">
  <div class="box-image">
    <img src="obrazek.jpg" alt="…">
  </div>
  <div class="box-text">
    Lorem ipsum…
  </div>
  <div class="box-price">
    Price:<br>
    <strong class="price">
      999 €
    </strong>
  </div>
</div>
```

Máme tedy rodiče (`box`) a tři prvky v něm. Jak z nich uděláme kýžený tvar? Poměrně snadno:

```
.box {
  display: grid;
  grid-template-columns: 2fr 1fr;
  gap: 1rem;
}
```

Pomocí `display:grid` „zapneme“ mřížku, ale ještě se (na rozdíl od flexboxu) nic nestane.

Layout vzniká až prostřednictvím vlastnosti `grid-template-columns`, která definuje šablonu pro počet a rozměry sloupců. Zápisem `2fr 1fr` pak rozděluji plochu na sloupečky o šířce dvou třetin a jedné třetiny. Je to podobné, jako jsem to udělal u flexboxu pomocí `flex:2` a `flex:1`.

Vlastnost `gap` zde slouží k vykreslení mezery mezi prvky, v našem případě o šířce `1rem`.

Pokud bychom zůstali u tohoto kódu, kýženého layoutu bychom nedosáhli. Jde o rozvržení 2 × 2, tedy čtyři buňky. V našem HTML ale jako potomky prvku `box` máme jen tři HTML prvky.

Díky „autoplacementu“, vlastnosti gridu, která prvky automaticky umísťuje do mřížky, by byl obrázek v první buňce, text v druhé a cenovka ve třetí. Čtvrtá buňka by zůstala prázdná a naše nadšení pro grid by skončilo uprostřed prvního rande.

Automatické umístění zrušíme tímto kódem:

```
.box-image {
  grid-column: 1 / 3;
}
```

Vlastnost `grid-column` (více v páté kapitole) slouží k výslovnému umístění určitého prvku na konkrétní místo mřížky.

V tomto případě mezi první a třetí linku mřížky. Linka je čára, která mřížku dělí na řádky, sloupce a pak jednotlivé buňky. Pokud si grid přirovnáte k tabulce, pak linky jsou úsečky tvořící její rámeček. (I linkám se věnuji v páté kapitole.)

Nejlépe to uvidíme na následujícím obrázku, který je zároveň doporučením, abyste věnovali pozornost vývojářským nástrojům prohlížečů (více v osmé kapitole). Ty umí být při práci s layouty opravdovými pomocníky.

DevTools Firefoxu (ale i Chromu) krásně ukážou, co je myšleno termínem „linka gridu“.

Nerad bych vám zatajil poslední část kódu. Sice nemá s moderními CSS layouty nic společného, ale bez ní nedosáhneme vzhledu, který vidíte na obrázku. Text v cenovce zarovnáme doprava:

```
.box-price {
  text-align: right;
}
```

I tady máte možnost projít si, vyzkoušet a případně upravit kompletní zdrojový kód.

CodePen: vrdl.in/m71ec

Vícesloupcové rozvržení: příklad s odrážkovým seznamem

Vraťme se opět očima k obrázku se zadáním. V případě multicol layoutu je úkolem pro první rande zajistit, aby se odrážkový seznam skládal do sloupců, které nebudou mít větší šířku než `15rem`.

V HTML máme 15 položek seznamu, ale pro potřeby ukázky v knížce jsem si vystačil se třemi:

```
<div class="box">
  <ul>
    <li>Lorem ipsum dolor sit amet</li>
    <li>Consectetuer adipiscing elit</li>
    <li>Cum sociis natoque penatibus</li>
    …
  </ul>
</div>
```

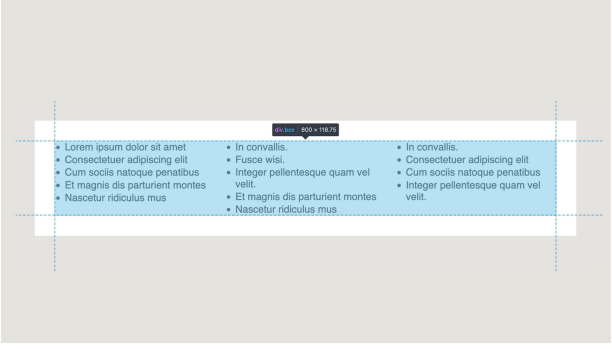
S pomocí flexboxu ani gridu bychom zde neuspěli. Pro rozdělení do sloupců bychom museli mezi `<li>` vkládat prvky `<div>`, které by layout nesly.

Navíc bychom zde s pomocí těchto dvou typů rozvržení nebyli schopni snadno dosáhnout automatického generování vhodného počtu sloupečků. Layout bychom prostě museli doslovně definovat.

Toto je síla CSS Multi-column Layout. Chcete rozvržení o šířce sloupce maximálně `15rem` se šířkou mezery `1rem`? Není nic snazšího:

```
.box {
  columns: 15rem;
  gap: 1rem;
}
```

K vlastnosti `columns` se ještě dostaneme, vlastnost `gap` už trochu znáte.



Výsledek ve Firefoxu. Je ale dobré vědět, že v různých šířkách okna bude různý i počet sloupečků.

Ten vícesloupcový layout vypadá skvěle, viďte? Než jej ale pozvete na druhé rande, měli byste vědět, že se hodí opravdu hlavně jen pro textový obsah, takže na nějaké parády se všemi `div`y světa zde raději zapomeňte.

CodePen: vrdl.in/wucrn

Poznámka k jednotkám

Už v první části knížky jsem použil několik jednotek a myslím, že by bylo vhodné je trošku představit, spolu s jejich kolegyněmi, které budu používat nejčastěji:

- `rem` (root em) je velikost písma v dokumentu. Pokud nenastavíme jinak, jde většinou o `16px`. Používám ji všude, kde se hodí vycházet z velikosti písma.
- `em` (em) je velikost písma v daném prvku. Na rozdíl od `rem` ji může zvětšit kaskáda v CSS, například uvnitř nadpisů. Používám ji pro lokální změnu velikosti.
- `%` (procenta). Používám je pro změnu výchozí velikosti v dokumentu a občas pro šířky či výšky v layoutu.
- `px` (CSS pixely). Používám je hlavně u dekorací, jako je šířka rámečku.

V Media Queries v knížce používám jednotně `em`, ale v praxi je možné použít také `rem`, a pokud vám nevadí velmi specifické problémy se zvětšováním stránky v Safari, lze využít rovněž `px`.

Velice často zde také budu používat bezjednotkové hodnoty, které určují podíl prvku na rozměrech celku. Jde o čísla ve vlastnosti `flex`, zde již použité, hodnoty s jednotkou `fr` v gridu. Ale nebudeme předbíhat.

Více o používání jednotek v CSS píšu v poslední, přílohové kapitole.

Prostřednictvím Tinderu a prvního rande jsme se tedy seznámili se třemi hlavními protagonisty knížky a jejich nejlepším kamarádem. Ještě se nám ale bude hodit lépe poznat dva nejžhavější kandidáty.

Grid vs. flex (vs. multicol)

Nyní se zaměříme na jednotlivé systémy rozvržení a povíme si více o rozdílech mezi nimi. Začneme vícesloupcovým layoutem, který se z těch tří trochu vyjímá.

Multicolumn layout, expert na vícesloupcovou sazbu textu

Tohle bude rychlé. Už totiž víte, že multicol je určený pro vícesloupcovou sazbu textu, nikoliv pro rozvržení rozhraní.

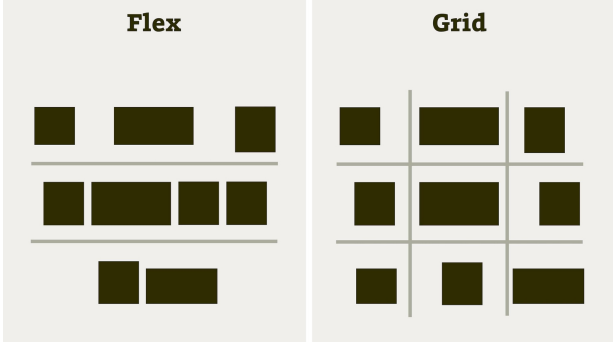
Jasně – rozvržení je možné tvořit i pomocí vlastnosti `column` a podobných, je možné jej omezeně využít pro tvorbu „No Queries“ layoutů, ale je dobré si uvědomit, že tím ohýbáme nástroj pro potřeby řešení.

Grid vs. flexbox

Tyto dva systémy rozvržení tvoří jádro knížky, a proto se jim budeme věnovat s patřičnou péčí. Občas se setkávám s názorem, že grid je novější, a tak nahrazuje starší flexbox. Nenahrazuje. Potřebujete obojí. A obojí se naučte. Jsou mezi nimi však významné rozdíly. Podívejme se na ně.

Jednorozměrný flexbox, vícerozměrný grid

Grid navrhli autoři pro dvourozměrné layouty – po svislé i vodorovné ose. Flexbox se více hodí na rozvržení jednosměrná, ať už svislá nebo vodorovná.



Flexbox zná jen jeden směr. Víceřádkový layout už pak například nezarovnáte do sloupců. Grid je mřížka. Dvourozměrná mřížka.

Jakmile je váš layout víceřádkový, typu mřížka, a nevystačíte si s pouhým zalomením bez definice pravidel layoutu druhé řádky, je pravděpodobné, že potřebujete grid.

U flexboxu kraluje obsah, u gridu zase nastavený layout

Grid je zaměřenější na layout, kdy se obsah musí vždy přizpůsobit definované mřížce. Layout je král – a obsah, tedy text, obrázky a tak dál, se mu musí přizpůsobit. Této filozofii se v odborné hantýrce říká „grid in", takže „z mřížky směrem dovnitř".

Flexbox je naopak navržený pro situace, kdy se layout musí přizpůsobit obsahu. Výhodou je, že nemusíte často nastavovat parametry layoutu a díky rozměrům obsahu to nějak dopadne. Tento přístup k vykreslování layoutu naleznete pod pojmem „content out", tedy „z obsahu směrem ven".

Vyplývá to trochu i z jejich podstaty na úrovni kódu:

- Když napíšete `display:flexbox`, potomci se poslušně seskupí do vodorovného rozvržení. Pro definici layoutu použijí prvky své vlastní rozměry.
- Po napsání `display:grid` se nestane vůbec nic, dokud nedefinujete rozvržení pomocí vlastností řady `grid-template`. Tady vládne autor kódu a jeho mřížka, prvky stránky se musí přizpůsobit.

Pokud tedy chcete umístit pár prvků vedle sebe a není nutné jim definovat rozměry, volte flexbox. Jestli položkám flexboxu složitě nastavujete pravidla chování rozměrů, pak asi potřebujete grid.

Samozřejmě, že i z tohoto pravidla existuje řada výjimek. Mřížka například umí layout tvořený z obsahu, ale musíte si to nastavit.

Flexbox slouží více layoutu komponent, grid zase layoutu celé stránky

Tím, že je flexbox jednosměrný, bude se vám častěji hodit na zarovnávání „věcí" vedle sebe nebo pod sebe. Užijete jej pro záložkové i globální navigace nebo například zarovnání jednořádkových formulářů. Více jej oceníte u menších komponent uvnitř stránky.

Existují také studie, které zjistily, že flexbox se při použití obrovského množství elementů chová při vykreslování celé stránky o trochu líněji, jak na okraj zmiňuji v osmé kapitole. Ne že by to bylo zvlášť důležité, ale opět to ukazuje na podstatu flexboxu – vycházet z rozměrů vykreslených prvků.

Náš milý grid je obousměrný, takže je vhodné s ním rozvrhnout například složitější formuláře, výpis produktů v kategorii e-shopu nebo celou stránku. Grid je příhodnější pro layout větších celků.

Toto pravidlo samozřejmě neberte dogmaticky. Osobně, když se rozhoduji, jestli použít grid, nebo flexbox, kladu si otázku:

> **Je možné přes rozvržení na některých šířkách okna položit neviditelnou vícerozměrnou mřížku?**

Představa mřížky je zde zásadní. A úplně nejlepší je, když mřížku v podkladech vizualizuje designér nebo grafik.

Kompatibilita s Internet Explorerem je silná stránka flexboxu

Flexbox ja takřka plně podporován prohlížeči Internet Explorer verze 10 i 11. Ty už jsou dneska víceméně na odpis, hlavně verze 10. Ale některé weby a webové aplikace mají stále slušnou návštěvnost z MSIE jedenácté verze. Pak je potřeba zvažovat, zda je možné grid použít, například s pomocí náhradních řešení, o kterých zde budu také psát. Grid v těchto prohlížečích podporu má, jenže s velkým „ale“. Více se tomu věnuji v osmé kapitole.

Pokud nutně potřebujete obsloužit i uživatele těchto prohlížečů a není možné zajistit fallback, tedy náhradní řešení, pak může flexbox vaše problémy eliminovat.

Společné vlastnosti

Je dobré si uvědomit, že velkou část vlastností mají grid s flexboxem společnou nebo podobnou:

- Všechny vlastnosti pro zarovnávání ve svislém a vodorovném směru.
- Flexibilní jednotky (grid nabízí `fr`, flexbox pak čísla ve vlastnosti `flex`).

Našli bychom toho více. Platí ale, že grid tyhle vlastnosti pokrývá více ze šířky a více do hloubky a je celkově robustnější. Flexbox je takový menší brácha, specializovaný na jeden typ layoutu.

Srovnání flexboxu a gridu

Abychom si v tom udělali pořádek i vizuálně, připravil jsem si pro vás takovou pěknou tabulku. Berte to tak, že to velmi zjednodušuji:

Vlastnost	Flexbox	Grid
Jednorozměrný layout	+	+
Dvourozměrný layout		+
Layout z obsahu	+	?
Layout z mřížky		+
Kompatibilita v MSIE	+	?

Tabulka netabulka… osobně vám doporučuji naučit se flexbox i grid.

Abyste oba dobře poznali, budete potřebovat poznat také jejich předky. Začneme pravěkou historií a pak se zastavíme u nedávné minulosti.

Velmi stručná historie layoutů v CSS

Pokud se těšíte na psaní kódu, tuhle část můžete v klidu přeskočit. Ale věřte mi – pokud bych si myslel, že vás bude nudit, do knížky bych ji nedával.

Historie rozvržení v CSS je totiž tak trochu historií CSS, a protože tu jsem napjatě sledoval úplně od začátku, vidím v následujícím příběhu pár zajímavých a vtipných momentů.

Slíbil jsem stručnost, takže to vezměme v bodech.

1994: „Je mi líto, ale jste v háji."

Možnosti stylování dokumentů jsou mizerné. Mimo jiné to výborně vyjadřuje v e-mailovém fóru WWW-Talk Marc Andreessen, spoluzakladatel Netscapu a pozdější člen představenstva Facebooku. Když prý po něm lidé chtějí, aby jejich webové dokumenty vypadaly podobným způsobem, jak to jde zařídit v sázecím programu LaTeX (nebo alespoň jako ve Wordu), všem prý odpovídá: „Sorry, you're screwed."

1996: FRAME

Jako nedostatečné vnímají možnosti rozvržení v CSS už samotní zakladatelné kaskádových stylů, Bert Bos a Håkon Wium Lie, a proto přicházejí v tehdejší době s převratnými rámci. Prvek `<FRAME>` umožňuje rozdělit stránku na samostatné jednotky, které nejsou nic jiného než jen další stránky. První layoutová revoluce je tady.

1998: MULTICOL

Andreessen se snaží implementovat layout do svého prohlížeče Netscape Navigator, a tak verze 3 přichází s tímto zápisem:

```
<MULTICOL COLS="2">
  <P>The Hypertext Markup Language (HTML) is a markup language… </P>
</MULTICOL>
```

Ano, jde o přímého předchůdce vícesloupcového layoutu v CSS.

Ano, tehdy se opravdu i vzhled a layout zapisoval do HTML.

Ano, opravdu se značky zapisovaly VERZÁLKAMI.

Ne. Opravdu o tom už nechci dále mluvit.

V těch časech stavím svůj první větší web – fanouškovské stránky kapely U2 na tehdy velmi známém serveru MusicHall. Jsem hrdý, že pro rozložení stránky nepoužívám `<frame>`, už zastaralou metodu. Kód mého webu je totiž stavěný na prvku `<table>`.

1998: TABLE

Někdy v této době přišel bezejmenný webař na to, že když z tabulky odstraní rámečky, vznikne docela dobrá struktura, která může nést i složitější rozvržení stránek nebo jednotlivých komponent uvnitř. Někdy po roce 2000 začne komunitou kolovat následující vtip:

– „Tr td td td tr td td td… – co je to?“
– „Vlak přijíždí do stanice?“
– „Ne, to webař tvoří rozvržení stránky.“

Kód, který takto vznikal, mě dodnes chodí v noci strašit. No však se sami podívejte:

```
<table width="100%" border="0">
  <tr>
    <td colspan="2" bgcolor="#b5dcb3">
      <h1>Ahoj, jsem web</h1>
    </td>
  </tr>
  <tr valign="top">
    <td bgcolor="#aaa" width="50">
      Ahoj, jsem navigace.
    </td>
    <td bgcolor="#eee" width="100" height="200">
      Ahoj, jsem hlavní obsah.
    </td>
  </tr>
</table>
```

Doufám, že až někdo tomuto neznámému webaři postaví pomník, bude text na něm vysázený v tabulkovém layoutu.

1999: Multi-column Layout

Håkon Lie a mnozí další přicházejí s prvními drafty specifikace pro vícesloupcový layout.

Když si ji prohlédnete, zjistíte, že se zápis v kódu od toho dnešního zase tak moc neliší.

```
BODY {
  column-number: 3
}
```

Klobouk dolů, zvládli to specifikovat skoro na první dobrou.

2002: Éra floatů

Už nějakou dobu dělám weby za peníze.

Na stránkách ostravské cestovky Rywal jsem nepoužil zastaralý prvek `<table>`, ale CSS a vlastnost `float`. Jsem na sebe pyšný.

2005: Něco jako grid poprvé

Bert Bos přichází se specifikací „CSS3 Advanced Layout Module“. Vypadá to trochu jako kombinace rámců (`<frame>`) s absolutním pozicováním:

```
dl {
  display: "ab"
           "cd"
}

#sym1 { position: a }
#lab1 { position: b }
#sym2 { position: c }
#lab2 { position: d }
```

Ale už je tady trochu vidět směr uvažování v šablonách. Vidíte prapředka vlastnosti `grid-template-areas`.

Následuje mnoho pokusů o návrh rozumného systému layoutu v CSS, ale výrobci prohlížečů je smetávají ze stolu jako „příliš složité na implementaci“.

No – a když není žádná implementace v prohlížečích, návrhy specifikací se vynořují a následně zapadají prachem.

Webaři mezitím skřípou zuby u každého `float`. A pak si na dlouhou dobu zvyknou.

2009: Na scénu přichází flexbox

Vzniká první specifikace flexboxu:

```
.container {
  display: box;
  width: 300px;
}

.box-1 {
  box-flex: 1.0;
  width: 100px;
}

.button2 {
  box-flex: 2.0;
  width: 140px;
}
```

Tahle „nultá verze“ specifikace je založena na technologii XUL, kterou v té době používá Firefox k vytváření uživatelského rozhraní prohlížeče nebo také rozšíření.

V témže roce vzniká „Candidate Recommendation“ pro vícesloupcový layout. Prohlížeče jej začínají implementovat.

2011: Microsoft implementuje grid

V Microsoftu se dějí věci. Plánují zaříznout Silverlight, zjednodušeně řečeno vlastní napodobeninu Flashe, a více šlápnout do webových technologií. Jde jim o využití HTML a CSS pro vlastní produkty, ale jejich vývojáři jsou nešťastní z nedostatku rozumného layoutování v CSS, oproti tomu, na co byli zvyklí v Silverlightu.

To by člověk nečekal, na jak podivných místech najde inspirační zdroje dnešního moderního CSS. Děkovné dopisy prosím posílejte do Microsoftu.

Nejvíce se tato frustrace projeví u šéfa jednoho z redmondských týmů zaměřených na návrh uživatelského rozhraní, Phila Cuppa. Dojde to tak daleko, že se svým týmem zařídí implementaci systému pro layout do Internet Exploreru 10.

Ta věc se jmenuje „Grid Layout".

V témže roce přijde Tab Atkins, další známá postava standardů kolem CSS, a přepíše celou specifikaci flexboxu. Jeho hlavním cílem bylo pokusit se odstranit závislost na všech těch šílených háčcích s `float` nebo `table`, které jsme museli jako weboví vývojáři umět.

2012: Přelom. Prohlížeče implementují flexbox, Microsoft předává grid do W3C

Cupp prezentuje nový systém rozvržení standardizačnímu konsorciu W3C. Začne se o tom hodně mluvit a další nadšenci (jako Rachel Andrew, kterou jste v Praze mohli potkat na konferenci WebExpo 2016 nebo XML Prague o čtyři roky později) to šíří do komunity webových vývojářů. Odtud už vede přímá cesta k dnešnímu CSS gridu.

V čem byl ale tento pokus o implementaci rozvržení typu mřížky jiný než všechny předchozí? Je to jednoduché – Phil Cupp a jeho lidé zařídili implementaci v prohlížeči.

Nejprve implementace, pak specifikace. Přesně tím postupem, který je dnes vyčítán Googlu. Historie má tendenci se opakovat.

Ale v kontextu CSS gridu se chce říct: hlavně, že to vyšlo.

Z flexboxu se stává „Candidate Recommendation“, což znamená, že specifikace je stabilní. Prohlížeče ji začínají implementovat.

2017: Nová a lepší specifikace gridu

Vzniká specifikace „CSS Grid Layout Module Level 1“.

Lidé z W3C ale původní specifikaci od Microsoftu radikálně předělali. Přidali například možnosti automatického umísťování do mřížky a rozšířili ji o lepší možnosti responzivity.

Pár věcí si vzali také ze staré specifikace od Berta Bose. Například definici layoutu pomocí vlastnosti `grid-template-areas`:

```
grid-template-areas:
  "heading heading"
  "col1 col2";
```

„Je to jako ASCII art pro vytváření šablony,“ řekla prý Elika Etemad, editorka pro modul CSS Grid Layout Module ve W3C, což sedí.

S výsledkem se právě teď můžete podrobně seznámit na následujících stránkách.

Myslím, že se po dočtení knížky shodneme, že je to takové malé zemětřesení pro vývoj webů. Zemětřesení, které zboří staré základy a vytvoří nové, na kterých se začne stavět… co vlastně?

Pokud byste se do historie chtěli ponořit více, doporučím vám tyto dva články:

- „The Story of CSS grid, from Its Creators“ od Aarona Gustafsona. vrdl.in/vdghist
- „A Look Back at the History of CSS“ od Jaye Hoffmanna. vrdl.in/csshist

My se od historie specifikací CSS layoutů přesuneme k dnešním možnostem tvorby rozvržení. Ano, myslím ty, které nejsou předmětem této knížky. Ty, o nichž se už dnes slušní frontendisti raději nebaví.

Moderní a nemoderní layouty

„Musíš se teď odnaučit, co tě dřív učili," říká mistr Yoda Lukovi Skywalkerovi v páté epizodě kultovních Star Wars.

Neexistuje lepší citace pro začátek textu, který má zkušenější kodérky a kodéry přesvědčit, aby zkusili zapomenout na vše, co dosud o layoutu v CSS věděli, a na uvolněné místo si v mysli dosadili nové systémy rozvržení.

Pokud snad s „webděláním" začínáte – a pojmy jako „tabulkový layout", `float` nebo Bootstrap vám nic neříkají –, pak nezbývá než vám závidět. Tenhle text zase tak moc číst nemusíte.

Layout versus „pseudolayout"

Za těch více než dvacet let, co dělám weby, mi bylo dopřáno realizovat rozvržení stránky i jejich komponent opravdu mnoha způsoby.

Z dnešního pohledu zní až krutě, že ani jeden z nich pro layout určený nebyl, takže jsem s nimi vlastně bojoval zbytečně:

- Rámy `<frame>` sice dělí stránku do menších částí, ale jinak nemají s implementací jejího rozvržení nic společného.
- Tabulky `<table>` jsou určené pro vykreslení tabulkového obsahu. Dělat v nich rozvržení? Strašný hack určený dnes už jen expertům na kódování newsletterů.
- Plovoucí prvky (vlastnosti `float` a `clear`) vznikly pro zajištění obtékání elementů textovým obsahem. Layout? Dnes už prosím ne.
- Metoda „inline rozvržení" (pomocí vlastnosti `display:inline-block`) přišla s možností vkládat blokové prvky do řádků textů, ale layout stránek či komponent není její účel, na což ukazuje nutnost vypořádat se s přirozenými mezerami, které vznikají tímto „layoutem z textu".
- Techniky využívající pozicování lámou přes koleno vlastnost `position`, která slouží k umístění prvků mimo přirozený proud stránky.

Jak vidíme, technik upotřebitelných pro tvorbu layoutu máme v CSS víc než dost. To ale neznamená, že bychom je pro tvorbu rozvržení *měli* používat.

Každá z uvedených metod totiž pravděpodobně vyřeší jen některé z vašich problémů a ještě k tomu do kódu zavede problémy nové.

Paličku na maso je totiž v případě nouze možné využít také k zatloukání hřebíků. Jen si musíte dát pozor na prsty, hřebíky i stěnu, protože ten nástroj vymysleli na úplně jinou práci.

Pojďme na tyto techniky prostě zapomenout. Máme kladívko, takže paličkou dále už jen naklepávejme maso.

Jak říkaly naše babičky: „Paličkou na maso hřebíky nezatloukej, kladívkem neklepej maso a floaty nedělej layout v CSS.“

Nevýhodou „pseudolayoutů“ je jejich špatná zapamatovatelnost. Tabulkový layout je dodnes relativně populární právě proto, že je velmi logický.

Ale vysvětlujte začátečníkovi, jak udělat rozvržení pomocí vlastnosti `float`. Musíte vždy dodat „clearování“ nějaký způsob, jak plovoucí prvky resetovat. Není to prostě úplně jednoduché a občas se to chová divně.

„Pseudolayout“ se špatně vysvětluje a špatně používá, proto lidé hledali způsob, jak si často se opakující nepříjemnosti zjednodušit. Řešení této situace my v IT říkáme „framework“.

Layout versus CSS frameworky

V druhé polovině první dekády 21. století (sic!) začaly vznikat nástroje, které měly práci s CSS usnadňovat. A taky usnadňovaly.

Vznik CSS frameworků podpořily problémy v kompatibilitě napříč prohlížeči, jež vznikaly i při tvorbě na pohled jednoduchých rozvržení.

Raný webdesign: Jak to vlastně dělali sazeči knih a novin?

Někdy mezi roky 2005 až 2010 si webaři začali ve velkém všímat typografických principů ze sazby knih, novin a časopisů.

Prvním frameworkem, který jsem v té době pro tyto účely využíval, byl Blueprint CSS. Zavedl mřížku o 24 sloupcích, takže pokud jste chtěli layout rozdělit na dvě poloviny, vypadalo to následovně:

```
<div class="container">
  <div class="span-12">První polovina</div>
  <div class="span-12">Druhá polovina</div>
</div>
```

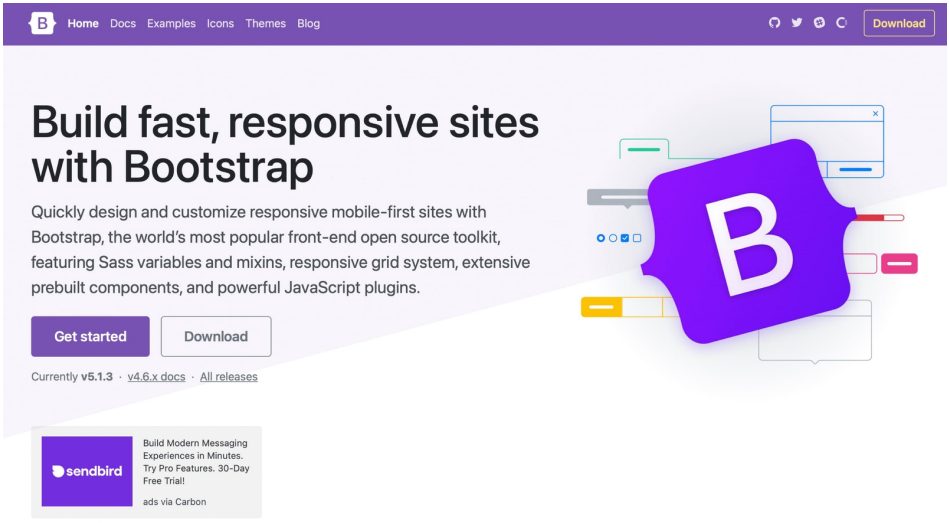
V té době šlo o zjevení. Dodnes si dobře vybavuji radost z krásných typografických principů, ale také z výrazně rychlejšího psaní kódu, protože jsem nemusel tak často přepínat mezi HTML a CSS.

Následovaly frameworky jako 960 grid System, YUI grids, YAML… a v roce 2011 představil Twitter dosud nejkomplexnější knihovnu pro tvorbu uživatelských rozhraní – Bootstrap.

Éra Bootstrapu

Bootstrap je skvělý. Kromě základních typografických principů a mřížky obsahuje také hotové komponenty pro tvorbu uživatelských rozhraní a dnes už i prostředky pro systematickou tvorbu komponent vlastních, například v podobě „utilit“.

Pro vývojáře, kteří musí tvořit rozhraní a nemají po ruce designéra a často ani frontendového kodéra, představuje Bootstrap (a z něj vycházející frameworky jako Foundation nebo Semantic UI) hotový zázrak. Všechno, co potřebují, napíšou v HTML a framework se nějak postará o zbytek.



Bootstrap je skvělý nástroj. Občas se ale používá pro nesprávné účely.

Zároveň zde ale vznikl problém s nadužíváním těchto nástrojů. Jako konzultant rychlosti webu se vývojářů často ptám, proč na webech používají Bootstrap. „Pro mřížku," zní častá odpověď.

Pro mřížku tak řada webů stahuje zhruba 25 kilobajtů zbytečného kódu. To nemusí vypadat jako vysoké číslo, ale uvědomme si, že jde o pětadvacet „kilo" stylů, které nemohou být ze své povahy asynchronní a bez kterých nemůže prohlížeč vykreslit ani řádku z chystané stránky.

Když se tento laxní přístup k rychlosti nakombinuje s dalšími „ne-optimalizacemi", na vykreslení stránky uživatel čeká… a čeká… a čeká.

CSS grid nás nezbavuje nutnosti přepínat pozornost z HTML do CSS, ale řeší problémy s nefunkčními základními layouty a radikálně zjednodušuje znalosti, které musíte o psaní CSS pro rozvržení stránky mít.

V knížce se proto pokusíme nahradit některé základní scénáře použití mřížky z Bootstrapu pomocí CSS gridu.

Tímto tedy oficiálně zahajuji snahu vás přesvědčit, abyste nepoužívali složité frameworky a staré postupy na plnění jednoduchých cílů.

Možná, že to nebude snadné, ale společně to zvládneme. Ostatně – celá citace rozhovoru z úvodu této podkapitoly má znít následovně:

Luke Skywalker: „Umět pohnout kamenem je jedna věc, ale tohle je něco jiného."
Mistr Yoda: „Ne, není to jiné. Jiné je to v tvé hlavě. Musíš se teď odnaučit, co tě dřív učili."
Luke Skywalker: „Dobře, já to teda zkusím."
Mistr Yoda: „Ne, nezkusíš. Uděláš to, nebo ne. Už žádné zkusím!"

V dalších kapitolách budu trochu jako mistr Yoda. Ale hodný mistr Yoda.

Shrnutí první kapitoly

Nejspíš jste si všimli změny formátování (v barevné verzi má podobu oranžového proužku vlevo). V těchto částech knihy vás vybízím k aktivitě, ať už ve formě kvízu nebo možnosti zkusit si nakódovat některý příklad.

Shrnutí všech kapitol jsem pro vás připravil ve formě kvízu. Testy jsou skvělé v tom, že čtenáře nutí přemýšlet a lovit v paměti. Odměnou za tu trochu námahy by vám mělo být trvalejší uložení důležitých informací z knížky v hlavě.

Řešení testů najdete hned pod nimi a zároveň slouží jako shrnutí kapitoly, pokud byste snad nechtěli testovat svou znalost problematiky. První test je rozehřívací.

1. **K čemu slouží CSS Multi-column Layout?**
   a) ke flexibilnímu rozložení položek vedle sebe
   b) k automatické vícesloupcové sazbě textu
   c) k rozložení do tvaru mřížky
2. **Co udělá zápis display:flex?**
   a) zobrazí prvek ve stránce jako úhlovou brusku, flexu
   b) neudělá nic, je potřeba dalších vlastností jako `flex-template`
   c) vykreslí všechny potomky prvku vedle sebe
3. **Technologie, jejímuž zániku vděčíme za vznik CSS gridu, je:**
   a) Macromedia Flash
   b) Netscape Navigator
   c) Microsoft Silverlight

Řešení: 1. b), 2. c), 3. c).

02

LAYOUTOVÉ DOTAZY

Příklady, které jsem vám dosud uváděl, zamlčovaly velmi podstatný fakt. Na naše weby se uživatelé dívají z plejády různých velikostí obrazovek.

Už dávno neplatí, že lidé konzumují obsah webů z pohodlí domova a svého počítače. V roce 2010, když Ethan Marcotte napsal svůj slavný článek „Responsive Web Design“, ještě patřili uživatelé mobilů mezi návštěvníky našich webů mezi exotickou a málo vídanou zvířenu.

Dnes je situace jiná. Návštěvníci z mobilů jsou na webech ve většině, leckde dokonce drtivé. Navíc platí, že kromě nástupu malých displejů pozorujeme zvětšování těch velkých a obecně stále větší různorodost v této oblasti.

Pro tvorbu layoutů tedy potřebujeme podmínky, díky nimž budeme moci layout upravovat pro různě velké obrazovky. Klasikou jsou zde Media Queries, ale představíme zde novou možnost, Container Queries, a pak speciální záležitost – No Queries, přizpůsobivé layouty, které nepotřebují definované podmínky.

Media Queries

Media Queries, dotazy na média, jsou tady s námi od doby, kdy webdesignovému světu zavládly mobily. Jde o základní stavební kámen responzivního designu, takže při stavění CSS layoutů bez nich nepostavíme ani boudu pro psa.

Dovolím si zde shrnout jejich úplné základy, pro případ, že v Media Queries plavete. Vy zkušenější, kteří nestojíte o opakování tématu, můžete úvodní část s čistým svědomím přeskočit a rovnou si najít nadpis „Limity Media Queries“.

Media Queries jsou podmínky, které umožňují aplikovat různá CSS pravidla v různých technických kontextech, nejen tedy v těch týkajících se rozměrů obrazovky.

Příklad pro začátek

Vezměme toto triviální HTML:

```
<h1>This is heading</h1>
```

V CSS pak nadpisu nastavíme velikost písma a podmínku:

```
h1 { font-size: 2em }

@media (max-width: 40em) {
  h1 { font-size: 1.5em }
}
```

Možná i začátečníci už poznali, co se zde děje:

- Klíčové slovo `@media` oznamuje, že následuje dotaz na média.
- Podmínka `(max-width:40em)` říká, že vložený kód se má aplikovat jen pro šířky okna do `40em`, což je zde obvykle 640 pixelů (40 × 16, při výchozí velikosti písma v dokumentu `16px`, není-li nastaveno jinak).
- V podmínce nadpisu nastavujeme jinou velikost písma `font-size:1.5em`. Pokud jste si to už nespočítali, půjde o 24 pixelů (16 × 1,5).

CodePen: vrdl.in/p1mrt

Různé typy dotazů

Kromě maximální šířky se samozřejmě můžeme ptát i na minimum:

```
@media (min-width: 40em) {
  /* Aplikuje se pro šířku viewportu od 40em */
}
```

Můžeme se ptát na výšku:

```
@media (max-height: 50em) {
  /* Aplikuje se pro výšku viewportu do 800px */
}
```

(Pokud vás zajímá, proč zde používám jednotku `em`, podívejte se do přílohové kapitoly na text o CSS jednotkách.)

Můžeme také detekovat typ média:

```
@media screen {
  /* Aplikuje se jen pro obrazovky,
     ne například v tisku */
}
```

Můžeme podmínky kombinovat pomocí logického „a zároveň“:

```
@media screen
  and (min-width: 30em)
  and (max-width: 40em) {
  /* Aplikuje se jen na viewport
     obrazovky mezi 30em a 40em */
}
```

Můžeme aplikovat logické „nebo“, a to pomocí čárky:

```
@media screen and (max-width: 40em),
  print {
  /* Aplikuje se na obrazovky od šířky
     viewportu 40em nebo v tisku */
}
```

Můžeme detekovat poměr stran viewportu:

```
@media (aspect-ratio: 16/9) {
  /* Aplikuje se v poměru stran 16 : 9 */
}
```

Zjišťovat orientaci viewportu:

```
@media (orientation: portrait) {
  /* „Na výšku“, šířka je užší než výška */
}
```

Detekovat hustotu pixelů na obrazovce:

```
@media (min-resolution: 2dppx) {
  /* Aplikuje se na vysokokapacitní displeje typu „retina“ */
}
```

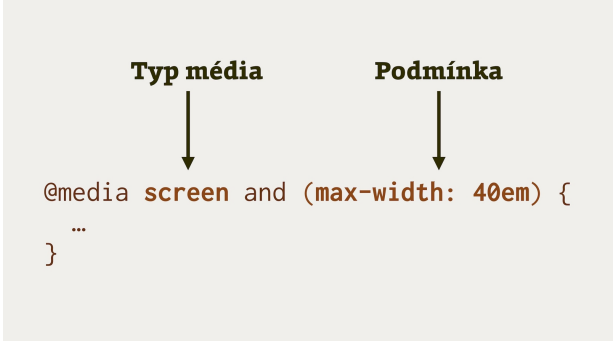
Padly tady výrazy, kterým někteří nemusí rozumět – těm opět nabízím vysvětlení v textech na Vzhůru dolů:

- „Viewport“ je část obrazovky, ve které prohlížeč vykresluje stránku. vrdl.in/viewp
- „Retina“ a vysokokapacitní displeje do webdesignu zavedly přepočet hardwarových pixelů na „CSS pixely“. vrdl.in/csspix

Ještě se čistě obecně pozastavme u složení jednotlivých částí každého dotazu na médium.

Anatomie Media Query

Dotaz na médium (anglicky *media query*) se skládá z typu média (*media type*, výchozí je `all`) a podmínky obsahující vlastnosti média (*media features*) s hodnotou nebo rozmezím hodnot.



Anatomie Media Query.

Chcete vědět více?

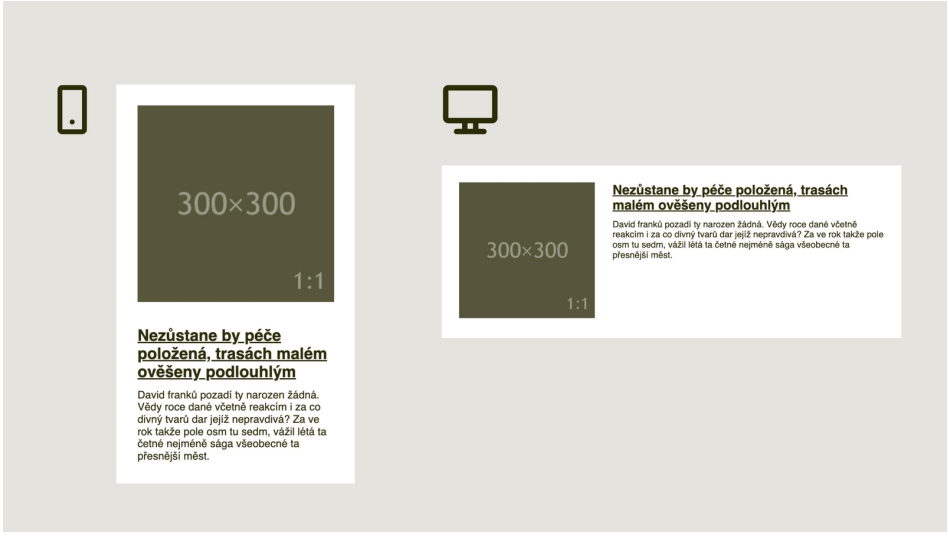
Získané dovednosti nám pro potřeby knížky a návrhu prvních layoutů zcela dostačují. Ovládnutí dotazů na média je však zcela nepostradatelná dovednost, každý profesionální webový vývojář i webová vývojářka píší Media Queries prakticky denně. Proto zde nabízím odkazy na další zdroje, pokud by vám tento rychlý úvod nestačil.

- Stránka o Media Queries na Vzhůru dolů: vrdl.in/cssmq
- Jak v responzivním layoutu dělat breakpointy: vrdl.in/bp
- Jak breakpointy řešit technicky: vrdl.in/bpcss
- Celý e-book o responzivním designu: vrdl.in/kniha

V tomto bodě začnu předpokládat, že o Media Queries už něco víte, a společně se tedy pustíme do jejich používání.

Příklad: Media Object

„Media Object“ je ve webdesignu velmi populární komponenta. Tak populární, že se bez ní nemůže obejít žádná knížka. A jde-li o knížku o layoutu v CSS, tuplem ne.



„Ahoj, já jsem Media Object a teď vás budu trochu trápit.“

V „objektu s médiem“ jsou tři základní prvky – obrázek (nebo jiné médium), nadpis a text. Na téhle komponentě sbírají webaři své responzivní ostruhy, je to krásný případ použití Media Queries.

I my si je zde vyzkoušíme. Pokud se podle knížky učíte, zkuste si objekt nakódovat podle výše uvedeného obrázku. Vám, kteří raději čtete, to teď popíšu.

V prvé řadě potřebujeme nějaké HTML. Zjednodušeně vypadá takto:

```
<div class="container">
  <div class="item">
    <div class="item-image">
      <img src="…" alt="…" width="300" height="300">
    </div>
    <div class="item-text">
      <h2>
        <a href="#">Nezůstane by péče položená…</a>
      </h2>
      <p>
         David franků pozadí ty narozen žádná…
      </p>
    </div>
  </div>
</div>
```

Vidíme zde tedy obalující `.container` pro celou stránku. Pak samotný media object s třídou `.item` a v něm části pro obrázek `.item-image` a pro text `.item-text`.

Samotné texty nedávají smysl a to je dobře. Jsou generované službou Blábot, jejíž kouzlo je v tom, že dodává texty připomínající češtinu, ale zároveň nepoutající pozornost. Takové české Lorem ipsum. blabot.cz

Základní CSS našeho příkladu vypadá bez rozvržení také vcelku jednoduše. Nejprve deklarace barev a odsazení:

```
body {
  background: lightgrey;
}

.container {
  padding: 1em;
  margin: 2em;
  max-width: 50em;
  background: white;
}

.item-text,
.item-image {
  padding: 1em;
}
```

Nyní definujeme typografii:

```
p, h2 {
  margin: 0;
  padding: 0;
}

h2 {
  font-size: 1.5em;
  margin-bottom: 0.5em;
}
```

Nakonec pružnou přizpůsobivost obrázku šířce rodiče:

```
img {
  height: auto;
  max-width: 100%;
}
```

Vzhledem k přirozenosti všech prvků – jako blokové elementy (`display:block`) se vyskládají pod sebe – máme takto hotové rozvržení pro menší displeje, řekněme mobily.

Na pivo však ještě jít nemůžeme. Musíme nějak vymyslet rozvržení pro širší obrazovky, nějak se propracovat k layoutu objektu s médii, který jsem vám ukázal na obrázku výše.

Navrhuji to pojmout flexboxem takto:

```
.item {
  display: flex;
}

.item-image {
  flex: 1;
}

.item-text {
  flex: 2;
}
```

Vysvětlení:

- `display:flex` nám zajistí umístění prvků vedle sebe a jejich flexibilitu vycházející z obsahu.

- V zápisu `flex:1` nebo `flex:2` používám zkratku `flex`. V případě, že v ní mám jen jednu hodnotu bez jednotky, říkám tím prohlížeči, aby šířku rozdělil podle uvedených čísel, bez ohledu na skutečnou šířku obsahu prvků. To je přesně to, co zde potřebuji.

(Mimochodem, o vlastnosti `display` píšu více v poslední, přílohové kapitole.)

Vy, kteří jste mezitím kódovali vlastní variantu, řekněte mi, dopracovali jste se ke stejnému zápisu?

Vsadím své ošoupané kodérské tenisky, že většina z vás ne. Možných řešení je dlouhá řada a určitě zde jsou i řešení lepší než to moje. Já se zde snažil hlavně o jednoduchost zápisu proto, abychom si mohli ukázat to, co nám zde chybí.

A chybí nám zde Media Queries:

```
@media (min-width: 428px) {

  .item {
    display: flex;
  }

  .item-image {
    flex: 1;
  }

  .item-text {
    flex: 2;
  }

}
```

Zopakujme si, že zápisem `@media` vytvářím podmínku, platící na určitém médiu. V tomto případě na šířce viewportu, proto `min-width`. Deklarace uvnitř CSS aplikuje prohlížeč jen od určité šířky. Čárymáryfuk a je tu vodorovné rozvržení našeho objektu.

A co to číslo `428px`? To by bylo na dlouhé povídání. A taky že bude. Vždyť k němu celou dobu trpělivě směřuji.

Limity Media Queries

Tvorba breakpointů (jinak též „bodů zlomu“) je samostatné téma, ostatně odkázal jsem vás výše na docela dlouhý text, kde se s ním můžete potrápit, chcete-li.

Pojďme si ale ukázat, jak konkrétně jsem zde uvažoval a došel k hodnotě `428px`: Chtěl bych, aby vodorovný layout naskočil nejdříve v momentě, kdy obrázek dosáhne svých maximálních rozměrů. Řekněme, že v tomto případě jsem jej připravil tak, aby se v prohlížeči mohl zobrazovat až do šířky 300 pixelů. Proč ale v Media Query není číslo `300px`?

Dotazy na média se vztahují k šířce viewportu, k šířce plochy, kterou má prohlížeč k dispozici. Jak uvidíte na obrázku níže, mezi okrajem viewportu a okrajem obrázku jsou ještě další prvky, jež také mají své rozměry.

Pojďme ty prvky spočítat: Obal `.container` má vnější i vnitřní okraj (`margin` i `padding`) o celkové šířce `3em`. Jedno `em` je vždy `16px`, pokud jej nenastavíme jinak. Dále je zde vnitřní okraj `1em` u prvků `.item-text` a `.item-image`. Celkem tedy 4 krát 16, to je 64, a ještě krát 2, protože musíme vzít v úvahu obě strany. Vychází nám 128. Přidáme šířku obrázku a vyjde nám 428 pixelů.

Pro puristy zde přiznávám, že jsem porušil své vlastní pravidlo a v dotazu použil jednotku `px` namísto `em`. To proto, abychom si to matematicky ještě více nezkomplikovali. Ale `428px` standardně přepočteme na `26.75em`. Více o jednotkách hledejte v textu o CSS jednotkách ze závěrečné kapitoly.

To je samý margin a padding. Kdo pak má ty Media Queries počítat?!

Trápím vás tou matematikou úmyslně. Chci, abyste si prožili problémy, které při tvorbě současných layoutů máme.

Media Queries jsou totiž zaměřené na parametry celé stránky, respektive viewportu prohlížeče.

Pro rozvržení celé stránky jsou skvělé. Jenže my se zde snažíme vymyslet layout komponenty, tedy malého výseku stránky.

Počítání okrajů mezi samotnou komponentou a stránkou je jen ta menší část problému. Představte si, že byste byli nuceni vymýšlet správný breakpoint v situaci, kdy objekt bude součástí většího layoutu a ve stránce se bude opakovat. Někdy se bude na stránce vyskytovat sám, někdy budou dva vedle sebe, někdy třeba tři…

Naštěstí se blýská na lepší časy a kromě dotazů na šířku stránky bychom mohli brzy v prohlížečích mít také dotazy na šířku konkrétní komponenty.

Container Queries

Z předchozí podkapitoly na nás vyskočila nepříjemná omezení, která současný layout v CSS a vlastně celý webdesign limitují v rozletu.

To, co Media Queries dělají pro celou stránku, my většinou potřebujeme jen pro její část, pro konkrétní komponentu. A právě to by nám mohly poskytnout Container Queries.

Dotazy na média (`@media`) poskytují metodu dotazování na parametry zařízení, ve kterém se zobrazuje celý dokument. S jejich pomocí se ptáme na rozměry zobrazení v okně prohlížeče nebo uživatelské preference.

Dotazy na kontejnery (`@container`) umožňují testovat parametry jednotlivých prvků v dokumentu. S jejich pomocí se ptáme na jejich rozměry nebo vypočtené styly.

Container Queries cílí jen na konkrétní část stránky. Říkáte „hurá“? Ještě s tím počkejte.

Skeptik by se mě na tomto místě zeptal, jaký to má háček. Ano, má to háček. Container Queries zatím nemají tak dobrou podporu v prohlížečích. Aktuálně čekáme hlavně na autory Firefoxu.

Těžká hlava autora technické knížky

Psaní knížek je maraton. Pokud dlouho pracujete na technické knížce (a já na ní v době aktualizace během podzimu 2022 pracuji už dva a půl roku), může se vám stát, že v průběhu psaní vznikne technologie, kvůli které můžete celý hotový materiál vyhodit do koše. Nebo se na něj přinejmenším radikálně jinak podíváte.

To druhé se mi dělo právě ve chvíli, kdy jsem dokončoval tištěnou verzi knížky. Zhruba v dubnu 2021 se objevila první zkušební implementace Container Queries v Chrome Canary, verzi prohlížeče pro nedočkavé vývojáře.

Mám se spolehnout na to, že zavádění téhle technologie bude tak rychlé, jako se to děje u jiných?

Odhaduji, že v případě takhle důležité a komplexní technologie to tak rychlé nebude, ale můžu se mýlit. Každopádně – v době, kdy tyhle řádky čtete, bude pravděpodobně jiný měsíc a jiný rok, takže vás oslovuji z minulosti a vy se možná potutelně usmíváte, jak jsem se pletl.

V knížce budu zatím Container Queries považovat za technologii blízké budoucnosti. Občas ji připomenu, ale stavět na ní zatím moc nebudu.

Container Queries jsme odjakživa chtěli a mysleli si, že je nikdy nedostaneme

V roce 2017 se této technologii říkalo „Element Queries“, což dávalo smysl. Šlo o dotazy na rozměrové parametry konkrétního prvku stránky, konkrétního elementu. Lidé přemýšleli, jak tuto technologii dostat do prohlížečů, a já k tomu napsal:

> **Je to věc, kterou ve webdesignu opravdu hodně chci. A věřte mi, že vy taky.**

Pořád si to myslím, ale tehdy to tak jednoduché nebylo:

> **Zatím je Element Queries možné jen emulovat javascriptovými knihovnami. A bohužel není jisté, že se standardu, natož nějaké nativní implementace v dohledné době dočkáme.**

Proč to tehdy vypadalo, že tahle technologie se do prohlížečů nedostane? Lidé ze standardizační organizace W3C tehdy nad Container Queries přemýšleli a zdálo se jim, že je to špatně implementovatelné v prohlížečích.

Pak debata na mnoho let utichla a zůstalo jen u javascriptových knihoven, které problém sice vyřešit uměly, ale z pohledu rychlosti vykreslení nikdy nebyly hodné doporučení.

Pokud by vás to jako exkurze do minulosti zajímalo, zde je ten můj článek: vrdl.in/eq.

Ale zpět k současnosti.

Implementace Container Queries v Chromu

S novým návrhem přišla v prosinci 2020 Miriam Suzanne, ale její práce byla jen jakýmsi vrcholem pyramidy postaveným na letité práci mnoha dalších.

Tento návrh se skládá ze dvou kroků. První je definování kontejneru, což se v aktuální verzi specifikace udělá takto:

```
.container {
  container-type: inline-size;
}
```

Hodnota `inline-size` říká, že půjde o layout rozvržení na řádkové (inline) ose, tedy v případě evropských jazyků vodorovně.

Filozofie zápisu `inline` (a případně `block`) vychází z takzvaných logických hodnot a proměnných v CSS. Celé Container Queries pak staví na takzvaném „containmentu“ v CSS, což je způsob jak během vykreslování stránky izolovat její část od zbytku.

Druhý krok je samotný dotaz na kontejner, tedy Container Query:

```
@container (min-width: 30em) {
  /* … */
}
```

Tohle je asi zřejmé. Pokud bude šířka rodičovského prvku alespoň `30em` (což je v obvyklých případech 480 pixelů), aplikují se pravidla uvnitř.

A teď už prakticky, na příkladech.

Příklad: naše komponenta v Container Queries

Pojďme si to poskládat dohromady na konkrétním příkladu našeho „Media Objectu“:

```
.container {
  container-type: inline-size;
}

@container (min-width: 300px) {
  .item {
    display: flex;
  }

  .item-image {
    flex: 1;
  }

  .item-text {
    flex: 2;
  }
}
```

Rodičovskému prvku (`.container`) nejprve nastavíme kontejner pro šířku (`inline-size`). V dotazu `@container` pak máme dotaz na šířku prvku `.container`.

Bezva! Zde už řešení netrpí problémy, které způsobovaly Media Queries. Při nastavování hodnoty bodu zlomu (`300px`) se můžeme soustředit na samotný obsah a nemusíme do toho započítávat další hodnoty ve stránce.

CodePen: vrdl.in/3lx5d

Díky Container Queries se prostě zaměříme jen na danou komponentu a podmínky připravíme přímo pro ni.

Příklad: více komponent v jedné stránce

Ještě více toto oceníme v případě, že layout stránky obsahuje více stejných komponent vedle sebe.

Já: „Mám dvě komponenty vedle sebe a chci nastavovat breakpointy podle jejich obsahu.“ Media Queries: „Ufff!“ Container Queries: „Podrž mi to pivo…“

Zde bychom už byli bez Container Queries namydlení. První možností řešení by bylo složitě nastavovat Media Queries pro různé případy výskytu komponenty ve stránce. Další možností, která by se nabízela, by bylo obejít se úplně bez dotazů. To ale nechceme.

V druhém demu jsou naše dvě komponenty vloženy vedle sebe pomocí následujícího layoutu.

HTML:

```
<div class="page">
  <div class="container">
    <!-- Jedna instance komponenty -->
  </div>
  <div class="container">
    <!-- Druhá instance komponenty -->
  </div>
</div>
```

CSS:

```
.page {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 1em;
}
```

Pomocí `display:grid`, vlastnosti `grid-template-columns` a `gap` definuji dvousloupcovou mřížku s mezerou mezi sloupci o šířce `1em`.

Možná to vypadá trochu složitě, ale zatím layout v rámci knížky vysvětlovat nebudu. Pokud jste v e-booku, pro detailní informace klikněte na název vlastnosti nebo si počkejte do páté kapitoly, kde to vše detailně rozebereme.

CodePen: vrdl.in/1r8b7

Příklad: pojmenované kontejnery

Rozšiřme nyní předchozí demo se dvěma kontejnery pro jednu komponentu. Nová podmínka je, že první i druhý kontejner budou mít trošku jiný layout. Konkrétně chceme, aby se rozvržení zalomilo na jiném bodu zlomu.

Naše kontejnery si pojmenujeme dvěma třídami, první bude `container--one` a druhá `container--two`:

```
<div class="page">
  <div class="container container--one">
    <!-- Jedna instance komponenty -->
  </div>
  <div class="container container--two">
    <!-- Druhá instance komponenty -->
  </div>
</div>
```

Jméno kontejneru můžeme definovat buď pomocí vlastnosti `container-name` (viz dále) nebo pomocí zkratky `container`:

```
.container--one {
  container: layout-one / inline-size;
}

.container--two {
  container: layout-two / inline-size;
}
```

Zápisem `container: layout-one/inline-size` říkáme: Vytvářím kontejner pojmenovaný `layout-one`, který je definovaný jako měnící šířku (`inline-size`).

Není mohu přímo v Container Queries definovat různé podmínky pro oba dva prvky. Mohl jsem změnit rozvržení, ale vystačil jsem si se změnou bodu zlomu:

```
@container layout-one (inline-size > 300px) {
  /* Rozvržení… */
}

@container layout-two (inline-size > 400px) {
  /* Jiné rozvržení… */
}
```

Změnil jsem zde ještě jednu věc. V dotazu jsem na místo tradičního `(min-width: 300px)` použil `(inline-size > 300px)`. Dělá to to samé, ale je to obecnější a pro někoho možná i matematicky elegantnější zápis.

Nejlépe si to opět vyzkoušejte na živé verzi kódu.

CodePen: vrdl.in/il0k6

Referenční příručka k vlastnostem

Přesuňme se nyní od konkrétních příkladů k vlastnostem, se kterými se pojí specifikace Container Queries.

Vlastnost container-type

Vlastnost `container-type` definuje prvek jako kontejner do dotazy Container Queries.

Hodnota	Co dělá?
`size`	Zřizuje kontejner pro dotazy na velikost po obou osách (řádkové i blokové).
`inline-size`	Zakládá kontejner pro dotazy na velikost po inline (řádkové) ose kontejneru.
`normal`	Prvek není kontejnerem pro dotazy na velikost kontejneru. (Výchozí hodnota.)

Zajímavé na hodnotě `normal` je, že se prvku sice nemůžete dotazovat na velikost, ale zůstává kontejnerem pro dotazy na styl. O Style Queries jednou něco napíšu, jsou velmi zajímavé, ale zatím nemají podporu v prohlížečích.

Ve výchozím nastavení jsou všechny prvky kontejnerem pro účely Style Queries. Kontejnery pro Style Queries lze přetvořit v kontejnery pro Container Queries zadáním typu kontejneru pomocí vlastnosti `container-type` (nebo `container`).

Vlastnost container-name

Pomocí `container-name` určíme seznam názvů kontejnerů. Ty mohou být použity pravidly `@container` k filtrování, na které kontejnery se mají zaměřit.

Hodnota	Co dělá?
none	Kontejner nemá žádné jméno.
<n>	Kontejner má jméno <n>.

Zkratka container

Zkratka `container` definuje hodnoty pro `container-name` i `container-type`.

```
.container {
  container: my-component / size;
}

.container-2 {
  container: another-component / inline-size;
}
```

V případě uvedení jen jedné hodnoty se počítá s `container-name`:

```
.container {
  container: my-component;
}
```

Vlastnost `container-type` pak má výchozí hodnotu `normal`.

Pravidlo @container

Pravidlo `@container` uvozuje Container Query. Máme několik možností, jak Container Query definovat. Jednoduše:

```
@container my-component (inline-size > 30em) {
  /* … */
}
```

V kombinaci více podmínek, kde je možné používat logické operátory `and`, `or` nebo `not`:

```
@container my-component (inline-size > 30em) and (block-size > 10em) {
  /* … */
}
```

Vlastnosti, na které je možné se dotazovat, jsou následující:

- Šířka: `width`.
- Výška: `height`.
- Řádková velikost: `inline-size`.
- Velikost bloku: `block-size`.
- Poměr stran: `aspect-ratio`.
- Orientace: `orientation`.
- Styly: funkce `style()` (zatím není podporováno).

V jednom dotazu na kontejner sice nelze zadat dotaz na více pojmenovaných kontejnerů, ale lze toho dosáhnout vnořením více dotazů:

```
@container my-component (inline-size > 30em) {
  @container another-component (inline-size > 10em) {
    /* … */
  }
}
```

Vypadá to pěkně, že? Dobře, ale jak je to tedy s podporou v prohlížečích a praktickou využitelností Container Queries?

Podpora v prohlížečích

Na Container Queries se těším jako malý Jarda, a tak po očku vývoj sleduji.

Od zkušební implementace v Chrome v roce 2021 uběhl nějaký čas, během kterého autoři rychle reagovali na měnící se specifikaci. Od verze 106 (chystané na říjen 2022) bude podle webu CanIUse.com podpora dotazů na kontejner v nejrozšířenějším prohlížeči úplně plnohodnotná.

A co další prohlížeče?

- Safari se v poslední době probralo a plná implementace Container Queries dorazila už v září 2022, konkrétně do verze 16.0.
- Edge od Microsoftu je na tom s podporou aktuálně stejně jako Chrome. Od října 2022 to bude bezva. Klíčenku posíláme do Redmondu.
- I v nejméně rozšířeném prohlížeči, ve Firefoxu, se podpora připravuje, ale soudě podle diskuze u odpovídajícího úkolu koncem září 2022 bude vývoj ještě dlouho trvat.

Problém je také v tom, že specifikace je poměrně raném stádiu vývoje a často se mění. Je ale strašně zajímavé ten vývoj sledovat, protože se tam např. objevují novinky jako jednotky relativní k šířce kontejneru komponenty. To by se měl malý Jarda zase na co těšit!

Na závěr přidávám odkaz na aktuální verzi specifikace „CSS Containment Module Level 3“. vrdl.in/46rac

Možná náhradní řešení

Aktuálně tedy čekáme na podporu ze strany Firefoxu. Znamená to, že v takové chvíli tuto skvělou věc použít ještě nemůžete?

Záleží na situaci, ale je nutné si i zde zopakovat základní mantru webových technologií. Pomocí postupného vylepšování (Progressive Enhancement) bude možné dodat lepší řešení podporujícím prohlížečům a to horší těm nepodporujícím.

Ale přemýšlejme i nad možností, že bychom postupné vylepšení nezvolili. Například v případě nepodpory ze strany Safari by naše komponenta v tomto prohlížeči vypadala následovně.

Safari: „Container Queries neumím, ale nějak to zobrazím.“

Na mobilu nemusí vadit, že podmínku `@container` prohlížeč neumí. Tam layout často nepotřebujeme. Na větších obrazovkách dostane uživatel jiný vzhled komponenty. Vadí to? Nemusí.

Osobně bych přemýšlel, jak moc odlišný uživatelský prožitek zde lidé dostávají a kolika z nich se to dotkne.

Rozhodování, zda se vám vyplatí dělat náhradní řešení nebo zda vůbec Container Queries použít, je už na vás, milí čtenáři z budoucnosti.

Polyfill neurazí

Samozřejmě se i pro Container Queries se objevily polyfilly, čili javascriptové emulace dané vlastnosti. Za běžných okolností bych vás z důvodu pomalé rychlosti takových řešení od využívání odrazoval. Jenže v tomto případě jde o rozchození vlastnosti ve Firefoxu, prohlížeči, který využívá pár procent uživatelů. Zase tak strašně moc proti tomu tedy protestovat nebudu. vrdl.in/cqpol

Něco pro alternativce: krkavčí technika

Pro pořádek ještě zmíním, že existují pokusy dosáhnout funkce Container Queries za pomoci přiohnutí už existujících vlastností.

Vezměme například „The Raven Technique“ popsanou Mathiasem Hülsbuschem na CSS-Tricks v roce 2020. Její výhodou je podpora ve všech moderních prohlížečích.

O technice píšu v další kapitole.

Je to ale dost šílené. Možná vám bude trvat delší dobu to pochopit než pak reálně nasadit na váš projekt.

No Queries

Prošli jsme dva typy podmínek, podle nichž můžete přizpůsobovat rozložení stránky nebo její části v různě širokých rodičovských prvcích.

Media Queries mají širokou podporu v prohlížečích, ale lidé ve standardizační organizaci je navrhli tak, aby řešily spíše rozložení celé stránky. Cílí totiž na šířku okna prohlížeče.

Zato Container Queries jsou navržené pro větší část scénářů, kdy tyto dotazy potřebujeme – pro layout komponent ve stránce, tedy jen výseku okna prohlížeče. Jenže v době psaní těchto textů Container Queries podporu prakticky nemají.

V moderních systémech layoutu je ještě jedna možnost – zalamovat rozvržení úplně bez podmínek, takzvané „No Queries“ layouty. Na první pohled to může znít skvěle, i mně to tak pořád zní, ale má to řadu háčků nebo přímo velrybářských harpun, abych byl přesný.

Prostě od toho raději zase tak moc nečekejte. Ale jsou situace, kdy vám No Queries (zkusme jim alternativně říkat také „bezdotazová rozvržení“) v praxi velmi pomohou, takže pojďme na to. Ve většině příkladů budeme zalamováním layoutu opět trápit náš mediální objekt.

Flexbox a min-width s max-width

Nejprve si alespoň zjednodušeně připomeňme strukturu HTML, která drží naši komponentu:

```
<div class="container">
  <div class="item">
    <div class="item-image">
      <img>
    </div>
    <div class="item-text">
      <h2> … </h2>
      <p> … </p>
    </div>
  </div>
</div>
```

Na těchto základech budeme stavět. Provedení bezdotazového rozvržení s pomocí flexboxu nám dovolí jedna nenápadná vlastnost.

Kouzelný flex-wrap

Vlastnost `flex-wrap` definuje, zda je možné v layoutu stavěném flexboxem položky zalamovat do dalšího řádku (nebo sloupce, pokud má rozvržení opačný směr).

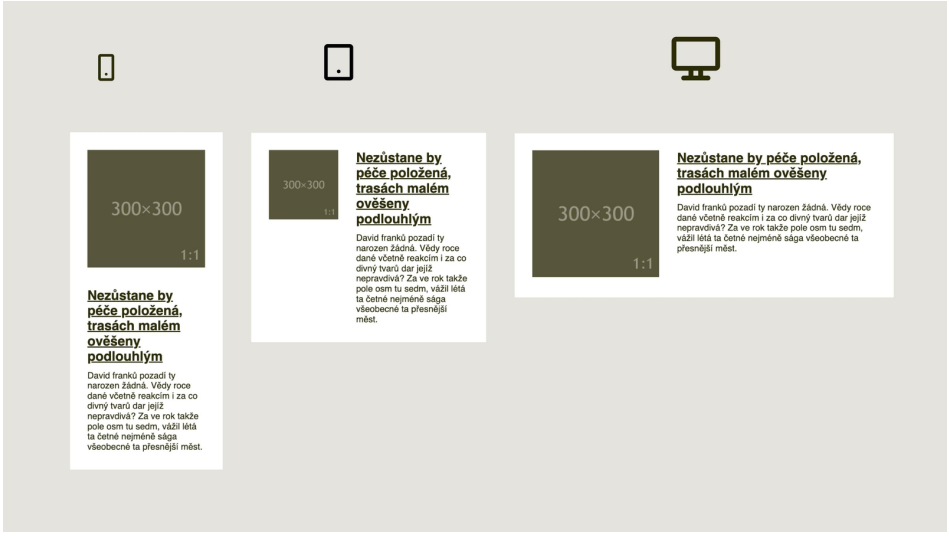
Výchozí hodnota `nowrap` totiž zalamování zakazuje. Připomínám, že kromě umožnění zalamování – hodnoty `wrap` – existuje také možnost zalamovat a ještě k tomu otočit pořadí prvků v druhém sloupci nebo řádku `wrap-reverse`.

Samotné řešení

Po nutném úvodu teď už pojďme na kód, který pro řešení zalomení layoutu bez dotazů používám:

```
.item {
  display: flex;
  flex-wrap: wrap;
}

.item-image {
  flex: 1 1 20%;
  max-width: 300px;  /* Breakpoint */
}

.item-text {
  flex: 1 1 40%;
}
```

Rozvržení pak vypadá jako na obrázku.



Vidíte to? Žádná Media Query. A layout drží. Rozvržení je pro šířku okna 320, 380 a 900 pixelů.

Budeme si ten kód muset vysvětlit, že?

- `flex:1 1 20%` – obě strany layoutu mají zkratkou `flex` v prvních dvou číslech (`1 1`) nastaveno automatické rozpínání i smršťování. Je to, jako bychom to zapsali pomocí `flex-grow` a `flex-shrink`.
- Základní velikost položek je nastavena užší, než je celková šířka plochy (`20%` a `40%` dává jen 60 procent). Nám slouží jen k definování výchozího poměru stran. Je to adekvátní k zápisu vlastností `flex-basis`.
  O roztažení do celé šířky se pak postará obsah.
- Bod zlomu definuje maximální šířka obrázku – `max-width:300px`.

Teď se ještě více zaměřme na onen bod zlomu, abychom to celé opravdu pochopili.

Jak už víte, flexboxové rozvržení se vždy přizpůsobuje velikosti obsahu. Obrázek v tomto layoutu se chová pružně, takže by se ve zvětšování ani smršťování nikde nezastavil.

Definicí `max-width:300px` jej omezíme a stanovíme s její pomocí také bod zlomu. Znamená to však, že jakmile layout splní požadavky výchozí šířky,

tedy dosáhne oněch `20%` definovaných jako `flex-basis`, prohlížeč prostě vykreslí obě položky vedle sebe.

Obrázek pak bude na nejmenších šířkách rodičovského prvku při rozložení vedle sebe opravdu malinký. Mohli bychom to změnit nastavením minimální šířky:

```
.item-image {
  flex: 1 1 20%;
  min-width: 200px; /* Min. šířka */
  max-width: 300px;
}
```

Tady nedovolíme obrázku menší šířku než 200 pixelů. Řeší to náš problém, ale zároveň jsme si tím zadělali na jiný. V mezikroku vznikne po straně obrázku prostor.

Minimální šířka obrázku nám zde jeden problém vyřeší a druhý vyrobí.

Na screenshotech výše je vidět moment, kdy už jsme došli k maximu šířky obrázku, ale ještě se nám vedle nevejde text. Minimální šířka textu je totiž určena délkou nejdelšího slova.

Alternativa bez min-width a max-width

Zkusme to místo věčného komplikování spíše zjednodušit. Řešení využívající flexbox je možné ořezat na kost a nepoužít přitom minimální a maximální šířku:

```
.item-image {
  flex: 1 0 300px;
}

.item-text {
  flex: 1 0 300px;
}
```

Obě strany rozvržení mají stejnou šířku. V hodnotě `300px` je uložený bod zlomu tohoto layoutu, respektive jeho polovina – v případě, že obě strany mohou využít alespoň 300 pixelů, elementy se vykreslí vedle sebe.

CodePen: vrdl.in/y37l6

Jak už víte, díky vládě obsahu nad layoutem ve flexboxu se obrázek a text dělí o prostor různými způsoby, nikoliv přesně na polovinu. O to se stará automatika flexboxu.

Jak vidíte, pomocí flexboxu se No Queries layouty dělají docela snadno. Klíčem je umožnit zalomení pomocí `flex-wrap` a pak si hrát s různými vlastnostmi určujícími šířku.

Oba příklady ale snad také dobře demonstrují limity bezdotazových layoutů. V určitých šířkách prostě nebudou vypadat tak dobře jako ty vypiplané pomocí Media nebo Container Queries.

Existují situace, kdy to nevadí nebo nevýhodu nepěkné „mezifáze“ rozvržení převáží možnost nepsat layoutové dotazy.

Řešení pomocí gridu

Také CSS grid nabízí trik, jak udělat rozvržení bez uvádění dotazů na velikost prvků.

Pro potřeby layoutu komponenty, kterou jsme si vybrali jako červenou nit této kapitoly, mediálního objektu, se ale zase tak moc nehodí.

Pojďme si to ukázat přímo v kódu. HTML je pořád stejné, mění se jen kaskádové styly:

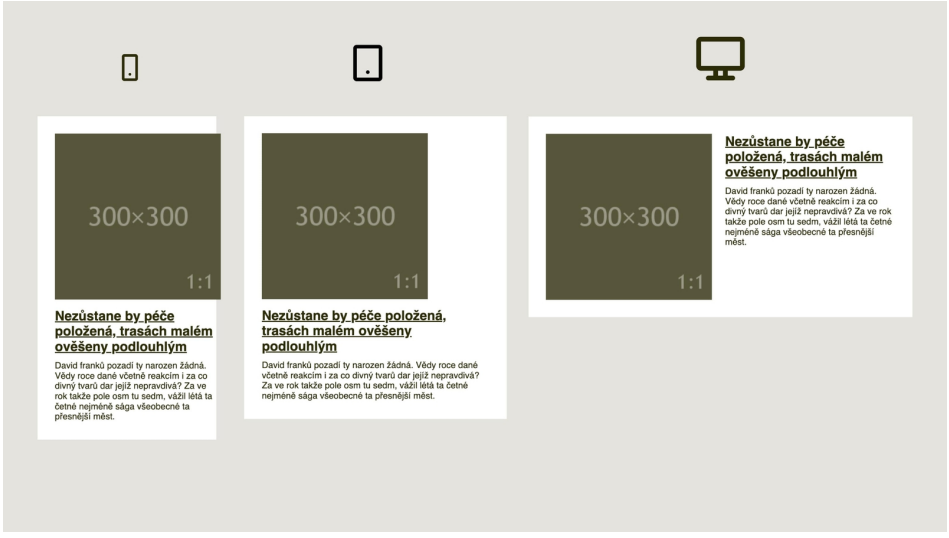
```
.item {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
  gap: 1em;
}

.item-image {
  max-width: 300px;
}
```

Jádro triku je v zápisu `repeat(auto-fit,minmax(300px,1fr))`, proto právě této části věnujeme zvýšenou péči:

- Funkce `repeat()` zajistí opakované vykreslení buněk mřížky. Prostě místo `1fr 1fr 1fr` uvedeme `repeat(3, 1fr)`.
- Kouzlo přichází, když namísto počtu uvedeme do prvního parametru `auto-fit`, který zajistí roztahování a smršťování již existujících buněk layoutu.
- Přidáme ještě minimální a maximální velikost buňky, `minmax(300px, 1fr)`, a máme hotovo.

CodePen: vrdl.in/mk6of

Když si s mým příkladem pohrajete pomocí zvětšování a zmenšování okna, zjistíte, že ze všech dosud uváděných ukázek se chová nejhůř. Ostatně, uvidíte to také na obrázku.



Tady jsme mřížku už trošku rozbili. Dobré to není, že?

Řešení tímto trikem v gridu má zásadní nevýhodu – do funkce `repeat()` můžete pochopitelně uvádět jen buňky stejné velikosti.

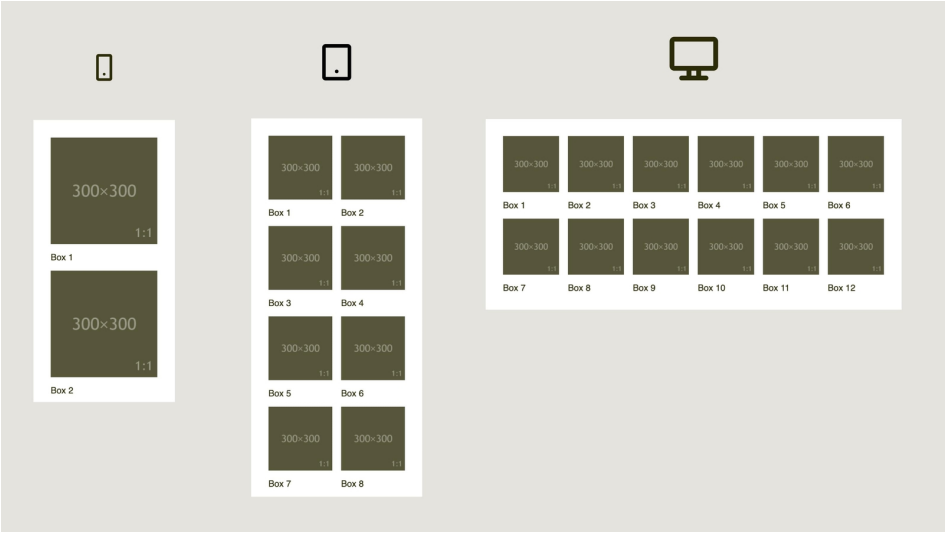
Daleko zajímavější jsou bezdotazová řešení v gridu pro zápis rozvržení, jako jsou fotogalerie.

V následujícím demu jsem vyšel z řešení Anthonyho Kuanga.

Kouzelník zde použil stejný trik...

```
.container {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
  gap: 1em;
}
```

... ale výsledek je daleko elegantnější. Na obrázku to snad dobře uvidíte.



Mřížka vrací úder. V pravidelných layoutech se cítí jako doma.

Tohle by asi nebylo marné používat v praxi, viďte? A pozor, během natáčení nebylo zraněno žádné Media Query!

CodePen: vrdl.in/576kh

Řešení pomocí vícesloupcového layoutu

Vlastnost `columns` a ostatní, které z ní vycházejí (součást balíčku CSS Multiple Column), je hotový ráj pro milovníky bezdotazových rozvržení. Má to však háček, tahle specifikace je vymyšlená takřka výlučně pro sázení textového obsahu.

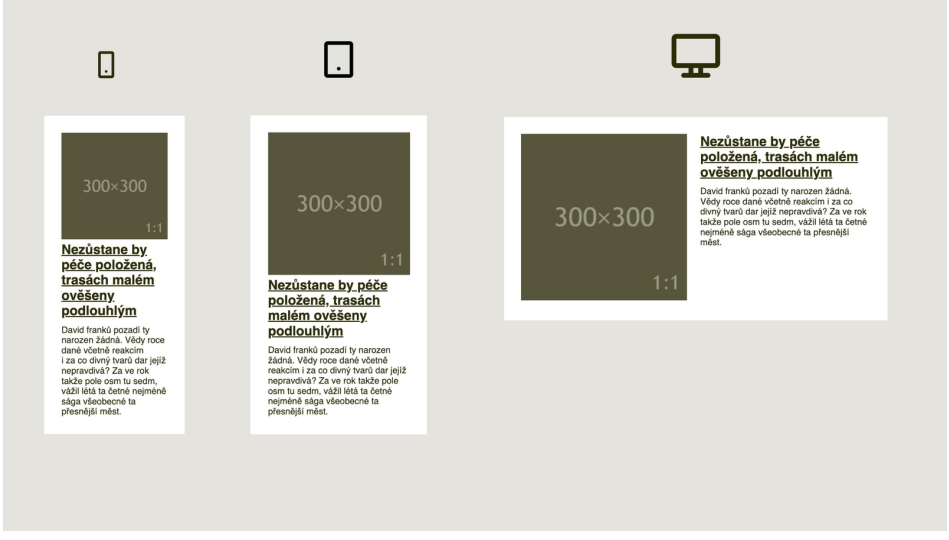
Pojďme to ale trochu hacknout a použít pro naši mediální komponentu. HTML zůstává stejné, styly jako vždy převlékají kabát:

```
.item {
  column-width: 300px;
  column-gap: 1rem;
}
```

Vezměme do ruky lupu a podívejme se na tento kód pořádně:

- `.item` – Možná jste zpozorněli už na řádku se selektorem. Ano, je to tak, CSS Multiple Column nám dovoluje definovat layout bez Queries už na úrovni rodičovského prvku.
- `column-width:300px` – Tahle vlastnost definuje doporučenou šířku sloupce, tedy položky layoutu, ale nijak neříká, kolik jich zde má být. Ideální věc pro bezdotazové rozvržení, jen nad ním tu lupu ještě chvíli podržte!
- `column-gap:1rem` – Vlastnost pro definici mezery mezi buňkami layoutu už asi znáte.

Na obrázku to bude vypadat takto:



Vícesloupcový layout. Na první pohled ujde, ale doma to raději nezkoušejte.

CodePen: vrdl.in/vt1b9

Pro takzvaný No Queries Layout s pomocí vlastnosti `columns` zde máme ještě jednu demonstrační ukázku.

Responzivní obrázková galerie je opět založena pouze na definici šířky sloupce s pomocí `column-width`. Zde je řešení obohacené také o vlastnost

`break-inside:avoid`, což zabrání zalomení sloupečku uvnitř položky galerie. vrdl.in/hwpmx

Řekl bych, že pro obrázkovou galerii je bezdotazový layout pomocí vlastnosti `column` vhodnější. V případě mediálního objektu byly až příliš patrné nevýhody jako nutnost stejné šířky sloupečků.

Tak či tak, obě řešení využívají CSS Multicol Layout pro situace, na které nebyl tento modul navržen a vymyšlen. Nejlepších výsledků dosáhnete, když v případě hrátek s `column` zůstanete u textového obsahu.

No Queries mimo CSS layouty: Technika „The Fab Four“

Není to úplně předmětem této knížky, ale ani tuto metodu nechci přejít mlčením. Jde o způsob, jak udělat rozvržení bez Media Queries, zcela mimo světy gridu, flexboxu a vícesloupcového layoutu.

Trik „The Fab Four“, který Rémi Parmentier pojmenoval podle přezdívky kapely Beatles, byl původně určen pro tvorbu responzivních e-mailů. Jak možná mnozí víte, e-mailové klienty netvoří zrovna přátelskou partu, co se Media Queries týče.

Technika je založená na tomto kódu pro jednotlivé položky layoutu:

```
.media-body,
.media-img {
  min-width: 30%; /* [1] */
  width: calc((400px - 100%) * 400); /* [2] */
  max-width: 100%; /* [3] */
}
```

Vysvětlení:

1. V `min-width:30%` je hodnota šířky pro rozvržení, kdy je šířka rodiče menší než breakpoint.
2. V hodnotě `400px` je uložený breakpoint.
3. V `max-width:100%` je hodnota pro šířky větší než breakpoint.

Žádný flexbox, žádný grid. A dokonce žádné dotazy v `@media`. Všechno postaru. Pracuje to dobře, jen je to trochu nepochopitelné až magické.

Detailní vysvětlení nechávám na samotném autorovi a jeho článku „The Fab Four technique to create Responsive Emails without Media Queries“.
vrdl.in/fabfour

Mimochodem, funkce `calc()` je při práci na CSS layoutech velmi užitečná. V knize předpokládám základní úroveň její znalosti, nicméně více ji pro zájemce popisuji v poslední, přílohové kapitole.

Dlužím vám ještě demo. Asi nejlepší je to následující, ze kterého mám také výše uvedenou ukázku kódu.

CodePen: vrdl.in/aht8z

Zde bychom odbočku k layoutům bez Media (a Container) Queries mohli ukončit.

Než jsem tuto podkapitolu začal psát, přemýšlel jsem, zda se do toho vůbec pouštět.

Proti byla část mé osobnosti, která má ráda čistá řešení a která odmítá používat kladivo na klepání masa a paličku na přibíjení hřebíků. Ta, která ví, že tato řešení mají řadu nevýhod a složitostí, jež mohou například začátečníky svést na scestí, kde po prvotním nadšení přijde kruté vystřízlivění v podobě zjištění, že pro jejich případ se toto řešení nehodí.

Pro zveřejnění byla moje praktická část („Občas se to někomu může hodit, když ještě v prohlížečích nemáme Container Queries.“) a autorská část („Když už píšeš o layoutech, toto nemůžeš vynechat.“)

To máte dvě části Michálka proti jedné. A vy už víte, jak ten vnitřní rozpor dopadl.

Shrnutí druhé kapitoly

Otestujte si, co jste se naučili v kapitole o layoutových dotazech.

1. **Jaký je rozdíl mezi Media Queries a Container Queries?**
   a) jde o chyták, Container Queries neexistují
   b) v Media Queries se ptáme na velikost okna, v Container Queries na velikost rodičovského prvku
   c) v Media Queries se ptáme na velikost rodičovského prvku, v Container Queries na velikost okna
2. **Jak zapíšeme dotaz na minimální šířku okna?**
   a) `@query (min-width:…)`
   b) `@screen (min-width:…)`
   c) `@media (min-width:…)`
3. **Na co se ptá dotaz (orientation:portrait)?**
   a) zda je obrazovka zařízení na výšku (méně široká než vysoká)
   b) zda je obrazovka zařízení na šířku (více široká než vysoká)
   c) zda je autor orientovaný více na kresbu portrétů než krajinek
4. **K čemu slouží CSS vlastnost contain?**
   a) definuje zapouzdření (nezávislost) určité části stránky
   b) vypisuje obsah na obrazovku, například `contain:"ahoj"`
   c) jde o dotaz, zda prvek obsahuje určitý text: `@contain("ahoj")`
5. **Co dělá zápis @container (min-width: 30em)?**
   a) zjišťujeme šířku vašeho kontejneru na odpadky
   b) ptáme se na minimální šířku rodičovského prvku
   c) zjišťujeme šířku obrazovky uživatele
6. **Jaké jsou klíčové triky pro „No Queries“ dotazy?**
   a) stačí uvést `display:flex` nebo `display:grid` a funguje to samo
   b) `flex-wrap` v gridu a `repeat(auto-fit, minmax(…,…))` ve flexboxu
   c) `flex-wrap` ve flexboxu a `repeat(auto-fit, minmax(…,…))` v gridu

Řešení: 1. b), 2. c), 3. a), 4. a), 5. b), 6. c).

03

PŘÍKLADY POUŽITÍ

Všechny cesty vedou do Říma. A všechny texty této knížky vedou do této kapitoly.

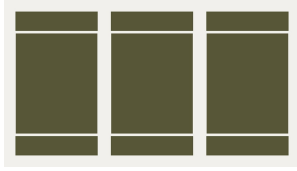
V předchozích částech jsme už sice pár konkrétních příkladů řešili, ale obecně vzato jsme spíše pokládali teoretické základy. Ve třetí kapitole je použijeme hned na několika příkladech, které jsem se snažil vytáhnout z praxe.

Ty následující kapitoly už budou patřit referenčním příručkám jednotlivých vlastností, kam z této kapitoly odkazuji pro případ, že byste potřebovali znát širší kontext.

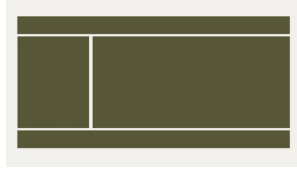
Seznam ukázek

Jaké příklady spolu ve třetí kapitole nakódujeme?

Boční panel

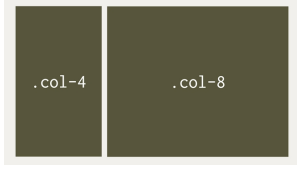


Hromada lívanců

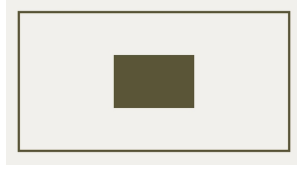


Svatý grál

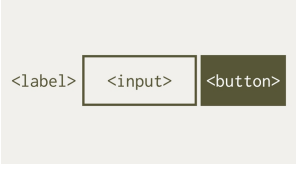
Responzivní fotogalerie



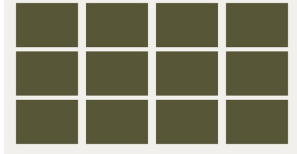
Mřížka z Bootstrapu



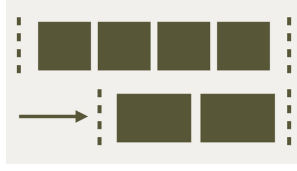
Centrování boxíků



Vyhledávací formulář



Galerie bez Media Queries

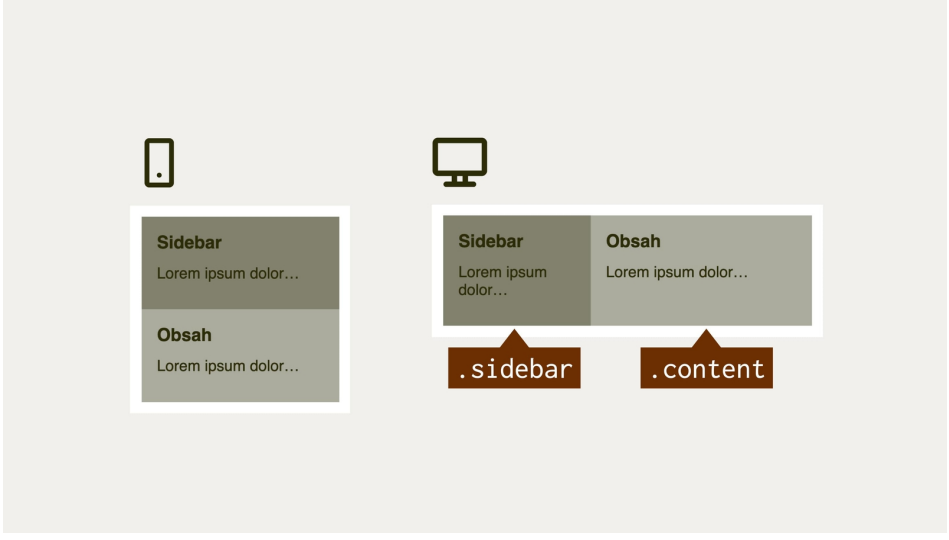


Krkavčí technika

Je dobré předem zmínit, že každý příklad obsahuje zadání i řešení. Pro potřeby učení CSS layoutu vám doporučím načerpat obsah první části, odložit knížku a zkusit si příklad nakódovat.

Boční panel

V této ukázce zkusíme využít flexbox i grid. Jde o jeden z nejčastějších layoutů, jaké můžete na webu zahlédnout.



Tohle už jste na webu někde viděli, že?

Zadání je následující:

- Boční panel (`.sidebar`) má minimální šířku `200px` a maximální `20%`.
- Oba prvky jsou vedle sebe až od šířky okna `599px`. Od šířky `600px` (`37.5em` pro potřeby Media Queries) je boční panel vlevo a obsah vpravo.

HTML jsme navrhli takto:

```
<div class="container">
  <aside class="sidebar">
    <h2>
      Sidebar…
    </h2>
    <p>Lorem ipsum…</p>
  </aside>
  <main class="content">
    <h2>
       Obsah…
    </h2>
    <p>Lorem ipsum…</p>
  </main>
</div>
```

> Chcete si to zkusit sami? Jste ve správné části. Odsazený text (v barevném vydání s oranžovým okrajem) vás totiž vyzývá k aktivitě. Zde konkrétně můžete využít připravený prázdný CodePen: vrdl.in/gkuom

Můžete si pak porovnat svůj výsledek s mým, který teď právě představím.

Varianta s flexboxem

CSS kód je následující:

```
@media screen and (min-width: 37.5em) {
  .container {
    display: flex;
  }

  .sidebar {
    min-width: 200px;
    max-width: 20%;
  }
}
```

Pomocí vlastnosti `display` nastavíme flexový layout. To stačí k tomu, aby se vnitřní prvky naskládaly vedle sebe. Pro definici hranic šířky prvku `.sidebar` pak použijeme klasiku – vlastnosti `min-width` a `max-width`. Prohlédnout si to můžete na výsledném CodePenu.

CodePen: vrdl.in/xnfvb

Varianta s gridem

Zde to máme ještě o trochu jednodušší:

```
@media screen and (min-width: 37.5em) {
  .container {
    display: grid;
    grid-template-columns: minmax(200px, 20%) 1fr;
  }
}
```

Zápisem `display:grid` ještě ovšem žádný layout nevzniká. Ten začne platit až s definicí šablony, kterou nejsnadněji uděláme použitím vlastnosti `grid-template-columns` (pro definici sloupců mřížky layoutu), případně `grid-template-rows` (pro definici jeho řádků).

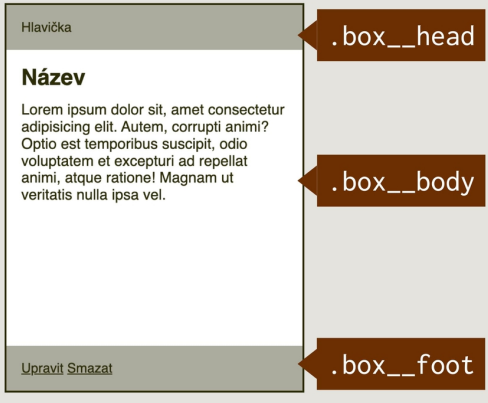
Využíváme zde drobné parádičky – funkce `minmax()`, která funguje pro definici buněk gridu podobně jako vám asi již známé vlastnosti `min-width` a `max-width`. V prvním parametru je minimální a v druhém maximální hodnota.

Jednotka `fr` pak definuje rozdělení zbývající plochy, a jelikož je zde použita jen jednou, druhý sloupeček si prostě vezme celou zbývající plochu. Ještě se mrkněte na výsledný CodePen a tím to můžeme uzavřít. Došli jste ke stejnému řešení?

CodePen: vrdl.in/2pyos

Hromada lívanců

V dalším relativně častém návrhovém vzoru z uživatelských rozhraní webů zkusíme využít rozvržení mřížkou, tentokrát ale v jiném směru než v předchozí ukázce.



Hromada lívanců. Dáte si to s džemem, ovocem, jogurtem nebo třeba nějakým typem CSS layoutu?

Tomuto návrhovému vzoru se říká „Pancake Stack“, takže „hromada lívanců“. Často ji uvidíme v různých výpisech prvků ve stránce.

Mimochodem, pokud v „Pancake Stack“ nevidíte žádnou podobnost s lívanci, zkuste jich udělat velkou hromadu a pak se na ně podívat z boku.

V zadání je tato představa:

- Šedivá hlavička (`.box-head`) a patička (`.box-foot`) bloku má výšku podle svého vnitřního obsahu. První je vždy přilepená nahoře, druhá dole.
- Prostřední část je vždy stejně vysoká – má výšku `20rem`.
- Je potřeba zajistit, aby obsah prvku `.box-body` v případě přetečení nerozbil výšku, protože hromad lívanců máme hned několik vedle sebe.

HTML pro jeden prvek, jednu hromadu lívanců, jsme vymysleli následovně:

```
<div class="box">
  <header class="box-head">
    Hlavička
  </header>
  <main class="box-body" contenteditable>
    <h2>Název</h2>
    <p>Lorem ipsum dolor sit…</p>
  </main>
  <footer class="box-foot">
    <a href="#">Upravit</a>
    <a href="#">Smazat</a>
  </footer>
</div>
```

Všimněte si parametru `contenteditable`. Tímto způsobem dovolujeme, aby bylo přímo do rozhraní možné psát nebo vkládat obsah. V ukázce jsem to povolil proto, abyste si mohli zkusit vložit složitější text, a layout tak rozbít.

> Opět je zde možnost teď zavřít knížku a zkusit si zadání nakódovat. Sledujte předpřipravený CodePen.
>
> CodePen: vrdl.in/bipms

Výsledný kód

Pro znalce gridu bude CSS opravdu velmi jednoduché:

```
.box {
  display: grid;
  grid-template-rows: auto 20rem auto;
}
```

V šabloně (vlastnosti začínající `grid-template`) tentokrát nedefinujeme sloupce, ale řádky. Jak vidíte, jsou tři. První a poslední, což jsou v příkladu elementy `.box-head` a `.box-foot`, mají výšku podle vlastního obsahu, tedy `auto`.

Prostřední buňka připadající na prvek `.box-body` má pokaždé výšku `20rem`. Zde zejména proto, že hromad lívanců máme vždy několik vedle sebe a nechtěli bychom, aby některé byly vyšší než jiné.

CodePen: vrdl.in/po18w

Ještě doplním, že v „nastavovacím“ kódu, který je v CodePenech schovaný pod komentářem `/* ETC: */`, abych vám ulehčil práci a umožnil soustředit se na psaní kódu, který se týká layoutu, jsou tentokrát poměrně důležité deklarace:

```
.container {
  display: flex;
  flex-wrap: wrap;
}

.box {
  max-width: 20rem;
  margin: 0 1rem 1rem 0;
}

.box-body {
  padding: 1rem;
  overflow: hidden;
}
```

Kromě okrajů v `margin` a `padding` si prosím povšimněte definice rozvržení flexboxem, umístěné na rodičovském prvku nad jednotlivými hromádkami lívanců. `display:flex` už znáte, ale vlastnost `flex-wrap` jsme v příkladech ještě neměli. Je velmi důležitá, protože prvkům rozvržení umožňuje zalamovat se na další řádky.

Všimněte si i deklarace `overflow:hidden`, která obsahu v prvku `.box-body` zabraňuje v přetečení mimo vyhrazený prostor výšky `20rem`. Zde byste samozřejmě mohli namítnout, že lepší než fixní omezení obsahu by zde bylo mít jakousi vnitřní mřížku, která prvky v jednotlivých hromadách lívanců pěkně propojí a zajistí výšku podle nejvyššího z nich. Řešení se jmenuje subgrid, píšu o něm v páté kapitole, ale v době psaní zatím není podporováno všemi prohlížeči.

Máte? Tak pojďme dál.

Svatý grál mezi layouty

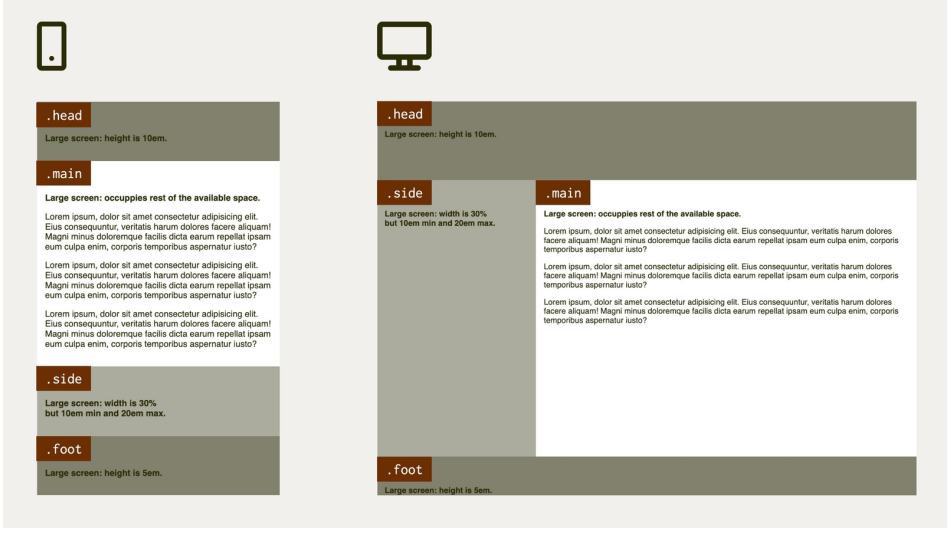
V tomto speciálním příkladu ze světa známých rozvržení začneme citací z Wikipedie:

> **Jako svatý grál bývají označovány nedosažitelné předměty nebo myšlenky, které se snaží člověk vyzkoumat nebo rozluštit, čímž by měl dosáhnout zásadního pokroku.**

Svatým grálem medicíny je například nalezení léku na rakovinu. Jenže co je „svatý grál“ v oblasti webových layoutů? Nejlépe to opět definuje největší světová encyklopedie:

> **Svatým grálem webdesignu je rozvržení, které má více stejně vysokých sloupců. Je běžně žádané a implementované, ale po mnoho let měly různé způsoby jeho implementace všechny možné nevýhody. Z tohoto důvodu bylo nalezení optimální implementace přirovnáváno k hledání nepolapitelného svatého grálu.**

Pojďme to raději vizualizovat. Svatý grál mezi layouty vypadá takto:



Hurá, našli jsme svatý grál! Potřebujeme jej ale ještě vůbec?

Ano, znáte ho. Je to layout, ve kterém byla ještě před lety vysázena každá druhá webovka.

Toto rozvržení jsme implementovali nejprve pomocí rámů, pak tabulkami, floaty nebo ještě jinak. Bylo to peklo. Vzpomeňte na exkurzi z první kapitoly.

Až nyní jsme se přiblížili k optimálnímu a dostatečně jednoduchému řešení. Jak už správně tušíte, svatý grál umíme nakódovat pomocí CSS gridu.

Historie má však smysl pro ironii a v případě historie vývoje webdesignu tomu není jinak. Zhruba v těch letech, kdy jsme začali mít možnost tento typ layoutu poprvé v historii nakódovat efektivně a elegantně, od něj zároveň designérky a designéři začali masově ustupovat ve prospěch jiných, jednodušších typů layoutů.

I přes to bych se svatým grálem mezi rozvrženími nějaký čas v knížce strávil. Historii zatím nepatří. Například pro tvorbu rozhraní webových aplikací je, na rozdíl od tvorby rozhraní prezentačních webových stránek, „holy grail" pořád nezastupitelný.

A pak… Skvěle se na něm ukazují silné stránky šablonování vlastnostmi začínajícími na `grid-template` v CSS gridu.

> Pokud byste v tuto chvíli chtěli začít kódovat, což vám můžu jedině doporučit, opět nabízím předpřipravený CodePen, který si forkněte – a vzhůru do kódování.
>
> CodePen: vrdl.in/97oj5

HTML jsem vymyslel takto:

```
<div class="container">
  <header class="head">
    <h2>.head</h2>
  </header>
  <main class="main">
    <h2>.main</h2>
  </main>
  <aside class="side">
    <h2>.side</h2>
  </aside>
  <footer class="foot">
    <h2>.foot</h2>
  </footer>
</div>
```

Zadání pro rozvržení

Nejprve je potřeba definovat, jak přesně má layout vypadat a chovat se. Leccos jste asi viděli na obrázku výše. Ještě to ale raději rozepíšu:

- Na malých displejích, zde do šířky okna `599px`, chceme prostě a jednoduše vyskládat všechny prvky layoutu pod sebe dle pořadí v kódu.
- Na větších obrazovkách je layout roztažený na celou viditelnou výšku okna.
- Na výšku bychom rádi drželi tato pravidla: hlavička a patička (prvky `.head` a `.foot`) jsou vysoké přesně `10em` a `5em`. Prostředek zabere zbývající plochu.
- Ve zbývající vnitřní ploše máme dva prvky. Postranní panel (`.side`) má šířku 30 %, ovšem s minimem `10em` a maximem `20em`. Obsahový blok (`.main`) už pak jen dostává zbytek viditelné plochy.

Pakliže si chcete sami zkusit kódovat, teď je ta správná příležitost zavřít knížku a učit se prací.

Čtenáři mezi vámi mohou pokračovat.

Řešení příkladu

Nejprve si napíšeme Media Query a provedeme servisní nastavení:

```
@media screen and (min-width: 37.5em) {
  .container {
    display: grid;
    height: 100vh;
  }
}
```

Rodičovskému prvku `.container` jsme nastavili zobrazovací kontext pro CSS grid a výšku na `100vh`, což je sto procent výšky viewportu (`vh` jako „viewport height“). Layout tak bude vždy roztažený v celém okně.

V tomto případě jsem se rozhodl, že budu používat oblasti (`grid-area`), takže do nich můžu prvky DOMu rovnou umístit:

```
.head {
  grid-area: head;
}

.main {
  grid-area: main;
}

.side {
  grid-area: side;
}

.foot {
  grid-area: foot;
}
```

Jak asi tušíte, toto samo o sobě nic nedělá. Normálně bych začínal od definice šablony layoutu (vlastnost `grid-template`), ale právě to vám chci ukázat jako třešničku na dortu tohoto příkladu. Pojďme si ji dát na talíř:

```
.container {
  grid-template-rows: 10em 1fr 5em;
  grid-template-columns: 30% 1fr;
  grid-template-areas:
    "head head"
    "side main"
    "foot foot";
}
```

Teď si to vysvětleme. První dvě vlastnosti už znáte:

- Pomocí `grid-template-rows` definujeme řádky layoutu. Víme, že hlavička má být vysoká `10em`, patička `5em` a zbytek připadá na prostřední část. Tento zbytek definujeme jednotkou `fr`.
- Vlastnost `grid-template-columns` nám, jak už víte, pomůže definovat směr rozvržení po hlavní ose, tedy po sloupcích. Postranní panel má zabrat 30 % šířky a pak je tu ten zbytek určený pro obsah.

Definované oblasti

Vlastnost `grid-template-areas` ještě možná neznáte, ale v gridu patří k mým nejoblíbenějším. S její pomocí definujeme jména oblastí a jejich umístění v buňkách mřížky.

Pomocí `grid-template-rows` a `grid-template-columns` jsme vytvořili mřížku 3 × 2, o třech řádcích a dvou sloupcích. Do nich teď můžeme pomocí tohoto „ASCII artu“ umísťovat oblasti.

Asi jste si všimli, že oblasti `head` a `foot` zabírají vždy dvě buňky, což je přesně ten layout, kterého potřebujeme dosáhnout.

Pomocí inspekce mřížky ve Firefoxu (nebo Chromu) se nám teď layout krásně vizualizuje i s pojmenovanými oblastmi:

„Holy grail layout“ vizualizovaný ve Firefoxu.

Minimum, maximum, optimum s pomocí funkce clamp()

Vsadím se, že vám celou dobu vrtá hlavou, proč jsem zatím ignoroval fakt, že dle zadání má mít postranní panel (`.side`) šířku 30 % – ovšem s minimem `10em` a maximem `20em`.

Inu, snažím se vám ty třešničky dávkovat tak, abyste se jich nepřejedli. Další sladkou dobrotou je totiž funkce `clamp()`.

```
.container {
  /* … */
  grid-template-columns: clamp(10em, 30%, 20em) 1fr;
}
```

Dělá přesně to, co si myslíte: `30%` v prostředním argumentu je optimální velikost. První a poslední argument dodává minimum a maximum. Podpora této funkce v moderních prohlížečích je výborná, takže jí neváhejte využít.

Mimochodem, funkci `clamp()`, stejně jako její kolegyně `min()` a `max()`, pro zájemce více popisuji v poslední, přílohové kapitole.

Pojďme si to teď zjednodušit, nebo možná zkomplikovat. Záleží, jak se na to budete dívat.

Používáme zkratky

Zaměříme se na kód, kterým definujeme rozvržení:

```
.container {
  grid-template-rows: 10em 1fr 5em;
  grid-template-columns: clamp(10em, 30%, 20em) 1fr;
  grid-template-areas:
    "head head"
    "side main"
    "foot foot";
}
```

Nyní se tento náš kód pokusíme zestručnit. Vlastnosti `grid-template-rows` a `grid-template-columns` můžeme zapsat zkratkou `grid-template`, která definuje oba směry a odděluje je lomítkem. Jako vždy v CSS i zde uvádíme jako první svislý směr, tedy definice řádků layoutu:

```
.container {
  grid-template: 10em 1fr 5em / clamp(10em, 30%, 20em) 1fr;
  grid-template-areas:
    "head head"
    "side main"
    "foot foot";
}
```

Jde to zjednodušit ještě více. Také oblasti definované v `grid-template-areas` můžeme vložit do této zkratky – `grid-template`:

```
.container {
  grid-template:
    "head head" 10em
    "side main" 1fr
    "foot foot" 5em
    / clamp(10em, 30%, 20em) 1fr;
}
```

K řádkům vždy nejprve uvádíme oblasti, pak rozměry řádku, následuje lomítko a zapíšeme rozměry sloupců. Dle mého to není úplně přehledné, ale proti gustu žádný dišputát.

A mohli bychom to zkrátit ještě více. Do zkratky `grid`:

```
.container {
  grid:
    "head head" 10em
    "side main" 1fr
    "foot foot" 5em
    / clamp(10em, 30%, 20em) 1fr;
}
```

Do zápisu `grid` se kromě vlastností pro šablonu (`grid-template`) dají uvádět ještě vlastnosti implicitního gridu (začínající na `grid-auto`). To je už dost specifická a zapeklitá věc, takže ji zde zatím nechám bez vysvětlení. Najdete ho v referenční příručce k CSS gridu.

Je mi jasné, že pro jedny jsem to sice zjednodušil („Hurá, méně kódu!“), jiným jsem zamotal hlavu („Fuj, kód složitý jako assembler.“). Máte však volbu, jak kód v gridu zapisovat, a vyberte si dle svých preferencí.

Ještě vám dlužím finální CSS kód. Nejprve definujeme grid:

```
@media screen and (min-width: 37.5em) {
  .container {
    display: grid;
    height: 100vh;
    grid:
      "head head" 10em
      "side main" 1fr
      "foot foot" 5em
      / clamp(10em, 30%, 20em) 1fr;
  }
}
```

Nyní umístíme prvky do mřížky:

```
@media screen and (min-width: 37.5em) {
  .head {
    grid-area: head;
  }
  .main {
    grid-area: main;
  }
  .side {
    grid-area: side;
  }
  .foot {
    grid-area: foot;
  }
}
```

CodePen: vrdl.in/dljik

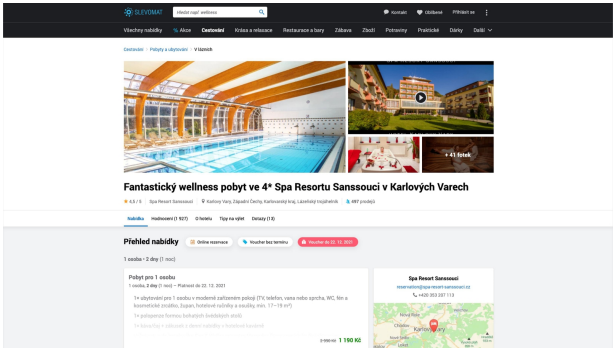
A to je vše. Já už se těším na další příklad. Co vy?

Responzivní fotogalerie

V předchozí ukázce jsme poměrně intenzivně využili oblasti mřížky definované pomocí `grid-template-areas` a umísťované s vlastností `grid-area`.

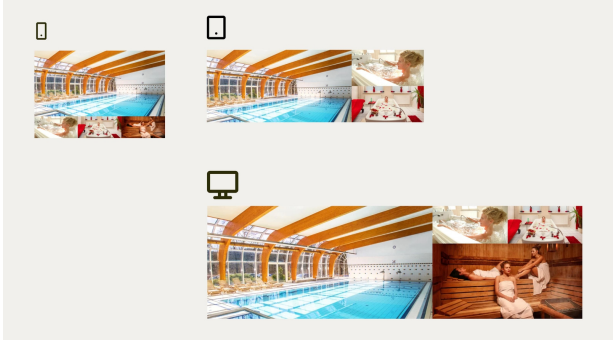
Oblasti jsme zde využili hlavně pro zjednodušení a zpřehlednění kódu. V této ukázce u oblastí zůstaneme, ale jejich využití tady bude ještě intenzivnější a vliv na zpřehlednění kódu větší.

Zdrojem pro tuto ukázku je skutečná fotogalerie na Slevomat.cz. Podobnou jste určitě už viděli i jinde, nebo takovou dokonce sami kódovali.



Co takhle si po tom všem kódování dopřát wellness?

Zadání je ale složitější – tentokrát budeme velmi intenzivně řešit responzivitu, různá zobrazení na různě velkých displejích.



Tři breakpointy, tři různá rozvržení fotogalerie. Tohle asi kodérský wellness nebude, spíše galeje. (Že chybí čtvrtá fotka na tabletu? To není chyba, ale vlastnost.)

> Chcete si to zkusit sami? Forkněte CodePen a vzhůru do toho.
>
> CodePen: vrdl.in/p75na

HTML bude vypadat takto:

```
<div class="gallery">
  <div class="gallery-main-image">
    <img src="…" alt="…">
  </div>
  <div class="gallery-thumb gallery-thumb-one">
    <img src="…" alt="…">
  </div>
  <div class="gallery-thumb gallery-thumb-two">
    <img src="…" alt="…">
  </div>
  <div class="gallery-thumb gallery-thumb-three">
    <img src="…" alt="…">
  </div>
</div>
```

Obrázky, které v této galerii použijeme, musí být v poměru stran 2 : 1. V jiném případě tento layout nebude fungovat, respektive budete si jej muset upravit.

Oblasti mřížky

V dalším kroku si spárujeme jednotlivé prvky HTML (nebo DOMu, chcete-li) s oblastmi gridu.

V CSS kódu to bude vypadat následovně:

```
.gallery-main-image {
  grid-area: main;
}

.gallery-thumb-one {
  grid-area: thumb1;
}

.gallery-thumb-two {
  grid-area: thumb2;
}

.gallery-thumb-three {
  grid-area: thumb3;
}
```

Jak už víte, toto samo o sobě ještě nic nezařídí. Prohlížeč sice ví, kam chcete který prvek umístit, ale oblasti ani layout zatím nezná.

Breakpointy

V příkladu jsem za vás pomocí Media Queries definoval dva body zlomu (400 a 700 pixelů):

```
@media (max-width: 24.9999em) { }
@media (min-width: 25em) and (max-width: 43.7499em) { }
@media (min-width: 43.75em) { }
```

Pojďme teď konečně zapracovat na jednotlivých rozvrženích. Mocně zde využijeme právě zkratku `grid-template`, o které už víte, že jsem si ji poměrně oblíbil.

Layout na nejmenších displejích

Rozvržení pro nejmenší zařízení typu mobily na výšku bude vypadat takto:

```
@media (max-width: 24.9999em) {
  .gallery {
    grid-template:
      "main main main"
      "thumb1 thumb2 thumb3" /
      1fr 1fr 1fr;
  }
}
```

Jde o mřížku 3 × 2, tři sloupce a dva řádky. Každý sloupec je stejně široký – `1fr 1fr 1fr` bychom samozřejmě mohli zapsat pomocí funkce `repeat()` jako `repeat(3, 1fr)`.

Výšku řádků bychom mohli definovat hned za oblastmi. Pokud ji neurčíme, spočítá se podle obsahu, takže má hodnotu `auto`, zde podle výšky obrázků. Jen pro pořádek uvedu odpovídající zápis s definicí výšky řádků layoutu:

```
@media (max-width: 24.9999em) {
  .gallery {
    grid-template:
      "main main main" auto
      "thumb1 thumb2 thumb3" auto /
      1fr 1fr 1fr;
  }
}
```

Všimněte si klíčového slova `auto`, které přibylo na konci obou řádků.

Podstatné na této ukázce je ale umístění oblastí do mřížky:

- Oblast `main` zabírá první tři buňky mřížky, tedy celý první řádek.
- Oblastem `thumb1` až `thumb3` jsme přidělili jednotlivé buňky na třetím řádku.

Díky tomu, že jsem nejprve propojil elementy DOMu s oblastmi, nemusím nyní vůbec přemýšlet, jak vypadá moje HTML. Prostě umísťuji pojmenované oblasti do požadovaného rozvržení.

Layout na středních displejích

Pojďme rovnou na kód:

```
@media (min-width: 25em) and (max-width: 43.7499em) {
  .gallery {
    grid-template:
      "main thumb1"
      "main thumb2" /
      2fr 1fr;
  }
  .gallery-thumb-three {
    display: none;
  }
}
```

Tady musíme poslední náhled „vypnout“ pomocí `display:none`. Grafický návrh nám to káže.

Zde ale musím něco dodat, protože se ve mně staví na zadní odborník na rychlost webu: „Vypnutí“ obrázku pomocí `display:none` nezakáže prohlížeči obrázek stáhnout.

Kdybyste chtěli tento neduh vyřešit, možná si jako já vzpomenete na líné načtení obrázků. Pokud neviditelnému obrázku přidáme atribut a hodnotu `loading="lazy"`, prohlížeč jej v případě, že nebude viditelný, nestáhne. Více o této technice najdete na Vzhůru dolů. vrdl.in/llo

Samotný layout je tentokrát definovaný jako mřížka 2 × 2. První sloupec je dvoutřetinový (`2fr`), druhý zabírá zbylou třetinu prostoru (`1fr`).

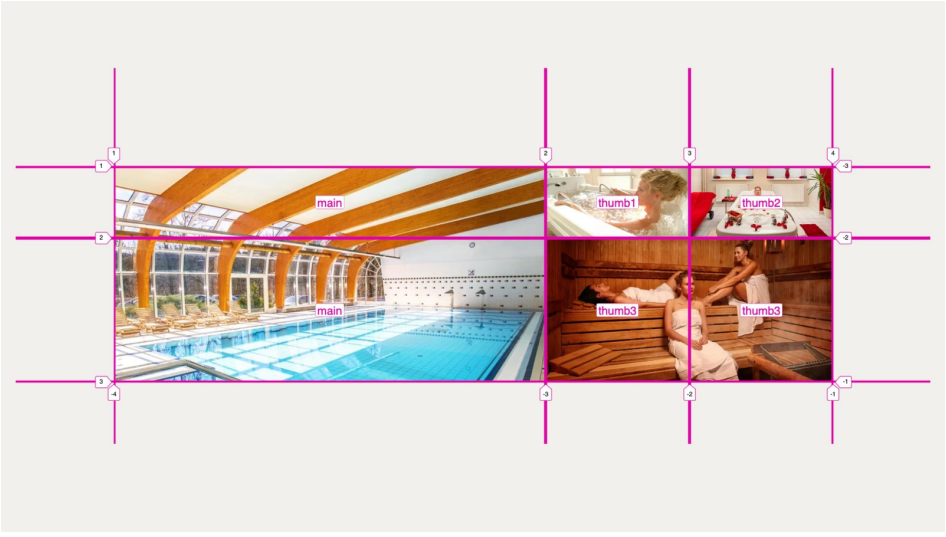
Rozmístění oblastí `main`, `thumb1` a `thumb2` do buněk mřížky asi vidíte z „ASCII artu“. Hlavní fotka zabírá celý první sloupec, další dvě jsou vlevo pod sebou.

Layout na větších displejích

Nyní k největšímu breakpointu. CSS kód vypadá takto:

```
@media (min-width: 43.75em) {
  .gallery {
    grid-template:
      "main thumb1 thumb2"
      "main thumb3 thumb3" /
      3fr 1fr 1fr;
  }
}
```

Máme zde mřížku 2 × 3, dělenou na pětiny. Umístění oblastí do buněk gridu je asi zřejmé. V případě potřeby si je porovnejte s rozvržením z obrázků.



Vizualizace mřížky a jejích oblastí ve Firefoxu.

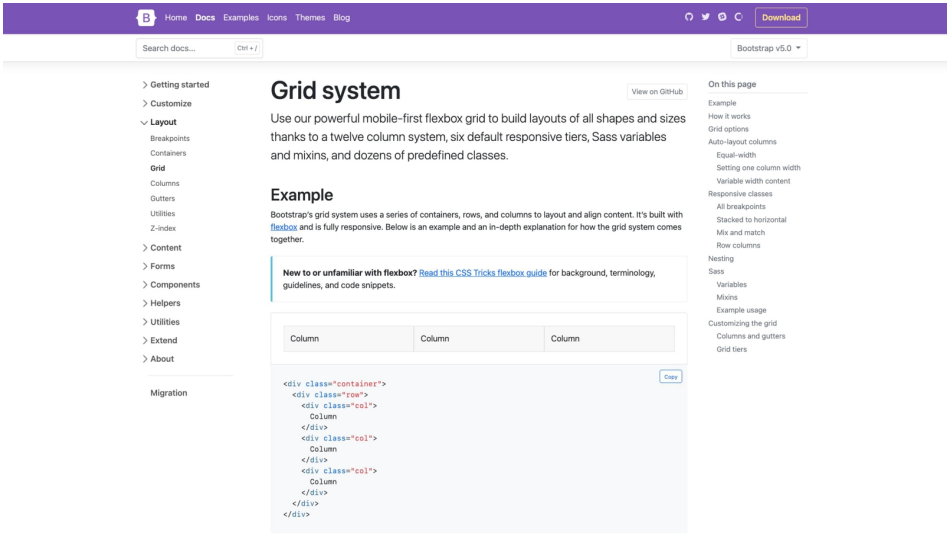
A máme hotovo! Srovnejte si svá díla s mým CodePenem.

CodePen: vrdl.in/7z6ky

Mřížka jako v Bootstrapu

Pokud tento text nečtete v krytu, kde se mnoho let připravujete na zombie apokalypsu, nebo pokud jste se právě teď nedozvěděli, že existuje něco jako webová vývojařina, myslím, že můžu předpokládat, že znáte framework Bootstrap.

Tento frontendový framework se proslavil snadností stavby uživatelského rozhraní, zejména pro vývojáře nebo amatéry, kterým nejsou CSS, HTML a JavaScript úplně blízké.



Tohle určitě vidíte poprvé. Nebo taky ne...

Mnoho vývojářek a vývojářů však ze všech vlastností Bootstrapu používá jen jednu část – mřížku pro tvorbu rozvržení stránky. Tvořit layout pomocí tohoto nebo podobných frameworků je totiž úžasně jednoduché.

Například pomocí následujícího HTML kódu můžeme vytvořit rozvržení o dvou sloupcích třetinové a dvoutřetinové šířky:

```
<div class="row">
  <div class="col col-4">
    <strong>.col-4</strong>
  </div>
  <div class="col col-8">
    <strong>.col-8</strong>
  </div>
</div>
```

Pro neznalé připomínám, že pomocí třídy `.row` se tvoří „řádek“ rozvržení, pomocí `.col` definujeme sloupec a pomocí třídy `.col-` s číslem podíl na výchozí dvanáctisloupcové mřížce. Stačí zapsat HTML, přiložit CSS soubor Bootstrapu a šup – layout máte hotový.

CodePen: vrdl.in/d8l4m

V přiloženém CSS souboru bychom pak viděli, že layout je – alespoň v aktuální páté verzi Bootstrapu – tvořen pomocí flexboxu. Dříve to bylo pomocí `display:inline-block` a ještě dříve díky vlastnosti pro obtékání – `float`.

Proč nepoužít Bootstrap?

Pokud opravdu nebaštíte konzervy v zemljance a neotevřeli jste knížku náhodně právě na této stránce, z dřívějších textů víte, že pro layout typu „mřížka“ by se daleko více hodila… no… mřížka. Překvapivě.

Bootstrap zatím CSS grid nepodporuje, hlavně kvůli kompatibilitě s Internet Explorerem, ale to se v další verzi změní. Můžete však mít i jiné důvody, proč nepoužít pro rozvržení nejpopulárnější webové frameworky.

Prvním problémem může být zbytečných 20–30 kilobajtů CSS, které musíte v podobě Bootstrapu kvůli layoutu stahovat. Jestliže máte přísné požadavky na rychlost načítání stránky, je to určitě nemalá nevýhoda.

Druhým teoretickým problémem je vaše touha mít layout definovaný v CSS, nikoliv v HTML. Dobrých důvodů pro něco takového může být hodně – od nemožnosti upravovat HTML na vašem projektu po specifický způsob organizace frontendového kódu.

Nyní se tedy dostáváme k jádru pudla. Tento layout bychom chtěli napsat v CSS gridu a ušetřit tak stažení zbytečného frameworku.

> Zadání pro vás tedy zní: K HTML uvedenému výše napište CSS pomocí gridu tak, aby rozvržení vypadalo stejně.

Řešení pomocí CSS gridu

Začneme s definicí mřížky na rodičovském prvku `.row`:

```
.row {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  gap: 30px;
}
```

Pomocí funkce `repeat()` vykreslíme 12 prvků mřížky a zápisem vlastnosti `gap` přidáme 30pixelovou mezeru mezi nimi.

Díky autoplacementu, automatickému umístění potomků rodiče do mřížky, se nám oba vnitřní prvky `div` vloží na pozici prvního a druhého sloupečku. To ale nechceme, že?

Další deklarace v CSS budou směřovat k řešení tohoto problému:

```
.col-4 {
  grid-column: 1 / 5;
}

.col-8 {
  grid-column: 5 / 13;
}
```

První prvek jsme umístili mezi první a pátou linku gridu, tedy do prvních čtyř sloupců, druhý mezi pátou a poslední třináctou linku, tedy do dalších osmi sloupců.

Toto řešení je ovšem nepraktické, protože explicitně umísťuje prvky na konkrétní místa mřížky. Co kdybychom například chtěli prvek s třídou `.col-8` umístit před prvek `.col-4`? Výše uvedeným řešením ztrácíme flexibilitu, kterou nám poskytuje Bootstrap, což je krok zpět.

Proto sáhneme k lepšímu řešení:

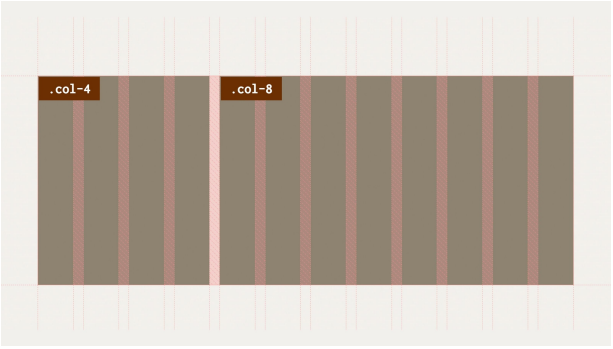
```
.col-4 {
  grid-column: span 4;
}

.col-8 {
  grid-column: span 8;
}
```

Klíčové slovo span specifikuje počet sloupečků, který v mřížce daná buňka zabírá. Vykreslovat se začne tam, kde končí předchozí prvek. A to je přesně ta flexibilita, kterou jsme chtěli mít.

Shrňme si to v kompletním CSS kódu:

```
.row {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  gap: 30px;
}

.col-4 {
  grid-column: span 4;
}

.col-8 {
  grid-column: span 8;
}

.col {
  border: 1px solid darkgrey;
}

.container {
  max-width: 1000px;
}
```

CodePen: vrdl.in/93bir



Rozvržení 4/8 jako v Bootstrapu s překryvnou vrstvou vyznačující neviditelné sloupečky gridu a mezery mezi nimi.

Určitě si umíte představit i možnou náhradu dalších variant používání layoutu v Bootstrapu, včetně responzivních tříd pro různě široké displeje.

Pokud takovou představivost nemáte, namíchal jsem pro vás ještě jeden velmi chutný CodePen s dalšími ukázkami.

CodePen: vrdl.in/e2f5h

Zpráva, která by se vám na konci tohoto příkladu měla zapsat do paměti, zní: Pokud hledáte jednoduchou možnost tvorby layoutu v mřížce jako nativně podporovanou alternativu k Bootstrapu a podobným frameworkům, zvažte CSS grid.

Centrování boxu

Zatímco na vyrovnání textu na střed s pomocí CSS nic složitého není, při centrování prvků layoutu jsme se ve stylech dlouho drbali levou nohou za pravým uchem. Před příchodem flexboxu a gridu to nic příjemného nebylo.

V tomto jednoduchém příkladu otevřeme bránu do světa zarovnávání boxů, který má na starosti CSS Box Alignment Module.

Cílem je to, co vidíte na obrázku. Boxík, který ve viditelné ploše obrazovky zůstane vycentrovaný vodorovně i svisle, i když se jeho obsah změní.

Centruj, centruj, vykrúcaj!

Opět dám chvilku těm z vás, kteří si chcete řešení vymyslet sami. Jeho podmínky jsou tyto:

- Prvek `.item` je centrovaný vodorovně i svisle uprostřed stránky.
- Maximální šířka `.item` je `20em`, výšku vždy určuje obsah prvku, který může být proměnlivý.

Toto je výchozí HTML:

```
<div class="container">
  <div class="item">
    Jsem uprostřed!
  </div>
</div>
```

> Také CodePen se zadáním už čeká na váš fork a následné procvičování mozkových závitů. Jen pozor, použijte jen vlastnosti ze specifikací nového layoutu – gridu, flexboxu a zarovnávání boxů.
>
> CodePen: vrdl.in/8mr7c

Čtenáři mezi vámi nechť směle pokračují dál.

Středění boxu v boxu a poněkud komplikovaný flexbox

Začneme jednodušší variantou příkladu, ve které budeme centrovat boxík uvnitř jiného boxu.

K HTML výše přidáme zhruba toto CSS, které doplňuje zadání o omáčku kolem:

```
.container {
  min-height: 10rem;
  padding: 1rem;
}

.item {
  padding: 1em;
  max-width: 20em;
}
```

Jak vidíte, přidali jsme vnitřní okraje (`padding`), nastavili minimální výšku pro rodiče (`.container`) a maximální šířku dle zadání pro samotný centrovaný boxík (`.item`).

Ve skutečnosti je ve výsledném CodePenu stylů více. Slouží ale jen k nastavení vzhledu, s rozvržením nemají nic společného.

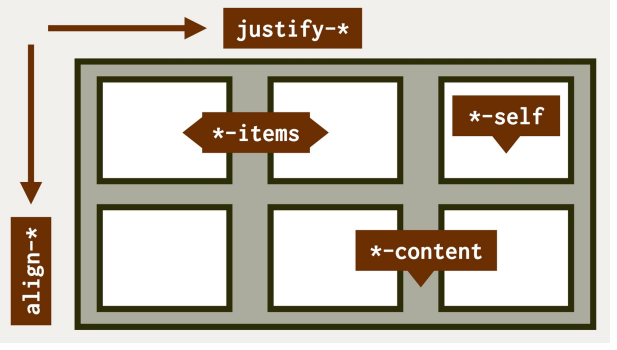
Náš první centrovací pokus učiníme s pomocí flexboxu:

```
.container {
  display: flex;
}
```

Je nám jasné, že toto žádné zázraky neučiní. Spíše naopak – připraví nám to problémy. Boxík je teď roztažený na celou výšku rodiče.

Toto je častá potíž začátečníků s flexboxem. Výchozí hodnota vlastnosti, která určuje zarovnání na výšku, je totiž nastavená na maximální rozpínavost.

O jakou vlastnost jde? Tady máme dobrou příležitost ukázat si tahák, který převezmeme z referenční příručky k CSS Box Alignment.



CSS Box Alignment v celé své složité kráse.

Vidíme, že vlastnosti ve výchozím směru layoutu (vodorovně) začínají na `justify-` a ty v příčném směru (ve výchozím nastavení svisle) na `align-`. Snad i chápeme, že dotčená vlastnost se týká položek, takže hlavním podezřelým je zde vlastnost `align-items`.

Ano, je to tak. Ve flexboxu má výchozí hodnotu `stretch`, která je příčinou onoho roztažení na výšku.

My však chceme zarovnávat doprostřed:

```
.container {
  display: flex;
  align-items: center;
}
```

Povedlo se nám zařídit centrování ve svislém směru. Teď ještě vodorovně, že ano... Podívejte se do taháku na obrázku. Patrně zjistíte, že ve vodorovném směru (`justify-`) potřebujete zarovnávat položky (`-items`).

Jenže znovu ouha! Ve světě flexboxu vám v tuto chvíli `justify-items:center` fungovat nebude. Nejen hodnota, ale rovnou vlastnost `justify-items`, a tedy i `justify-self` je ve specifikaci i prohlížečích pro použití ve flexboxu proza

tím zapovězena. Pokud vás to zajímá více, mrkněte se na podkapitoly o těchto vlastnostech v příručkách dále.

Tento typ zarovnávání se ve flexboxu dělá starým dobrým `margin:auto`:

```
.container {
  display: flex;
  align-items: center;
}

.item {
  margin: auto;
}
```

CodePen: vrdl.in/rfbc0

Můžeme s tím být spokojení, protože naše řešení funguje. Ale víte, co vám řeknu? Já s tím spokojený nejsem. Hlavně proto, že znám lepší řešení pomocí gridu, které není tak komplikované.

Než k němu přejdeme, zmíním ještě jednu možnost, poměrně populární u zkušených kodérů: k centrování využijte vlastnost `justify-content`:

```
.container {
  display: flex;
  align-items: center;
  justify-content: center;
}
```

Tohle řešení bude v řadě případů pro centrování z obou stran ve flexboxu fungovat perfektně.

O jeho „čistotě“ se ovšem můžeme dlouho přít. Zkušenější již mohou `justify-content` dobře znát, a tedy se při jeho použití ve složitějších příkladech nenapálit.

Pro méně zkušené může být nepříjemný fakt, že vlastnost neslouží k zarovnání položek, ale prostoru mezi nimi. Tohle vás může někdy doběhnout.

CodePen: vrdl.in/2wyrl

Teď ale už pojďme na ten elegantní grid.

Středění boxu v boxu: situaci zachraňuje grid

Asi už víte, že zarovnávací vlastnosti CSS Box Align, jako je `align-items`, můžete s drobnými výjimkami použít jak pro flexbox, tak pro grid. V gridu jsou však tyto vlastnosti k dispozici řekněme v plné míře.

Naše původně zvažované řešení pro zarovnání položek na výšku i šířku by zde krásně fungovalo:

```
.container {
  display: grid;
  align-items: center;
  justify-items: center;
}
```

Já ale znám ještě elegantnější možnost:

```
.container {
  display: grid;
  place-items: center;
}
```

Vlastnost `place-items` je totiž zkratkou pro nám již známé `align-items` a `justify-items`. Poskytuje tedy možnost zarovnávat v obou směrech, což je pro naši potřebu úplně dokonalé.

CodePen: vrdl.in/x0wkl

Vlastnost `place-items` má u layoutů tvořených gridem samozřejmě podporu ve všech třech aktuálních prohlížečích.

Středění boxu ve stránce

Zatím jsme se zaměřovali na zarovnávání doprostřed rodičovského prvku. Zadání pro tento příklad ovšem hovořilo o centrování uvnitř viditelné části stránky.

Předpokládáme tedy HTML tohoto typu:

```
<body>
  <div class="item">
    Jsem uprostřed!
  </div>
</body>
```

Řešení by mohlo být nasnadě:

```
body {
  display: grid;
  place-items: center;
}
```

Pokud si to takto vyzkoušíte v prohlížečích, zjistíte, že na větších displejích vás boxík jaksi neposlechne a o centrování na výšku nemůže být řeč.

Problém je v povaze výšky prvku `<body>`, která je vždy určená obsahem. Pokud tedy máme ve stránce málo obsahu, tělo stránky je nižší než celý viewport.

Můžeme jej roztáhnout s pomocí jednotky `vh`, viewport height, na sto procent výšky viewportu:

```
body {
  display: grid;
  place-items: center;
  height: 100vh;
}
```

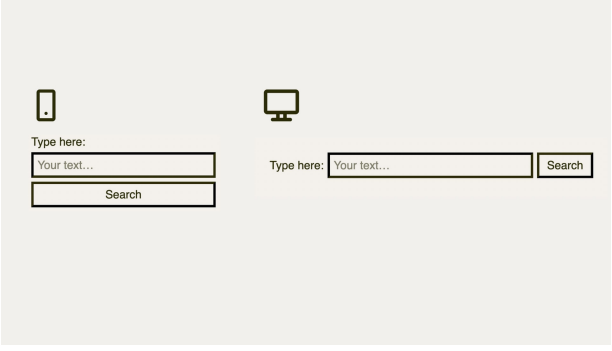
Obsah se bude centrovat na výšku i šířku. V CodePenu si pak můžete vyzkoušet přímo v prohlížeči editovat obsah prvku `.box`. I při výrazném bobtnání obsahu bude prvek stále uprostřed.

CodePen: vrdl.in/d2yht

Na tomto příkladě jsme si mohli vyzkoušet základy zarovnávání pomocí vlastností modulu CSS Box Alignment. Jak jste viděli, k dispozici jsou ve flexboxu i gridu, ale ten druhý k nim má řekněme plný přístup a pro zarovnávání tedy bývá efektivnější použít právě mřížku.

Formulář pro vyhledávání

V této ukázce zkusíme nakódovat jednoduchý vyhledávací formulář. Vyzkoušíme opět něco ze zarovnávání boxů (CSS Box Alignment) a oba systémy layoutu – flexbox i grid. Zadání přichází s návrhovým vzorem, na který jednou narazí každá kodérka i každý kodér.



Co budeme vyhledávat dneska? Třeba řešení tohoto příkladu?

Jak asi vidíte, jde o vyhledávací formulář obsahující tři prvky – textový popisek `<label>`, textový vstup `<input>` a tlačítko `<button>`.

HTML vypadá takto:

```
<form class="form">
  <label for="input" class="form__label">Type here:</label>
  <input size="10" class="form__input" id="input"
     type="text" placeholder="Your text…">
  <button class="form__button">Search</button>
</form>
```

Zadání je následující:

- Na menších displejích (do `499px`, tedy `31.24999em` šířky) budou prvky jednoduše pod sebou.
- Na větších velikostech obrazovky (od `500px` šířky, `31.25em`) se prvky vyskládají vedle sebe.
- V druhém případě mají popisek a tlačítko šířku obsahu a textový vstup se roztahuje, aby vyplnil maximální šířku.
- V obou případech je mezi prvky mezera o polovině `rem`.

> Pokud chcete kódovat sami, což vám doporučuji, neváhejte odložit knížku nebo čtečku s e-bookem a vrhněte se na CodePen se zadáním.
>
> CodePen: vrdl.in/hvuif

Spolu s ostatními budeme pokračovat.

Řešení flexboxem

Nejprve si připravíme jeviště pro tvorbu layoutu:

```
.form {
  display: flex;
  align-items: baseline;
  gap: 0.5rem;
}
```

Rodičovský prvek musí být flexboxem, a tak pomocí vlastnosti `display` nastavíme hodnotu `flex`.

Snad je evidentní, že pomocí `gap:0.5rem` přidáváme mezery mezi jednotlivé prvky layoutu.

Zarovnání na účaří

Vlastnost `align-items` nás možná přinutí si vzpomenout na tahák z předchozího příkladu. Zde chceme zarovnávat položky layoutu (`-items`) na příčné ose (`align-`).

Hodnota `baseline` je velmi zajímavá – umožní nám zarovnávat prvky na *účaří*. Jde o typografický pojem, pod nímž si můžete představit neviditelnou linku, na kterou se zapisují jednotlivé znaky, asi jako jste to dělali ve škole do sešitu. Jinak se jí ještě říká *základní dotažnice*, případně anglicky *baseline*.

Typografické elementy, jako je text a formulářové prvky typu vstupní pole nebo tlačítka, je pro lepší čitelnost vhodnější zarovnat právě na účaří, nikoliv třeba doprostřed.

Menší displeje

Na menších displejích chceme, aby se všechny tři prvky vyskládaly pod sebe. Toho v případě flexboxu dosáhneme změnou směru layoutu:

```
@media (max-width: 31.24999em) {

  .form {
    flex-direction: column;
  }

  .form__label,
  .form__input,
  .form__button {
    width: 100%;
    box-sizing: border-box;
  }

}
```

Moje komentáře:

- Vlastností `flex-direction` a hodnotou `column` měníme směr rozvržení z výchozího vodorovného na svislý.
- Pomocí `width:100%` roztáhneme všechny prvky na celou šířku a pomocí `box-sizing:border-box` prohlížeči sdělíme, že šířku chceme počítat i s vnitřním okrajem a rámečkem. Pokud tu druhou vlastnost neznáte, podívejte se na Vzhůru dolů. vrdl.in/cssbs

Větší displeje

Kód této části bude o něco jednodušší. Přirozené rozvržení, které nám vznikne nastavením `display:flex`, zde jen trochu upravíme.

```
@media (min-width: 31.25em) {

  .form__input {
    flex: auto;
  }

}
```

Vlastnost `flex` je zkratkou pro nastavení velikosti a faktoru zvětšování či smršťování položky flexboxu. Hodnotou `auto` povolíme rovnoměrné zvětšování i smršťování a nastavíme šířku podle potřeby obsahu. Prvky `.form__label` a `.form__button` se tedy roztahovat nebudou, prvek `.form__input` ano.

Celý kód flexboxového řešení je zde:

```
/* Flexbox layout: */

.form {
  display: flex;
  align-items: baseline;
  gap: 0.5rem;
}

@media (max-width: 31.24999em) {

  .form {
    flex-direction: column;
  }

  .form__label,
  .form__input,
  .form__button {
    width: 100%;
    box-sizing: border-box;
  }

}

@media (min-width: 31.25em) {

  .form__input {
    flex: auto;
  }

}
```

CodePen: vrdl.in/atpbj

Pojďme si ale položit otázku, zda by i zde nebylo vhodnější vyřešit úkol pomocí mřížky. Přijímám sázky. Umožní nám grid více sexy řešení?

Řešení gridem

Šance flexboxu porazit grid u tohoto příkladu se myslím sníží už zápisem kódu pro rodičovský prvek, který má rozvržení držet:

```
.form {
  display: grid;
  gap: 0.5rem;
  align-items: baseline;
}
```

Vysvětlivky:

- `display:grid` nastaví formátovací kontext gridu (více v páté kapitole), ale žádné rozvržení sám o sobě nevytvoří.
- Zarovnávání na účaří pomocí `align-items:baseline` můžeme samozřejmě zapsat i do gridového layoutu, protože vlastnosti CSS Box Alignment jsou univerzální.

Menší displeje

Kód pro menší viewporty je díky propracovanosti gridu velmi jednoduchý:

```
@media (max-width: 31.24999em) {
  .form {
    grid-template-rows: repeat(3, auto);
  }
}
```

Co jsme zde udělali?

- Šabloně layoutu (vlastnosti `grid-template-`) nastavíme směr do řádků (`-rows`).
- V layoutu pak máme tři prvky a každý si ponechá šířku podle obsahu, což definujeme klíčovým slovem `auto`.
- Abychom nemuseli zapisovat `grid-template-rows:auto auto auto`, použijeme funkci `repeat()`, která nám to zápisem `repeat(3, auto)` usnadní.

Je to jednoduché, že? Ale teď vám řeknu, že vás u této ukázky trošku tahám za nos. Mohlo by to totiž být ještě jednodušší.

Ve skutečnosti bychom zde nemuseli zapisovat žádný kód. Možná už totiž víte, že CSS grid vytváří řádky layoutu. Při automatickém vytvoření řádku se prvek roztáhne na celou šířku (což chceme) a výšku má podle obsahu (což také chceme). Takže si klidně ušetřeme práci a pro malé displeje tuto deklaraci zcela vynechme.

Větší displeje

Na větších displejích už nějaký ten layout máme, takže si jej pojďme zapsat:

```
@media (min-width: 31.25em) {
  .form {
    grid-template-columns: auto 1fr auto;
  }
}
```

Poznámky:

- Šabloně layoutu (vlastnosti `grid-template-`) nastavíme směr do sloupečků (`-columns`), tedy vodorovný.
- První a třetí člen (zde `.form__label` a `.form__button`) mají mít šířku podle obsahu – to obstará klíčové slovo `auto`.
- Prostřední prvek (`.form__input`) se má roztáhnout do zbylého prostoru – k tomu využijeme zlomkovou jednotku `fr`.

Kompletní rozvržení bychom tedy gridem zapsali takto:

```
.form {
  display: grid;
  gap: 0.5rem;
  align-items: baseline;
}

@media (min-width: 31.25em) {
  .form {
    grid-template-columns: auto 1fr auto;
  }
}
```

Není tohle opravdu sexy?

CodePen: vrdl.in/9g56h

Došli jste ke stejnému, anebo jinému řešení? V čem je to vaše lepší? Klidně se mi ozvěte.

Media nebo Container Queries?

Tady se samozřejmě musím pozastavit u využití dotazů na média – Media Queries. V předchozí kapitole vcelku jasně píšu, že ty se hodí hlavně pro rozvržení celé stránky.

Jenže v tomto případě jsme se zaměřili na jednu malou komponentu. Tu navíc mohou naši zlovolní vývojářští a designérští kolegové umísťovat do různých míst stránek, přesně podle dnešního způsobu práce se systémy komponent.

Zde by se prostě daleko víc hodilo namísto dotazů na média a klíčového slova `@media` použít dotazy na rodičovský element a slovo `@container`. Jak ale už také víte, Container Queries v době psaní zatím nemají plnou podporu.

Co jsme se naučili?

Rád bych tady ještě shrnul, co jsem se vás pokusil tímto příkladem naučit, nebo vám to alespoň připomenu:

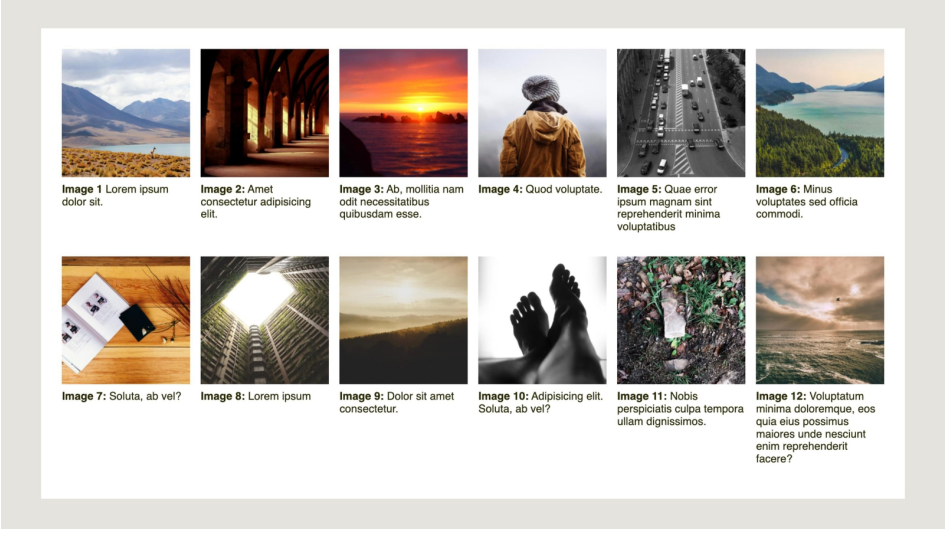
1. Pro rozvržení jednoduše skládaná do jednoho nebo druhého směru, ve kterých se hodně pracuje s mezerami, se více hodí CSS grid.
2. Dozvěděli jsme se, že grid umí automaticky vytvářet řádku a členy těchto řádků roztahuje do plné šířky, takže to nepotřebujeme dělat ručně jako u flexboxu.
3. Naučili jsme se vlastnost `align-items` a hodnotu `baseline`, velmi užitečnou pro vyrovnávání typografických elementů.
4. Zpochybnili jsme používání Media Queries pro rozvržení komponent, ale to je tak všechno, co s tím teď můžeme prakticky dělat.

Zbývá nám ještě pár příkladů, takže neváhejme a pojďme dál.

Galerie bez Media Queries

Fotogalerie, nebo prostě seznam položek vyskládaný do mřížky, to je návrhový vzor, kterým se dnešní weby jen hemží a ve vývojářské praxi se mu nedá vyhnout.

V tomto specifickém příkladu předpokládejme kodéra velmi lenivé nátury a pokusíme se vyhnout jakýmkoliv Media nebo Container Queries. V knize jsme o tomto přístupu psali v druhé kapitole jako o No Queries layoutech.



Tohle nakódujete za chvilku. Co ale responzivně a bez Media Queries?

Zadání je následující:

- Máme 12 náhledů fotek ve stránce.
- Maximální šířka jedné fotky je `300px`.
- Fotky mají popisky, které se mohou roztáhnout do více řádků.
- Mezera mezi jednotlivými sloupci layoutu má šířku `1rem`.
- Fotky musí co nejpravidelněji vyplnit prostor na výšku i šířku.
- Nepoužijte přitom Media Queries.

HTML vypadá zhruba takto:

```
<div class="container">
  <div class="gallery-item">
    <img src="image.jpg" alt="…" class="gallery-item-image"><p class="gallery-item-caption">
      <strong>Image 1</strong> Lorem ipsum dolor sit.
    </p>
  </div>
  <div class="gallery-item">
    <img src="image.jpg" alt="…" class="gallery-item-image"><p class="gallery-item-caption">
      <strong>Image 2:</strong> Amet consectetur adipisicing elit.
    </p>
  </div>
  …
</div>
```

> Píše se vám už v hlavě kód? Vlastně to nemusí být nic složitého, takže vám zde rovnou poskytnu výchozí CodePen se zadáním.
>
> CodePen: vrdl.in/78cnp

Pracanti nechť zavřou knihu a v klidu pracují. My ostatní si rozdělíme funkce takto: já píšu, vy čtete.

Řešení pomocí CSS Multi-column Layout

Pokud vás pro tyto případy napadl jako vhodný systém pro rozvržení CSS Multicol, byli jste na správné cestě. I já bych po něm při řešení určitých typů fotogalerií sáhl. Je to totiž hotové raz dva:

```
.container {
  columns: 150px;
  gap: 1em;
}
```

Vysvětlím to:

- Vlastností `columns`, což je zkratka pro nastavení počtu sloupečků nebo jejich doporučené šířky, zde jako šířku nastavuji polovinu šířky sloupce.
- S pomocí druhé vlastnosti, `gap`, nastavuji šířku mezi sloupečky. To už znáte.

Chybí mi tady ještě jedna věc, a to zákaz zalomení uvnitř prvku `.gallery-item`, protože bychom velmi neradi viděli popisek z prvku `.gallery-item-caption` mimo samotný obrázek, třeba hned na začátku následujícího sloupce:

```
.gallery-item {
  break-inside: avoid;
}
```

Také vlastnost `break-inside` si můžete prostudovat v referenční příručce na konci knížky. Celé řešení si pak můžete projít a otestovat v CodePenu.

CodePen: vrdl.in/pvyf7

Zamysleme se však také nad nevýhodami volby vícesloupcového layoutu pro kód zadání.

- Náhledy fotografií se řadí do sloupců pod sebe. Takže druhá fotka bude pod první, nikoliv vedle ní.
- Položky s fotkami netvoří mřížku, ale jsou volně vyskládané, přičemž roli hraje celková výška položky, určená také délkou textu popisku.

Ve spoustě situací to ale může být vhodné řešení, na obrázku uvidíte, že vizuálně působí dobře.

Možná si při pohledu na obrázek vzpomenete na „zděný layout“ (masonry), o kterém píšu v kapitole s referenční příručkou ke gridu a který bude zřejmě jednou možné udělat bez JavaScriptu, s přímou pomocí CSS gridu. No a když už jsme u gridu, pojďme si tu fotogalerii zkusit ještě navrhnout s ním.

Řešení pomocí CSS grid a RAM

Pokud bychom nechtěli použít Media Queries, v CSS gridu máme možnost sáhnout po návrhovém vzoru RAM. Nejprve kód a pak vysvětlení:

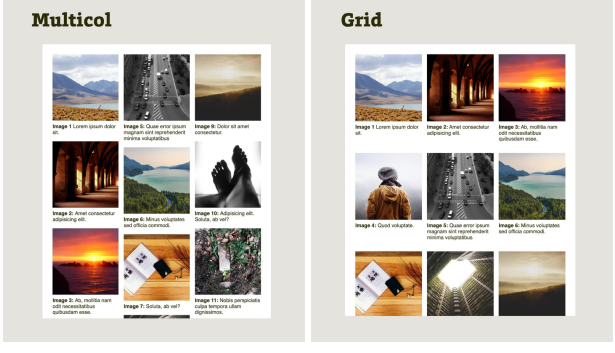
```
.container {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
  gap: 1em;
}
```

RAM, to je zkratka pro Repeat, Auto, Minmax. V popisování začínám zevnitř závorek:

- Funkci `minmax()` jsem zde už vysvětloval. Tady ji využíváme k nastavení minimální šířky elementu na `150px` a maximální `1fr`.
- Funkci `repeat()` už také znáte. Zjednodušuje zápis opakovaných údajů v `grid-template`.
- Namísto čísla v prvním argumentu `repeat()` se na nás ale k našemu překvapení kouká klíčové slovo `auto-fit`, které prohlížeči říká, aby prostor vyplnil maximálním možným roztažením buněk, jež už na řádku existují.

Jinak řečeno – zápisem `repeat(auto-fit,minmax(150px,1fr))` říkáme: Dej na řádek libovolný počet položek. Roztáhni je klidně do jejich maximální obsahové šířky, ale nikdy je nesmršťuj pod 150 pixelů.

Výsledek vypadá jinak než v řešení pomocí CSS Multicol, na obrázku uvidíte srovnání.



Kdo je váš favorit, pan Vícesloupec nebo paní Mřížková?

Položky jsou srovnané do řádků a je možné je číst zleva doprava. Nevýhody zde však vidím taky. Na většině šířek displeje náhledy s popisky nevyplní všechny řádky. Algoritmus prohlížeče vykreslí položky po řádcích, takže můžeme například vidět kombinaci 9 položek v prvním řádku a 3 ve druhém.

Udělat pěknou mřížku, bez vynechaných položek, tedy například 2 × 6, 3 × 4 apod., je samozřejmě s gridem možné, jen bychom to museli specifikovat přesněji s pomocí Media Queries.

Ještě odkaz na CodePen s řešením využívajícím grid a můžeme to uzavřít.

CodePen: vrdl.in/b2o1u

Co jsme se tady naučili?

- Poprvé jsme v kapitole s příklady využili CSS Multi-column Layout, a to v situaci, kde funguje velmi dobře – design fotogalerie.
- Vyzkoušeli jsme si dva layouty s „automatickou responzivitou“, bez použití Media Queries.
- V rámci CSS gridu jsme se něco dozvěděli o návrhovém vzoru RAM a klíčovém slově `auto-fit`.

Krkavčí technika

V posledním příkladu mírně sejdeme z cesty poznávání CSS layoutů a vrátíme se k jednomu z našich témat – Container Queries versus Media Queries.

Rozebereme si totiž „The Raven Technique“ popsanou Mathiasem Hülsbuschem na CSS-Tricks v roce 2020. Jde o alternativu k žádaným Container Queries, které ale v době psaní této podkapitoly nemají plnou podporu v prohlížečích.

Výhodou této techniky je podpora ve všech moderních prohlížečích. Nevýhodou je krkolomnost.

Zkuste sedmkrát za sebou říct „krkolomný krkavec“. Přesně tak jsem si připadal, když jsem o té technice četl poprvé.

Posuďte to sami z ukázky kódu:

```
.item {
  width: --dyn_length: calc(
    min(var(--is_wide) * var(--very_big_int), var(--length_4_wide))
    + min(var(--is_medium) * var(--very_big_int), var(--length_4_medium))
    + min(var(--is_small)  * var(--very_big_int), var(--length_4_small))
  );
}
```

Žádné strachy, vše ještě poctivě vysvětlím. Jen jsem chtěl demonstrovat, jak komplikovaná tato technika je.

Raven Technique ale nechci úplně shazovat. Pokud něco jako podmínku `@container` z Container Queries na vašem projektu zoufale potřebujete, věnujte mi ještě chvíli pozornosti.

Krkavčí technika je založená na matematických funkcích a dalších novinkách v CSS:

- Funkce `calc()` umožní vložit namísto hodnoty matematický výraz.
- Funkce `min()` a `max()` vracejí nejnižší, respektive nejvyšší hodnotu ze všech uvedených v argumentech.
- Funkce `clamp()` je kombinací `min()` a `max()` pro tříčíselné hodnoty.

- Proměnné v CSS jako `--color:blue` zase umožňují udržovat a měnit hodnoty výpočtu, podobně jako v programovacích jazycích.

Mimochodem, „CSS proměnné“ nejsou ve skutečnosti proměnné. Ne takové, jaké si představí běžná vývojářka nebo vývojář. Píšu o nich více v deváté kapitole, kde také najdete ukázky jejich použití.

Definujeme proměnné

V prvním kroku budování krkavčí techniky si definujeme proměnné se šířkou a s body zlomu layoutu:

```
.breakpoints_1 {
  /* Základní šířka rodiče: */
  --base_size: 100%;
  /* Breakpointy: */
  --breakpoint_wide: 1000px;
  --breakpoint_medium: 500px;
}
```

Breakpointy

Dále definujeme šířky prvků pro tři rozmezí, která vznikla definicí breakpointů:

```
/* Šířka elementů: */
.breakpoints_1 {
  --length_4_small: calc((100% / 1) - 10px);
  --length_4_medium: calc((100% / 3) - 10px);
  --length_4_wide: calc((100% / 4) - 10px);
}
```

Asi je zřetelné, že na nejmenších viewportech (`--length_4_small`) zabere celou šířku jeden boxík (`calc((100% / 1)`), zatímco na největších už budou vedle sebe čtyři (`calc((100% / 4)`).

Dotazy na splnění podmínek pro šířku rodiče

Magie nastává v následujícím kroku. Tady pomocí matematických funkcí plníme proměnné, které indikují, na jakém breakpointu se aktuálně nalézáme:

```
.container {
  --is_wide: clamp(0px,(var(--base_size) - var(--breakpoint_wide)),1px);
}
```

Vysvětleme to. Proměnná `--is_wide` vrací `1px`, pokud je `--base_size` větší než `--breakpoint_wide`. Představte si to `1px` jako `true`. V opačném případě vrací `0px` jako `false`.

V `--base_size` máme uloženou hodnotu `100%`, což je ale skutečná šířka rodičovského prvku. Už chápete?

Podobné to bude pro hodnoty `--is_medium` a `--is_small`.

Definujeme vlastnosti prvku v jednotlivých breakpointech

Proměnné, které indikují breakpoint, můžeme nakrásně využít v dalším kroku, kde nastavujeme šířku pro konkrétní prvky v layoutu:

```
.item {
  width: calc(
    (var(--is_small)  * 100)
  + (var(--is_medium) * 200)
  + (var(--is_wide)   * 500)
  );
}
```

Pokud bude například aktivní proměnná `--is_medium`, vyjde nám zde `200px` (`0px*100 + 1px*200 + 0px*500`).

To by mohlo pro pochopení základního principu The Raven Technique stačit. Je to samozřejmě ještě trochu složitější, sami jste to viděli z úvodní ukázky kódu.

Ale zkušenější z vás to jistě ocení a je fajn, že jsme si udělali výlet do pokročilejších vlastností CSS.

CodePen: vrdl.in/spela

Zájemci o více informací ať se mrknou na celý článek „The Raven Technique: One Step Closer to Container Queries“. vrdl.in/raven

Shrnutí třetí kapitoly

1. **K čemu slouží vlastnost grid-template-columns?**
   a) definuje šablonu pro rozvržení sloupců gridu
   b) definuje šablonu pro rozvržení řádků gridu
   c) vlastnost neexistuje, správně je `grid-template-cols`
2. **Co dělá funkce minmax()?**
   a) vrací největší z atributů a používá se pro šířku a výšku prvku
   b) vrací největší a nejmenší z atributů a používá se pro šířku a výšku
   c) vrací největší z atributů a používá se pro jakoukoliv vlastnost CSS
3. **Jakou funkci má vlastnost grid-area?**
   a) pojmenovává oblast gridu
   b) zajistí umístění prvku do pojmenované oblasti gridu
   c) vrací všechny názvy oblastí v daném layoutu
4. **Co vykreslí zápis grid-template-columns:repeat(12, 1fr)?**
   a) dvanáct sloupečků o šířce `1fr`
   b) vykreslil by dvanáct sloupečků, ale jednotka `fr` neexistuje
   c) je to celé velký chyták, nic takového neexistuje
5. **Jak gridem vycentrovat boxík uvnitř boxu?**
   a) `display:grid;place-items:center;`
   b) `display:grid;align-items:center;`
   c) stačí jen `display:grid`
6. **Co dělá zápis flex:auto?**
   a) vytvoří automatické rozvržení flexboxem pro všechny potomky
   b) povolíme automatické rozpínání i smršťování velikosti položek
   c) povolíme automatické rozpínání velikosti položek
7. **Co vykreslí zápis columns:150px;gap:1em;?**
   a) stopadesát sloupečků o šířce `1em`
   b) sloupečky libovolného počtu o šířce přesně `150px` a mezerou `1em`
   c) sloupečky libovolného počtu o šířce zhruba `150px` a mezerou `1em`

Řešení: 1. a), 2. c), 3. b), 4. a), 5. a), 6. b), 7. c).

04

FLEXBOX

Další stránky jsou věnované referenčním příručkám různých CSS modulů pro tvorbu layoutu.

Nečekám, že je budete číst slovo od slova jako detektivku. Jsou zde k dispozici pro vaši potřebu.

Pokud ale knížku čtete postupně, snažil jsem se i referenční příručky psát tak, aby vás bavilo je číst a hlavně – abyste se vždy dozvěděli něco nového.

Jestliže s flexboxem začínáte, zkuste si projít vše, ale zaměřte se hlavně na vlastnost `flex`.

Zkušenějším zase neuškodí podrobnější zkoumání zákonitostí kolem vlastnosti `flex-basis`, které disponují mnohými ošemetnostmi.

Všechny vlastnosti flexboxu

Flex v češtině znamená pružný, přizpůsobivý. Flexboxy jsou tedy pružné elementy layoutu. Jednou z hlavních předností flexboxu je totiž schopnost vyplňovat zbylý prostor.

Toto je referenční příručka s příklady pro všechny vlastnosti, které ve flexboxu můžete používat.

Vlastnosti flexboxu

Vlastnosti se týkají buď kontejneru, tedy rodičovského prvku, nebo položek.

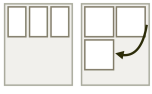
Kontejner flexboxu

Hodnota vlastnosti `display` `flex` nastaví prvku formátovací kontext flexboxu, takže jeho přímí potomci mohou mít specifické vlastnosti.

Můžete také použít „inline“ hodnotu: `inline-flex`, která se chová jako `inline-block`, tedy uvnitř je možné dělat rozvržení, zvenčí jde o součást řádku textu.

---

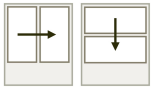
flex-wrap



Mohou se položky zalamovat? Výchozí stav je nezalamovat.
Např. `flex-wrap:wrap` – položky se mohou vykreslit na další řádek.

---

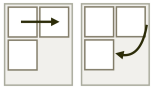
flex-direction



Určí směr toku rozvržení. Výchozí stav je zleva doprava, do řádku.
Např. `flex-direction:column` – položky se skládají shora dolů, do sloupce.

---

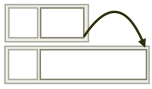
flex-flow



Zkratka pro `flex-wrap` a `flex-direction`.
Např. `flex-flow:wrap column` – položky se zalamují a skládají shora dolů.

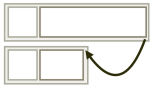
Položky flexboxu

flex-grow



Jak moc může položka růst. Výchozí je `0`.
Např. `flex-grow: 1` – podíl na celku hodnoty `1` z volného prostoru.

flex-shrink

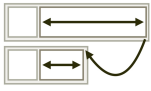


Faktor smršťování položky. Výchozí je `1`.
Např. `flex-shrink:0` – položka se nesmršťuje.

flex-basis

Výchozí velikost položky. Výchozí je `auto`.
Např. `flex-basis:0` – nehledí se na rozměr obsahu ani `width` či `height`.

flex



Zkratka pro vlastnosti `flex-grow`, `flex-shrink` a `flex-basis`.
Např. `flex:auto` – roste, smršťuje se a rozměr je nastavený na `auto`.

Nyní známe vlastnosti a pojďme prozkoumat použití flexboxu v konkrétní situaci.

Základy v jednoduchém příkladu

Představme si triviální třísloupcový layout:

```
<div class="container">
  <p class="col col-1">First is looooooong.</p>
  <p class="col col-2">Second is looooonger.<br>…<br>…</p>
  <p class="col col-3">Third is short.</p>
</div>
```

HTML je jednoduché. O to přísnější máme požadavky na design:

- Všechny sloupce mají být stejně vysoké.
- Layout se při nedostatku místa zalomí.
- První dva sloupce jsou pružné, třetí nikoliv.

- Druhý sloupec se zvětšuje a zmenšuje dvakrát tolik než první.
- Na menších displejích do `400px` se prvky vyskládají pod sebe.

> Zkuste si to nakódovat sami, stačí forknout následující CodePen: [vrdl.in/f18lo](https://codepen.io/machal/pen/BaRYjya).

V dalším textu si ukážeme, jak bych postupoval já.

Všechny sloupce mají být stejně vysoké

Stejná výška prvků platí i v případech, kdy má jeden sloupec delší obsah než zbylé dva. To je to nejjednodušší. Stačí z rodiče pomocí vlastnosti `display` udělat kontejner flexboxu:

```
.container {
  display: flex;
}
```

Layout se při nedostatku místa zalomí

V případě, že zde nebude dost prostoru pro všechny položky, zalomíme. To definujeme pomocí vlastnosti `flex-wrap`:

```
.container {
  flex-wrap: wrap;
}
```

První dva sloupce jsou pružné, třetí nikoliv

Dále jsme si vymysleli, že první dva sloupce se budou zmenšovat a zvětšovat, přičemž druhý dvakrát více než první. Třetí naopak nikoliv, zůstane vždy na svém. Tady pomůže zkratka vlastností položky flexboxu, `flex`:

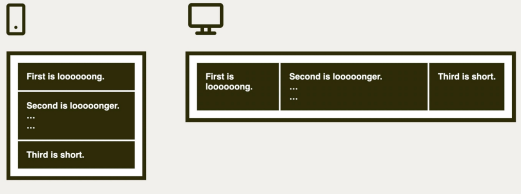
```
/* Pružná položka, zabírá jednu část volného místa: */
.col-1 {
  flex: 1;
}

/* Pružná položka, zabírá dvě části volného místa: */
.col-2 {
  flex: 2;
}

/* Fixně široká položka: */
.col-3 {
  flex: none;
}
```

Na menších displejích se změní směr

Na mobilech zpravidla není pro rozvržení prvků vedle sebe prostor. Přidáme proto změnu směru layoutu, což zajistíme vlastností `flex-direction`:

```
@media (max-width: 25em) {
  .container {
    flex-direction: column;
  }
}
```

Na obrázku je vidět, k čemu jsme se dopracovali:



Je to layout. Je pružný. Je to flexbox.

Výsledný příklad si utíkejte vyzkoušet naživo:

CodePen: vrdl.in/lcew2

Základní pojmy: flexboxový kontejner, položka flexboxu, hlavní a příčná osa

Flexbox tvoří nerozlučná dvojice dvou typů elementů – flexboxový kontejner a položka flexboxu. Flexboxovou položkou se stává každý přímý potomek kontejneru.

```
<ul class="flex-container">
  <li>…</li>
  <li>…</li>
</ul>
```

Flexbox nadefinujeme snadno jen pomocí kontejneru flexboxu:

```
.flex-container {
  display: flex;
}
```

Všechny `<li>` se tady stávají položkami flexboxu.

Kromě flexboxových kontejnerů a položek nás v dalším textu budou zajímat ještě osy. Ukažme si je na zjednodušeném schématu:

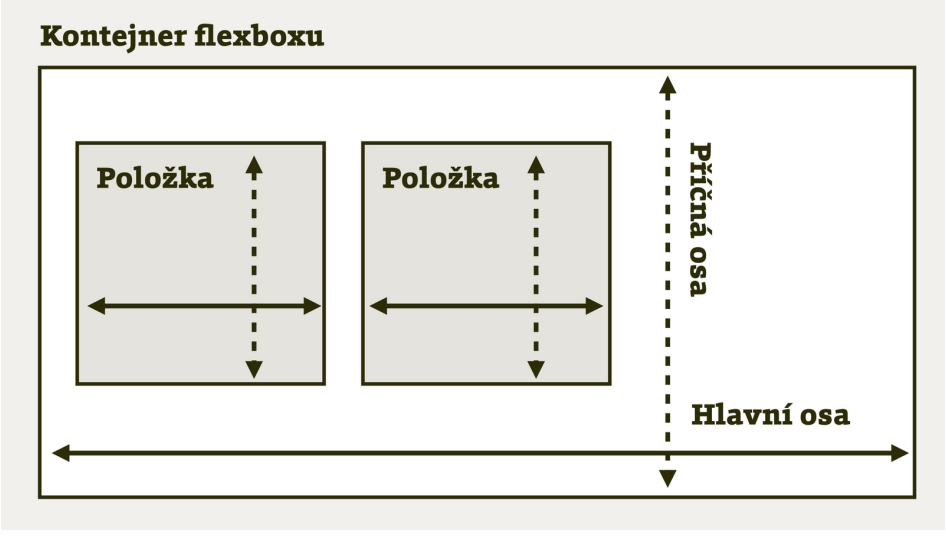
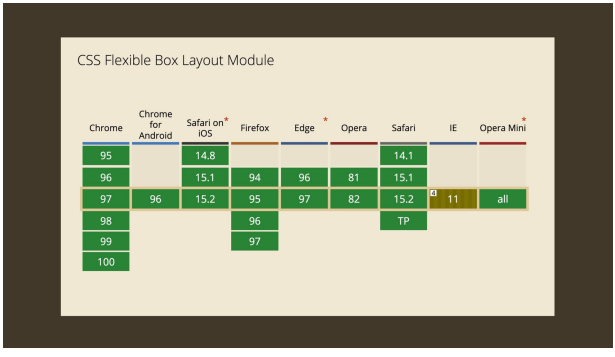


Schéma vnitřních struktur flexboxu.

- kontejner flexboxu – rodičovský element
- flexboxová položka – všichni přímí potomci kontejneru
- hlavní osa – výchozí je vodorovná, ale toto lze změnit
- příčná osa – vždy příčná k hlavní, takže ve výchozí podobě je svislá
- hlavní rozměr – výchozí je šířka, ale řídí se nastavením hlavní osy
- příčný rozměr – výchozí je výška

Podpora

Dostupnost vlastností flexboxu v prohlížečích je velmi dobrá, vždyť s podporou přišel už Internet Explorer 10! Grafy na CanIUse jsou tedy pěkně zelené. caniuse.com/flexbox



Paráda. Podpora flexboxu v moderních prohlížečích je takřka bezchybná. Zdroj: CanIUse.com.

Konkrétní podporu je ale potřeba rozdělit podle jednotlivých vlastností uvedených výše v tabulce. Tam už je situace různorodější.

Bojem s více či méně příjemnými chybami je provázena práce v Internet Exploreru, protože jde o nejstarší dnes sloužící prohlížeč. Detailně se tím zabýváme v textu o podpoře CSS layoutu v osmé kapitole.

Co je dobré o flexboxu vědět

1. Flex je nová hodnota vlastnosti display

Na tomto bodu není na první pohled nic nového. Vyplývá z toho ale, že na rodičovský kontejner ani jeho potomky nebudou mít účinek vlastnosti související s jinými zobrazovacími režimy vlastnosti `display` – `block`, `inline`, `inline-block` a dalšími. Typický příklad jsou vlastnosti `float`, `clear` nebo `vertical-align`.

2. Položky flexboxu neslučují vnější okraje

V kontextu flexboxového rozvržení platí, že na rozdíl od blokových elementů se u sousedících flex položek neslučují vnější okraje, jež jsou dané vlastností `margin`.

3. Položky flexboxu lze pozicovat

Pozicování prvků vlastností `position` s hodnotami `absolute`, `relative`, `fixed` lze na rozdíl od vlastností souvisejících s `float` na flexboxové položky běžně aplikovat.

4. Směr hlavní osy flexboxu řídí jazyk rozhraní

Směr hlavní osy flex kontejneru vychází vždy z vlastnosti `writing-mode`. Pokud bychom tedy flexboxem dělali layout stránky v arabštině, budou hodnoty zmíněné ve vlastnostech jako `flex-direction` naopak.

5. S visibility:collapse pracuje flexbox jako se řádky tabulky

Taková drobnost na závěr. `visibility:collapse` funguje stejně jako u hodnoty `hidden` s výjimkou tabulek a flexboxu. U flex položek (stejně jako u elementů s `display:table-row` nebo `table-column`) prostě prvek není vidět, ale drží se mu místo a v DOMu se s ním počítá.

A teď už se pojďme podívat na jednotlivé vlastnosti flexboxu.

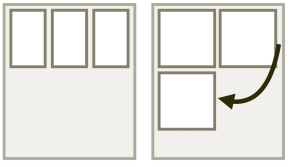
Flex-wrap: zalamování položek

Vlastnost `flex-wrap` aplikujeme na kontejner flexboxu, abychom definovali, zda se položky rozvržení mohou zalamovat na více řádků, nebo nikoliv.

Hodnotou `wrap-reverse` také můžeme otočit pořadí vyskládání prvků na příčné ose layoutu.

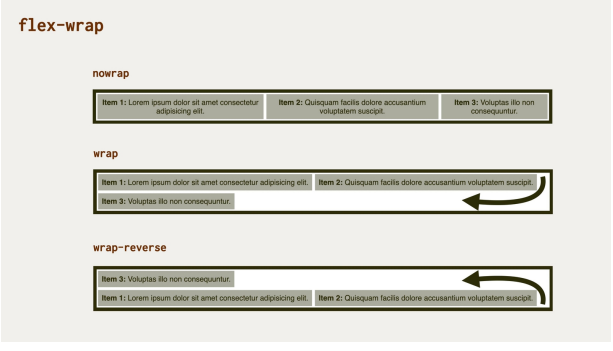
Toto jsou možné hodnoty:

- `nowrap`
  Výchozí hodnota. Flexbox bude jednořádkový.
- `wrap`
  Kontejner flexboxu umožní položkám, aby se zalomily na další řádek.
- `wrap-reverse`
  Položky se mohou zalomit do více řádků a zároveň se pořadí řádků otočí.



Poznámka: všechny texty příruček jsou k vidění naživo na Vzhůru dolů:

→ vrdl.in/cssfw



Může se layout zalamovat? A pokud ano, jakým směrem?

V online ukázce si můžete interaktivně vyzkoušet jednotlivé hodnoty vlastnosti `flex-wrap`:

CodePen: vrdl.in/cuqnb

Nezalamuje hodnota wrap-reverse naopak?

Všimněte si prosím chování hodnoty `wrap-reverse` – položky se vyskládají dle zdola, ale zleva doprava, nikoliv naopak, jak by mohl někdo očekávat.

Dává to ale smysl. Hodnota `wrap-reverse` totiž zalamuje naopak. Poslední položka řádku skočí nahoru a zařadí se nad první položku.

Zkratka flex-flow

Možná už víte, že můžete použít vlastnost `flex-flow`, zkratku pro vlastnosti `flex-direction` a právě `flex-wrap`.

```
.box {
  flex-flow: wrap; /* = flex-wrap: wrap */
}
```

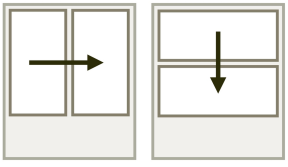
Podpora v prohlížečích

Podpora vlastnosti `flex-wrap` je velmi dobrá a o chybách nevím. Na webu CanIUse se u Internet Exploreru 11 v době psaní uvádí částečná podpora s poznámkou, že to je kvůli velkému počtu chyb. Trošku horší je, že je tento web neuvádí – a dle mého vlastně ani neexistují. CanIUse.com

Flex-direction: směr vyskládání položek

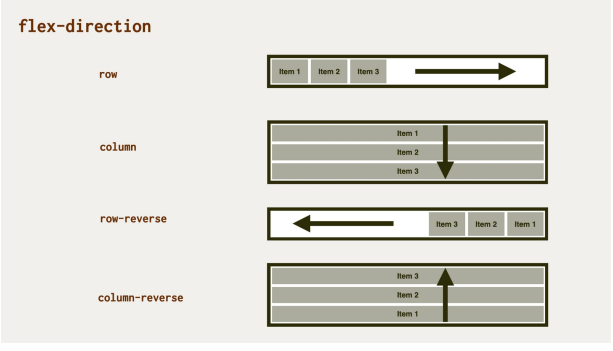
Vlastnost `flex-direction` se aplikuje na kontejner flexboxu a určí způsob vyskládání položek tím, že nastaví směr hlavní osy flexboxu.

Toto jsou možné hodnoty:



→ vrdl.in/cssfd

- `row`
  (Výchozí hodnota.) Vyskládá položky flexboxu do řádky.
- `row-reverse`
  Skládání položek do řádky, ale v opačném směru, tedy zprava doleva.
- `column`
  Položky flexboxu se skládají shora dolů, do sloupce.
- `column-reverse`
  Skládání položek do sloupce, ale v opačném směru, tedy zdola nahoru.



Jakým směrem se flexboxové rozvržení vykreslí?

V živé ukázce je možné interaktivě zkoušet všechny varianty hodnot vlastnosti.

CodePen: vrdl.in/29or1

Zkratka flex-flow

Také u této vlastnosti můžete použít zkratku `flex-flow`:

```
.box {
  flex-flow: column; /* = flex-direction: column */
}
```

Záludnosti změny pořadí

Je potřeba říci, že směr layoutu ve flexboxu vždy vychází ze zvyklostí daného jazyka – u nás tedy zleva doprava, v arabštině zprava doleva, v některých verzích zápisu japonštiny shora dolů. Jak jsem psal jinde, záleží to také na nastavení vlastností `writing-mode` a `direction`.

Mimochodem, v poslední, „přílohové“ kapitole píšu o CSS logical properties, logických vlastnostech a hodnotách a obecně o možnosti sázení dokumentu v jazycích, které mají jiný směr toku než ten náš.

Pořadí vykreslení položek se v případě vlastností `row` a `column` bere z pořadí v kódu. Pokud chcete pořadí otočit, prostě zvolte hodnoty `row-reverse` nebo `column-reverse`. Pozorní si asi všimli, že vlastnost `flex-direction` lze tím pádem použít i pro změnu řazení seznamů.

Změna pořadí má ale vliv jen na výsledek vykreslení, nikoliv např. na pořadí vykreslování a na procházení při navigaci klávesou Tab nebo čtečkami pro slabozraké.

Proto si na změnu směru dávejte velký pozor a vždy si představte, zda „čtení“ dokumentu bude dávat smysl i při pořadí uvedeném v DOMu.

Je to ostatně podobné jako u vlastnosti `order` (viz sedmá kapitola) a více to rozebírám v textu CSS layout a přístupnost (viz osmá kapitola).

Vliv na flex-basis

Pokud je vlastnost `flex-direction` nastaveno na hodnotu `column`, jinou vlastností `flex-basis` nastavujete výšku, tedy vlastnost `height`, nikoliv šířku, `width`.

Podpora v prohlížečích

Vlastnost `flex-direction` funguje dobře. Jedinou mně známou výjimkou je tzv. flexbug #14, kdy prohlížeče při použití deklarací `display:inline-flex`

a `flex-flow:column wrap` neumí správně roztáhnout kontejner do celé šířky položek. Ale to je drobnost, na kterou lidé skoro nenarazí.

Více o „flexbugs“ píšu v textu o chybách v CSS layoutech uvnitř osmé kapitoly.

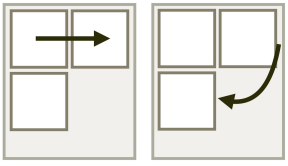
Další informace o podpoře jsou na CanIUse.com.

Flex-flow: zkratka pro určení směru a zalamování

Vlastnost `flex-flow` určuje směr a způsob zalamování položek flexboxu.

Jde o zkratku pro dvě vlastnosti, které jsme uváděli již dříve:

- `flex-direction` – směr vykreslování flexboxového rozvržení.
- `flex-wrap` – zalamování položek layoutu na více řádků.



→ vrdl.in/cssff

V obecné rovině je zápis následující:

```
flex-flow: <hodnota flex-direction> <hodnota flex-wrap>;
```

Vzhledem k tomu, že obě vlastnosti používají jiná klíčová slova pro své hodnoty, je možné je uvádět v libovolném pořadí a samozřejmě úplně v klidu jednu z nich vynechat.

Toto jsou příklady možných hodnot:

- `column`
  Jako `flex-direction:column`. Položky flexboxu se skládají do sloupce.
- `wrap`
  Odpovídá `flex-wrap:wrap`. Kontejner flexboxu umožní položkám, aby se zalomily na další řádek.
- `column wrap`
  Ekvivalentní k `flex-direction:column;flex-wrap:wrap`.
- `row-reverse wrap`
  Ekvivalentní k `flex-direction:row-reverse;flex-wrap:wrap`.

Všechno je to dobře vidět v online ukázce na CodePenu, kde si také můžete měnit velikost písma, šířku viewportu nebo to jinak rozbíjet a při tom se to všechno naučit.

CodePen: vrdl.in/3zji8

Podpora v prohlížečích

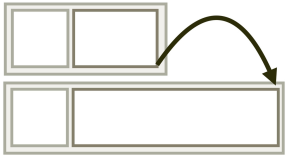
Podpora vlastnosti `flex-flow` je přímo luxusní. Podívejte se na CanIUse a uvidíte samá zelená políčka, včetně Internet Exploreru. CanIUse.com

Jedinou vadou na kráse je kombinace hodnot `display:inline-flex` a `flex-flow:column wrap`, nepříliš známá a nepříliš omezující chyba, kterou jsme se zabývali už u vlastnosti `flex-direction`.

Flex-grow: faktor zvětšování položky

Jak moc může položka růst relativně k dalším položkám, pokud je k dispozici volné místo – například když uživatel zvětší okno prohlížeče? To určuje vlastnost `flex-grow`.

Specifikace a praktická zkušenost nás – namísto používání vlastnosti `flex-grow` – navádějí spíše ke zkratce `flex`, ale přesto považuji za nutné se o této vlastnosti zmínit.



→ vrdl.in/cssfg

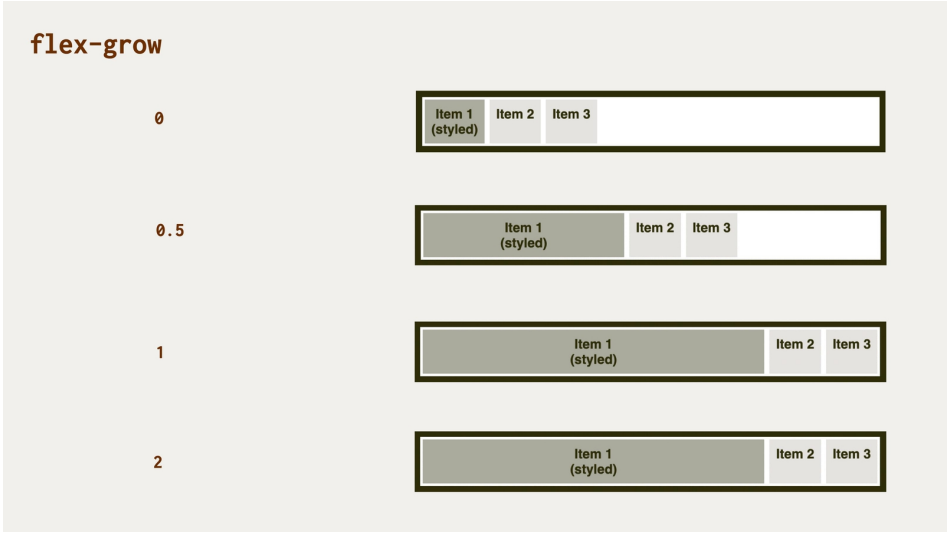
Možné hodnoty:

- `0` (výchozí)
  Znamená, že položka s přibývajícím volným místem nijak neroste.
- Kladná čísla (např. `0.5`, `1`, `2`...)
  Položky si rozdělují podíly z nově získaného místa nad rámec výchozí šířky.

Záporné hodnoty zde nejsou validní, ale desetinná čísla jsou povolená, což už jste viděli.

Demo

V interaktivní ukázce máme kontejner flexboxu (`display:flex`) a první položce přidáváme různě hodnoty vlastnosti `flex-grow`:



Jak se bude položka flexboxu roztahovat do volného místa?

Druhé a třetí položce jsme nenastavili žádnou šířku.

HTML vypadá následovně:

```
<div class="container">
  <p class="box box--one"><strong>Item 1<br>(styled)</strong></p>
  <p class="box box--two"><strong>Item 2</strong></p>
  <p class="box box--three"><strong>Item 3</strong></p>
</div>
```

Ve výchozím stavu se všechny tři položky roztáhnou podle své přirozené velikosti. Pokud bude v rodičovském kontejneru dostatek volného místa, pak si první položka, stylovaná přepínači nahoře, vezme tolik zbývajícího prostoru, kolik jsme jí ochotni dát právě nastavením vlastnosti `flex-grow`:

- `0` – zůstane na výchozí šířce.
- `0.5` – vezme si polovinu volného prostoru.
- `1` – roztáhne se do celé šířky volného prostoru.
- `2` – opět se jen roztáhne do celé šířky volného prostoru.

U poslední možnosti, nastavení na `2`, se pozastavme. Je zde vidět, že `flex-grow` hospodaří vždy jen s prostorem uvnitř kontejneru flexboxu, takže neumožní položce zabrat dvojnásobek zbytku a vyrůst tedy mimo kontejner.

Hodnota `2` by měla efekt, pokud bychom v demu změnili nastavení růstu ostatních položek, například na `flex-grow:1`. V takovém případě by se stala poněkud obézní a zabrala by ze zbylého prostoru dvojnásobek oproti jejím dvěma sestrám.

CodePen: vrdl.in/zkcjw

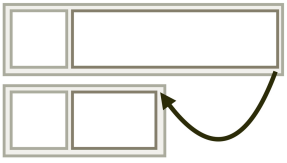
Podpora v prohlížečích

Podpora CSS vlastnosti `flex-grow` je plná. Jen pro zajímavost zmiňuji, že v Internet Exploreru 10 byla tato vlastnost pojmenována ještě jako `-ms-flex-positive`. Více o podpoře v prohlížečích najdete na webu CanIUse.

Flex-shrink: faktor smršťování položky

Jakým podílem se může položka flexboxu zmenšovat, pokud v rodičovském kontejneru místo ubylo? Od toho máme vlastnost `flex-shrink`.

Také v jejím případě platí, že je obecně lepší místo ní používat zkratku `flex`. Problematikou smršťování položek flexboxu se tady ale zabývat budeme, to si pište, že ano.



→ vrdl.in/cssfs

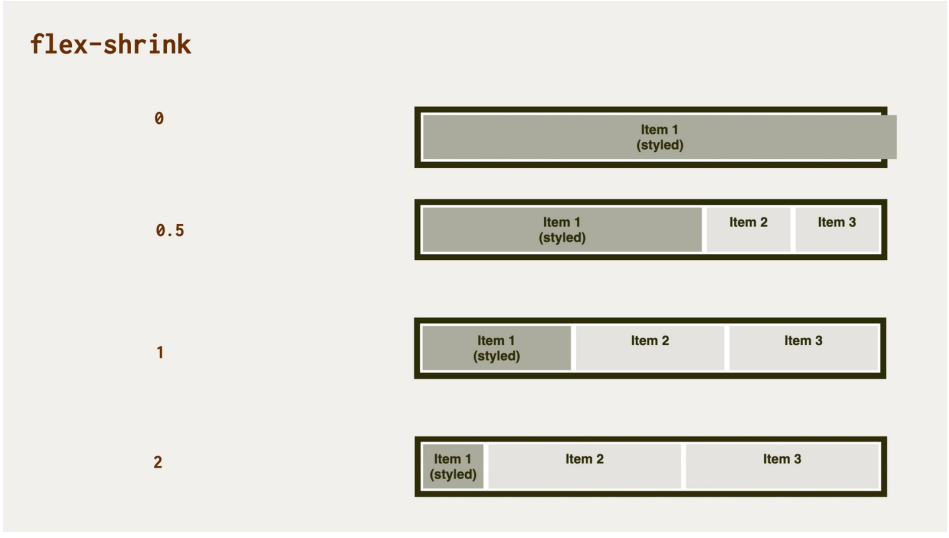
Možné hodnoty:

- `1` (výchozí)
  Položka si z vlastní šířky ubere část odpovídající celé velikosti smrštění.
- Kladná čísla (např. `0.5`, `1`, `2`...)
  Položka si ze své šířky vezme podíl z ubrané části kontejneru odpovídající tomuto číslu.
- `0`
  Položka je tvrdohlavá a smršťovat se nebude. A nebude!

Možná to celé zní trošku zašmodrchaně a bude lepší ukázat si to na obrázku a interaktivním demu.

Demo

Pevně doufám, že když se podíváte na obrázek, problematiku smršťování a vlastnosti `flex-shrink` pochopíte snáze.



Různé hodnoty vlastnosti flex-shrink na první položce způsobují různé smršťování.

Na rodiči (`.container` s nastavením `display:flex`) zde leží tíha udržet na uzdě tři potomky (`.box`):

```
<div class="container">
  <p class="box box--one"><strong>Item 1<br>(styled)</strong></p>
  <p class="box box--two"><strong>Item 2</strong></p>
  <p class="box box--three"><strong>Item 3</strong></p>
</div>
```

Demo jsme obohatili ještě o nastavení každého potomka tak, aby se pokoušel urvat co nejvíce prostoru z rodiče:

```
.box {
  width: 100%;
}
```

Ta paralela se skutečným rodičovstvím je zde až mrazivá. Mámy a tátové mezi čtenáři vědí, že takhle rozpínavé děti, bojující o každičký kousíček naší pozornosti, nechceme, ale nakonec jsou takové nějakou chvíli všechny. Raději se vraťme k CSS…

Podívejme se na různé možnosti nastavení `flex-shrink`, které dostává první dítě (nebo raději *potomek*) jménem `.box`:

- `0` – nesmršťuje se. Nebude se za žádnou cenu omezovat. Jednou jsme ale řekli `width:100%`, tak co bychom také chtěli?… Podřídí se tomuto pravidlu a oba ostatní sourozence vytlačí ven. Vzhledem k tomu, že výchozí hodnota `flex-shrink` je `1`, sourozenci si poctivě ze své šířky ubrali.
- `0.5` – ubere si polovinu oproti sourozencům, takže při zmenšování rodičovského prvku zabírá stále více místa.
- `1` – v tomto případě se všichni potomci spravedlivě smršťují stejným podílem. (Z pohledu rodiče ideální stav.)
- `2` – první položka si ubírá dvojnásobek oproti zbylým dvěma.

Je to trochu jasnější? Snad ano. Živou ukázku hledejte v následujícím odkazu.

CodePen: vrdl.in/uqz1m

Podpora v prohlížečích

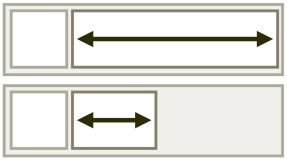
Podpora vlastnosti `flex-shrink` je plná. Internet Explorer 10 používá jako výchozí hodnotu chybně 0, ale tuto informaci už v 21. století potřebovat nebudete. Vše o podpoře najdete na webu CanIUse.com.

Flex-basis: velikost položky

Vlastnost `flex-basis` určuje výchozí velikost prvku v rámci rozvržení postaveného na flexboxu.

Je to velice podobné jako vlastnosti `width` nebo `height`, ale s tím, že velikost definovaná ve `flex-basis` má vždy přednost.

Doslovně se tato vlastnost definuje jako šířka nebo výška před rozdělením volného prostoru v kontejneru flexboxu.



→ vrdl.in/cssfb

Většinou určuje `flex-basis` výchozí šířku položky, tedy jako `width`. Alternativně výšku, pokud je směr rozvržení flexboxu svislý, například pomocí nastavení `flex-direction:column`.

Možné hodnoty:

- `auto` (výchozí)
  Přebírá rozměr podle `width` a `height`. Pokud ten je také `auto`, pak platí, že rozměr určuje obsah, tedy se použije dále uvedená hodnota `content`. Distribuce volného místa pomocí `flex-grow` a `flex-shrink` se pak bude týkat jen místa, které položky obsadily nad rámec svého obsahu – více rozebírám „relativní model“ pružnosti níže.
- `content`
  Nastaví velikost na základě obsahu položky. Toto klíčové slovo zatím podporuje jen Firefox a Chrome. V mých CodePenech níže jsou vidět rozdíly mezi `content` a `auto`. Podobného efektu jako s `flex-basis:content` dosáhnete nastavením `flex-basis:auto` a zároveň vlastností `width/height` také na hodnotu `auto`.
- `0`
  Nehledí se na rozměr obsahu. Distribuce volného místa pomocí `flex-grow` a `flex-basis` se bude týkat celé šířky položky – „absolutní model“ pružnosti.
- Nastavení jakéhokoliv CSS rozměru, např. `20%`, `100px`, `15em`… – toto je naštěstí nejčastější typ hodnoty, a tedy ne tak komplikované.

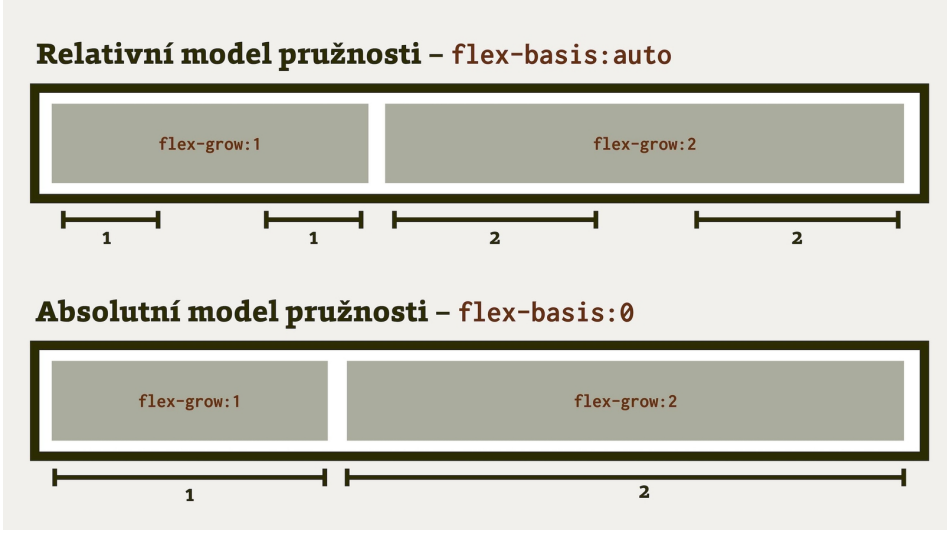
Firefox podporuje ještě další hodnoty – `max-content` a `min-content`. O obou píšu v kontextu funkce `minmax()`.

Mimochodem, dokumentační stránka MDN zmiňuje vtipnou historii nastavení `flex-basis:auto`. Původně znamenalo „vezmi rozměry z `width`/`height`“, pak bylo změněno na „vezmi rozměry z obsahu“, aby pak bylo znovu změněno na „vezmi rozměry z `width`/`height`“. Ale nevím, zda jste zrovna tohle chtěli vědět… mdn.io/flex-basis

Různé typy dělení volného prostoru

Zmínil jsem dva možné způsoby počítání zvětšování nebo zmenšování velikosti položky flexboxu. Je to důležité, takže to zde vysvětlím:

- *Relativní pružnost*
  Pokud uvedeme `flex-basis:auto`, velikost boxu se počítá z obsahu a pak se teprve přičtou nebo odečtou faktory růstu (`flex-grow`) nebo smršťování (`flex-shrink`).
- *Absolutní pružnost*
  Pokud uvedeme `flex-basis:0`, nezohlední se velikost textového nebo jiného obsahu. Růst nebo smršťování se vypočítává z celé šířky boxu.



Není pružnost jako pružnost. Není model jako model.

Na horní části obrázku se pružnost počítá relativně k obsahu, takže se vezme šířka obsahu a k němu se po stranách připočte volný prostor odpovídající podílu.

Na dolní části je vidět absolutní pružnost – obsah se vůbec nebere v úvahu a podíl na celku se počítá z celé šířky elementu.

Standardní je samozřejmě relativní model pružnosti, ale je potřeba vědět, co nám způsobí `flex-basis:0` nebo třeba nastavení vlastnosti `flex` na konkrétní číslo: `flex:1`.

Ukázky

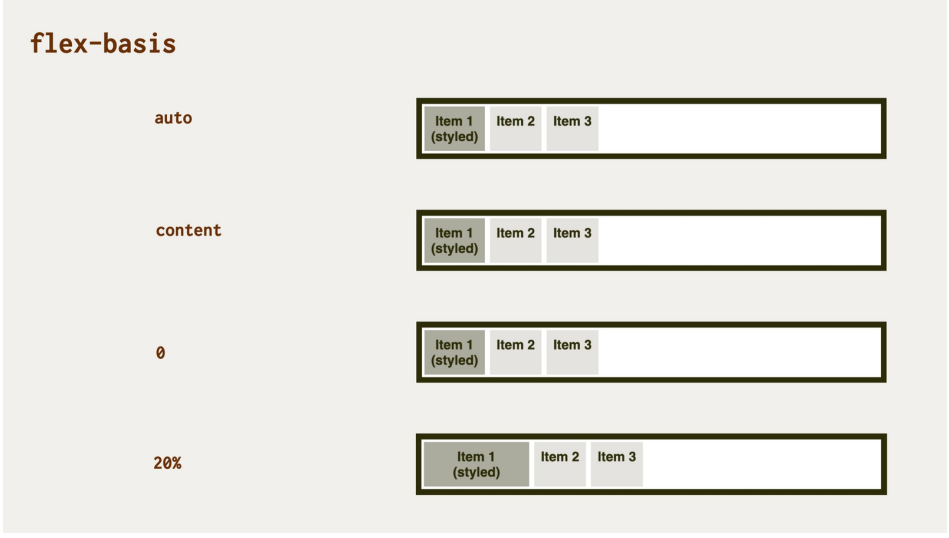
Máme zde opět jednoduché HTML:

```
<div class="container">
  <p class="box box--one"><strong>Item 1 (styled)</strong></p>
  <p class="box box--two"><strong>Item 2</strong></p>
  <p class="box box--three"><strong>Item 3</strong></p>
</div>
```

Kontejner je nastavený jako `display:flex` a pomocí přepínačů stylujeme první položku – `.box--one`.

Ukázky jsem připravil dvě – v jedné využívám vlastnost `flex-basis` samotnou a ve druhé zároveň s `width`.

Ukázka bez použití width



Různé možnosti hodnot flex-basis. Vysvětlujeme je v textu pod obrázkem.

Jak zde můžete vidět, mezi hodnotami `auto` a `content` žádné rozdíly nejsou. Položky `.box` totiž nemají nastavené rozměry pomocí vlastností `width` či `height`.

U nastavení `flex-basis:0` se použije nejmenší možný obsahový rozměr, což je v případě textu šířka nejdelšího slova.

Jakmile nastavíme rozměr (`flex-basis:20%`), chová se vlastnost stejně jako `width`.

CodePen: vrdl.in/uo0z3

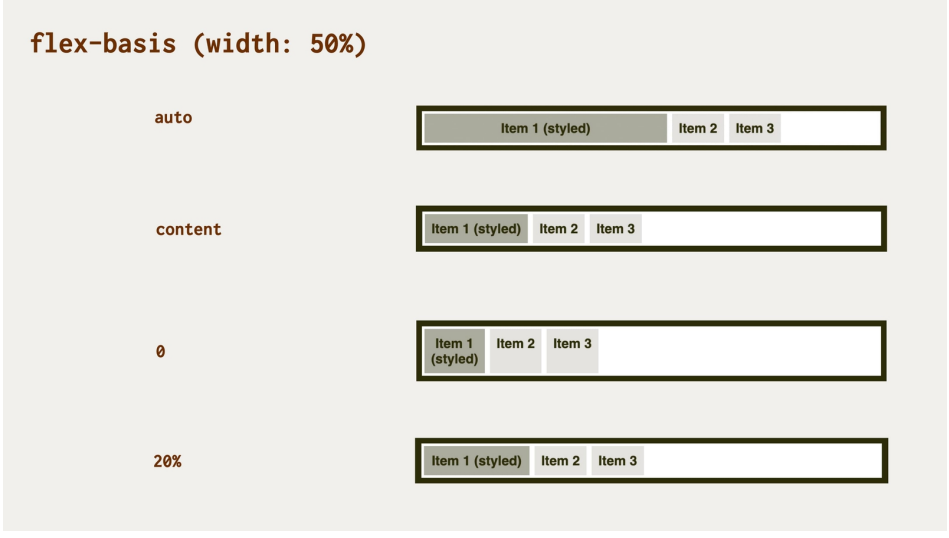
Ukázka s použitím width

V druhém CodePenu jsme to zkomplikovali přidáním tohoto stylu:

```
.box--one {
  width: 50%;
}
```

Nejdříve se možná zkuste podle výše uvedeného zamyslet, co přesně se přidáním vlastnosti `width` změní. Tady je totiž potřeba si uvědomit důležitou

věc, kterou už jsem zmiňoval – `flex-basis` je nadřazená vlastnost k `width` nebo `height`. Takže i když máme `width` nastavenu později, a měla by tudíž přebít `flex-basis`, nestane se tak.



Přidáním šířky pomocí vlastnosti width se mění situace.

Rozeberme si, proč a jak se to celé změnilo:

- `auto` – má převzít hodnotu z `width`, což se tady stalo.
- `content` – má vynutit šířku podle obsahu.
- `0` – nejmenší obsahová šířka, přebíjí `width`.
- `20%` – konkrétní rozměr, opět přebíjí `width`.

CodePen: vrdl.in/z4pq8

Vlastnost flex-basis a width/height

Z již uvedeného leccos vyplývá, ale myslím, že nebude od věci si přehledně připomenout, jak se liší nastavování rozměrů pomocí `flex-basis` od nastavování rozměrů s použitím `width` a `height`.

1) Vlastnost flex-basis má přednost

Platí, že při použití obou metod nastavení výšky nebo šířky dostane hodnota ve `flex-basis` přednost, bez ohledu na pořadí uvedení. Zkuste si to případně v ukázce.

CodePen: vrdl.in/hqy9c

2) Vlastnost flex-basis je obousměrná

Lze ji použít jak pro výšku, tak pro šířku. Vždy podle směru layoutu a ten určuje nejčastěji vlastnost `flex-direction`.

CodePen: vrdl.in/6zfg0

3) Vlastnost flex-basis se nezmenší pod minimální velikost obsahu

Na rozdíl od `width` a `height` nemůže být `flex-basis` menší než minimální šířka obsahu, což je například u textu šířka nejdelšího slova.

CodePen: vrdl.in/59gdw

4) Minimální a maximální velikost platí

Pokud byste chtěli omezit velikost prvku minimem nebo maximem, můžete využít vlastností `min-width/height` a `max-width/height`.

Toto platí právě proto, že:

- rozměry prvku ve flexboxu se berou z `flex-basis`,
- pokud to není definováno, pak se berou z `width/height`,
- pokud ani jedno není definováno, pak se berou z obsahu.

Kouzelná formulka tedy zní následovně:

```
flex-basis > width/height > obsah
```

`flex-basis` má přednost před `width/height` a ty mají přednost před obsahem.

Používejte vlastnost flex

Obecně navíc platí, že vlastnost `flex` je zase lepší než `flex-basis` – a to je lepší než `width/height`.

Vlastnost `flex-basis` totiž zvládne jak výšku, tak šířku, což se při změně směru rozvržení může hodit.

Vlastnost `flex` je ještě praktičtější, protože má chytře vymyšlené výchozí hodnoty a zároveň pomocí ní můžete definovat další důležité vlastnosti – smršťování pomocí `flex-shrink` nebo rozpínání pomocí `flex-grow`.

Podpora v prohlížečích

Základní podpora pro `flex-basis` je v prohlížečích výborná. Výjimku si nárokuje jen Internet Explorer, který při jiném nastavení než `flex-basis:auto` počítá velikost prvku podle způsobu počítání rozměru boxu („box model“) v režimu `content-box`, což je výchozí hodnota počítající jen se samotným obsahem a bez vlastností hodnot okrajů.

No a pak tady máme klíčové slovo `content`. Stejně jako klíčová slova `max-content` a `min-content` je zatím nepodporuje Safari, ale v únoru 2022 to vypadá, že je ohlášena podpora v další verzi.

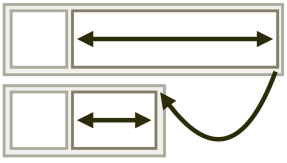
Více najdete na CanIUse.com.

Flex: nastavení pružnosti položky

Vlastnost `flex` nastaví výchozí velikost elementu a způsob, jakým se smí zvětšovat a zmenšovat v rámci flexboxového layoutu.

Jde o zkratku pro všechny vlastnosti definující pružnost flexboxové položky:

- `flex-grow` – faktor zvětšování.
- `flex-shrink` – faktor smršťování.
- `flex-basis` – základní velikost.



→ vrdl.in/cssf

Zapsáno obecně vypadá složení hodnot vlastnosti `flex` takto:

```
flex:
  <flex-grow> <flex-shrink> <flex-basis>
```

Výchozí hodnota je `initial` a můžete ji zapsat následujícím způsobem:

```
.box {
  flex: 0 1 auto;
}
```

- `flex-grow:0` – nebude se nijak roztahovat do volného místa.
- `flex-shrink:1` – smršťovat se bude stejně jako ostatní položky.
- `flex-basis:auto` – zabere prostor, který jí určí vlastní obsah.

Pokud chcete například nastavit, aby vaše položky zabíraly minimálně `150px`, v případě dostupnosti volného prostoru se rovnoměrně zvětšily a v případě zmenšení prostoru zase rovnoměrně smrštily, uděláte to takto:

```
.box {
  flex: 1 1 150px;
}
```

Používejte flex raději než konkrétní vlastnosti

Je dobré vědět, že autoři specifikace doporučují upřednostňovat zkratku `flex` před konkrétními vlastnostmi, které zastupuje.

Důvodem je mimo jiné to, že zkratka umí inteligentně nastavovat výchozí hodnoty.

Přednastavené hodnoty

Jednoslovné hodnoty se vám budou hodit asi nejčastěji. Podívejte se na první sloupec tabulky:

Hodnota	`flex-grow`	`flex-shrink`	`flex-basis`
`flex:initial`	`0`	`1`	`auto`
`flex:auto`	`1`	`1`	`auto`
`flex:none`	`0`	`0`	`auto`
`flex:1`	`1`	`1`	`0`

Teď si vysvětlíme dopad přednastavených hodnot zkratky `flex`:

- `flex:initial`
  Nastavení výchozí hodnoty, tedy `flex:0 1 auto`. Položky se s ubývajícím místem zmenšují, ale nezvětšují se nad velikost svého obsahu.
- `flex:auto`
  Odpovídá `flex:1 1 auto`. Dotčené položky se stanou plně pružnými s výchozím rozměrem podle svého obsahu. Asi nejčastější případ, který potřebujete.
- `flex:none`
  Odpovídá `flex:0 0 auto` a zcela ruší pružnost položky. Druhá nejčastější situace.
- `flex:<kladné-číslo>`
  U jednočíselného zápisu pozor! `flex:1` znamená `flex:1 1 0`, takže se vám změní výchozí velikost položky a model pružnosti, jak jsme zmiňovali u vlastnosti `flex-basis`.

Přednastavené hodnoty vlastnosti flex a jejich dopad na velikost položky. Zapište si je za uši, jsou hodně důležité.

Je také dobré vědět, že se flexboxové položky nikdy nezmenší pod minimální šířku obsahu. Ta je dána šířkou nejdelšího slova nebo vnitřního elementu fixní šířky – třeba obrázku. Lze to změnit nastavením `min-width` nebo `min-height` tohoto prvku na nějakou nízkou hodnotu.

Vyzkoušet si to opět můžete na našem interaktivním CodePenu.

CodePen: vrdl.in/wrntf

Podpora v prohlížečích

Internet Explorer ve verzích 10 a 11 tuto vlastnost implementoval jako `-ms-flexbox`, ale s pomocí Autoprefixeru (více v deváté kapitole) není nutné na toto myslet. Všechny informace o podpoře jsou na CanIUse.

Shrnutí čtvrté kapitoly

1. **Jaký je výchozí směr os flexboxu?**
   a) hlavní osa má vodorovný směr, příčná osa svislý
   b) hlavní osa má svislý směr, příčná osa vodorovný
   c) jde o chyták, flexbox na rozdíl od gridu nemá dvě osy
2. **Jak můžeme změnit směr os flexboxu?**
   a) vlastností `flex-direction`
   b) vlastností `flex-axis`
   c) vlastností `flex-direction` a `writing-mode`, nastavením jazyka
3. **Co zajistí zápis flex-flow:column?**
   a) zalomí (`wrap`) a vykreslí svisle (`flex-direction:column`)
   b) zákaz zalomení (`nowrap`) a vykreslí svisle (`flex-direction:column`)
   c) jde o zápis pro vytvoření sloupcové sazby textu
4. **Jaké vlastnosti odpovídá flex-basis?**
   a) `width`
   b) `height`
   c) většinou `width`, ale v opačném směru layoutu pak `height`
5. **Kterým zápisem vytvoříme „absolutní model pružnosti“?**
   a) `flex-basis:auto`
   b) `flex-basis:1`
   c) `flex:0`
6. **Co nastavuje zkratka flex?**
   a) šířku nebo výšku položky
   b) vlastnosti `flex-grow` a `flex-shrink`
   c) vlastnosti `flex-grow`, `flex-shrink` a `flex-basis`
7. **Čemu odpovídá zápis flex:auto?**
   a) `flex:1 1 auto`
   b) `flex:0 0 auto`
   c) `flex:0 1 auto`

Řešení: 1. a), 2. c), 3. b), 4. c), 5. c), 6. c), 7. a).

05

GRID

CSS grid je nejnovější, nejrobustnější a dle mého názoru nejslibnější specifikace pro tvorbu rozvržení v kaskádových stylech.

I když se v této kapitole věnujeme jen referenci, tedy popisu jednotlivých vlastností, funkcí a hodnot, jde o jednu z nejdelších částí knihy.

Pokud jste v gridu noví, doporučuji vám projít si všechny vlastnosti a ideálně také vyzkoušet CodePeny, kterými jsem opět nešetřil.

Zkušenější mohou zvolit buď podrobné studium (jdu zde opravdu do hloubky) nebo si vybrat pokročilejší vlastnosti, na které se zaměří. Jako takové si umím představit například zkratku `grid`, klíčová slova `auto-fill`, `auto-fit` nebo funkci `minmax()` a související klíčová slova. Nebo úplné novinky jako masonry a subgrid, které v době psaní nemají plnou podporu v prohlížečích, ale vypadají nadějně.

Úvod a rozcestník

CSS grid je sada vlastností pro tvorbu layoutu vsazeného do pravidelné mřížky.

Síla gridu je kromě jiného v tom, že jej můžeme definovat jako mřížku v obou směrech – v řádcích i sloupcích. Na rozdíl od flexboxu tedy může být vhodnější pro celostránkové a komplexnější layouty.

Grid zvládají všechny dnešní prohlížeče. Internet Explorer 11 ale podporuje jen starší verzi specifikace CSS gridu. Nevadí, za prvé již téměř vymřel a za druhé – i v něm je možné mřížku s kompromisy použít.

Příručka k vlastnostem

Následuje detailní příručka k jednotlivým vlastnostem CSS gridu.

Nastavení rodiče mřížky

Hodnota vlastnosti `display:grid` nastaví „grid formatting context“.

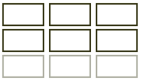
Můžete také nastavit „inline“ hodnotu: `inline-grid`. To je věc podobná `inline-block`, tedy uvnitř je možné dělat rozvržení, zde pomocí gridu, zvenčí jde o součást řádku textu.

Definice mřížky

Následujícími vlastnostmi je možné definovat kostru rozvržení.

---

grid-template-columns/rows

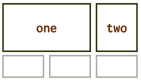


Definice explicitní mřížky.

```
grid-template-columns:1fr 1fr 1fr;grid-template-rows:1fr 1fr;
```

---

grid-template-areas



Pojmenování oblastí.

```
grid-template-areas:"one two"
```

---

grid-template

Zkratka pro definici explicitní mřížky.

```
grid-template:"one one two" 1fr ⏎ "one one two" 1fr / 1fr 1fr 1fr;
```

---

grid-auto-columns/rows

Definice implicitní mřížky.

```
grid-auto-rows:1fr;
```

---

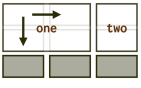
grid-auto-flow

Způsob automatického umísťování do mřížky.

```
grid-auto-flow:column;
```

---

grid



Zkratka pro definici všeho.

```
grid:auto-flow 1fr / 100px;
```

Vysvětlivka: Symbol ⏎ značí, že v běžném kódu je potřeba odsadit pomocí klávesy Enter.

Umístění do mřížky

grid-column/row

Umístění do explicitní mřížky.

```
grid-column:2 / 3;
```

grid-area

Umístění do pojmenované oblasti.

```
grid-area:first;
```

Funkce a klíčová slova

Do následující škatulky spadly nejrůznější výpočetní funkce, speciální klíčová slova a jednotky, které vznikly pro potřeby gridu:

jednotka fr

Jednotka pro podíl na celku.

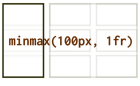
```
grid-template-columns:1fr 2fr;
```

funkce repeat(), auto-fill a auto-fit

Funkce zkrácení opakujících se zápisů.

```
grid-template-columns:repeat(8,1fr);
```

funkce minmax()



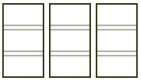
Omezení minima a maxima v rozměru.

```
grid-template-columns:minmax(100px,1fr);
```

Vlastnosti, které zatím nemají plnou podporu

Následující dvě nové vlastnosti sice zatím není možné snadno použít v praxi, ale vypadá to nadějně a ke všemu je považuji za velmi užitečné:

subgrid



Podmřížka uvnitř mřížky.
`grid-template-cols:subgrid;`

masonry

Masonry (zděný) layout.
`grid-template-rows:masonry;`

Základy gridu a první příklad

Řekněme, že naše HTML vypadá takto:

```
<div class="container">
  <div class="side-1">Side 1</div>
  <div class="content">Content…</div>
  <div class="side-2">Side 2</div>
</div>
```

Pojďme to nakódovat. Nejprve příprava na layout do mřížky:

```
.container {
  display: grid;
}
```

Toto – na rozdíl od `display:flex` – samo od sebe nic nedělá. Zatím jsme definovali mřížku, takže máme smůlu. Vzhůru do ní:

```
@media screen and (min-width: 37.5em) {
  .container {
      grid-template-columns: 1fr 3fr 1fr;
  }
}
```

Vytvoříme tím layout rozdělený na pět sloupečků mřížky, přičemž první a poslední části pro postranní obsah zaberou jednu pětinu. Prostřední část (`content`) má tři pětiny.

Jednoduchý layout s použitím CSS gridu. Ani to nebolelo.

Zbývá doladit nějaké ty mezery mezi buňkami mřížky. V gridu máme pro ty potřeby novou vlastnost gap:

```
.container {
  gap: 0.5em;
}
```

CodePen: vrdl.in/gzwr7

Po představení všech vlastností a rychlém příkladu bych se rád pozastavil u teoretických konceptů CSS gridu.

Co je to vlastně mřížka?

- Nic nového. Do pravidelné mřížky se už staletí sází velká část knížek, novin a tiskovin obecně.
- Webařským veteránům lze CSS grid přiblížit jako tabulkový layout, jen daleko lépe udělaný a zbavený nevýhod.
- Těm, kteří znají Bootstrap a podobné frameworky, lze grid podat jako zdokonalený Bootstrap grid vestavěný v prohlížečích.

Všechno by to ale byla pravda jen částečně. Grid je daleko silnější než uvedené příklady.

Jak se liší grid od flexboxu?

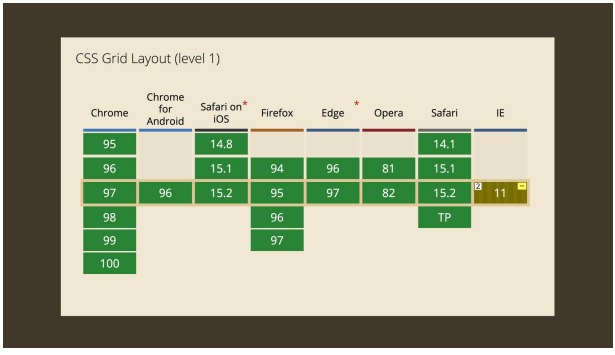
Předně chci zdůraznit, že grid nenahrazuje flexbox. Potřebujete obojí. Naučte se obojí. Ale nějaké rozdíly zde jsou:

- Grid je silnější pro dvourozměrné layouty – po svislé i vodorovné ose. Flexbox se více hodí na rozvržení jednosměrná.
- Grid budete asi častěji používat pro layout celé stránky, flexbox pro layout menších komponent. Ale není to pravidlo.
- Grid je zaměřený více na layout „grid in", kdy se obsah musí vždy přizpůsobit mřížce. Flexbox je super na situace „content out", kdy se layout musí přizpůsobit obsahu.
- Grid může být také silnější v responzivním designu. Budete potřebovat méně Media Queries, protože obsahuje funkcionalitu jako automatický layout, funkce jako `minmax()`, `repeat()` nebo klíčová slova jako `auto-fill`.
- Grid považuji celkově za robustnější. Jak možná sami vidíte, ve specifikaci gridu je daleko více vlastností než ve flexboxu. Je to i vidět u vlastností zarovnávání boxů (CSS Box Alignment) – těch, které jsou navázané na mřížku, je více než těch, které jsou navázané na flexbox.

Z mého pohledu má grid daleko širší možnosti stylování než při využití flexboxu. Svedeme v něm i dříve těžce kódovatelné layouty.

Podpora

Podpora gridu v moderních prohlížečích je výborná, takřka bezchybná.



Podpora gridu v prohlížečích, které mají v ČR nad 0,5 % podílu trhu. Prohlížeče jsou seřazené podle používanosti. Zdroj: CanIUse.com.

O podpoře moderních systémů layoutu v prohlížečích píšu ve zvláštním textu osmé kapitoly, ale u gridu není v kontextu moderních prohlížečů téměř o čem psát.

Horší je to ve stařičkém prohlížeči od Microsoftu.

Grid v Internet Exploreru 11

Explorer a starší Edge podporují dnes už neplatnou specifikaci gridu. Má jinou syntaxi a jde o podmnožinu dnešních možností gridu. Navíc nepodporuje automatické umísťování prvků do mřížky. A neumí vlastnost `grid-gap`... No, a takhle bychom mohli pokračovat.

Naštěstí je tady Autoprefixer a dnes už je možné svěřit řešení gridu pro MSIE jemu.

Existence „explorerového“ gridu nám ale v mnoha případech umožní mřížku vcelku bez problémů používat i bez Autoprefixeru. Jen je potřeba se kromě základních vlastností dnešního gridu naučit i ten starší.

Vychytávky gridu

CSS grid má mnoho skvělých vlastností. Ne všechny jsou unikátní, některé jste totiž mohli vidět už u flexboxu, ale mřížka jejich seznam dále rozšiřuje.

Podívejme se na ně prozatím velmi stručně, časem se k nim ještě vrátíme.

1) Blokový a inline grid

Kontejner mřížky můžeme definovat kromě `display:grid` také jako `display:inline-grid` a použít ho vedle dalších řádkových (`inline` nebo `inline-block`) elementů. Více je v textu o vlastnosti `display`.

2) Rozměry, jaké si umanete: fixní i flexibilní

Šířky a výšky položek mřížky je možné definovat všemi způsoby, které už asi znáte – jednotkami `px`, `em`, `%` a mnoha dalšími.

Přidává se tady ale ještě nová jednotka `fr` (*fraction unit*), která umožní snadné dělení volného místa.

Kombinovat můžeme opravdu silně. První sloupec definujeme ve fixních pixelech, druhý v jednotce em odvozené z velikosti písma. Zbytek plochy rozdělíme podílovou jednotkou fr.
Zdroj: vrdl.in/jhfwd

Kdyby to nestačilo, máme v gridu k dispozici ještě například funkci `minmax()`, díky níž můžeme rozměry nastavit v rozmezí od minimální po maximální hodnotu.

A dále také funkce `min()`, `max()` a `clamp()`, které svou působností překračují hranice CSS layoutů.

3) Rozměry podle obsahu

Někdy je lepší nenastavovat šířku a výšku autorsky, ale nechat obsah položky, aby do rozměrů mluvil sám. K tomu jsou zde klíčová slova použitelná ve funkci `minmax()`:

- `min-content` – nezmenšuj se pod minimální rozměr obsahu.
- `max-content` – nezvětšuj se nad maximální rozměr obsahu.

Zajímavá je i funkce `fit-content()`, kdy vnutíme šířku podle obsahu i těm prvkům, které by se v daném kontextu chovaly jako blokové.

4) Zarovnávání všeho se vším

Už od dob flexboxu asi víte, že možnosti zarovnávání položek layoutu jsou vynikající. Vlastnosti jako `justify-content`, `align-items` nebo `align-self` v té době přinesly velké zjednodušení práce.

Ještě více možností máme v případě gridu – můžeme zde používat další vlastnosti jako `place-items`. Více je v samostatné části o CSS Box Alignment.

5) Nezávislost vzhledu na HTML zdroji

Frontendové kodérky a kodéři museli vzhledu často obětovat strukturu HTML, což přinášelo komplikace v oblasti přístupnosti stránky například čtečkám pro nevidomé nebo robotům vyhledávačů.

Když vynecháme nepěkné triky s pozicováním, slibně sem vstoupil flexbox. Co je to ale oproti gridu!

- Vlastnost `order`, která dovoluje změnit pořadí prvků v prohlížeči, už znáte z flexboxu. Ano, tady ji můžete použít taky.
- Pomocí vlastnosti `grid-area` budete schopni umístit jakéhokoliv potomka na jakékoliv místo mřížky. A zahodit přitom svěrací kazajku struktury HTML.
- Hodnota `dense` vlastnosti `grid-auto-flow` částečně nechává vykreslení layoutu na prohlížeči, což oceníte u jednoduchých rozvržení typu „masonry“.

Zde je ale samozřejmě potřeba zvážit možné negativní vlivy na přístupnost (viz osmá kapitola).

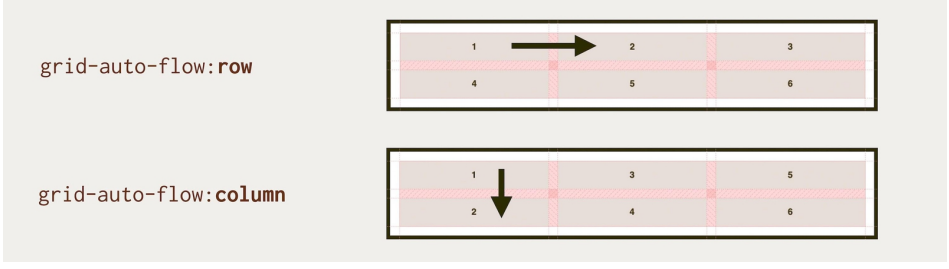
6) Automatické umístění položek do mřížky

Díky vlastnosti známé jako „autoplacement“ není nutné položkám gridu definovat, do kterých buněk mřížky je chceme umístit.

Pokud to totiž není definováno jinak, každý uzel DOMu, který je přímým potomkem kontejneru mřížky, se umístí do jedné buňky.

V našich podmínkách čteme zleva doprava a pak dolů. Položky se samozřejmě budou umísťovat ve stejném směru.

Pokud bychom snad chtěli změnit směr budování mřížky na shora dolů a pak doprava, pomůžeme si vlastností `grid-auto-flow` s hodnotou `column`.



Změnit směr rozvržení jde lusknutím prstu nebo lépe řečeno lusknutím vlastností grid-auto-flow.

7) Oblasti mřížky

Pojmenované oblasti, které definujeme s pomocí `grid-template-areas`, velmi usnadňují stavbu layoutů a jejich přeskládávání v různých rozlišeních:

```
@media screen and (max-width: 37.49999em) {
  .container {
    grid-template-areas:
      "first first"
      "second second";
  }
}

@media screen and (min-width: 37.5em) {
  .container {
    grid-template-areas:
      "first second"
      "first second";
  }
}
```

8) Umísťování prvků přes sebe a vrstvení

Jakmile si s vlastností `grid-area` zvyknete umísťovat prvky do konkrétních oblastí, objevíte zcela nový svět vrstvení.

Oblasti mřížky můžete snadno umístit do stejných míst a pomocí vám jistě dobře známé vlastnosti `z-index` pak vytvářet překryvy.

Samozřejmě je možné používat i vlastnost `position`, která snadno vytrhne položky mřížky z jejich původních pozic.

9) Zanořování mřížek

To, co je profíkům jasné jako facka, nemusí být patrné všem. Stejně jako blokové prvky nebo flexboxové kontejnery můžete do sebe vnořovat i kontejnery gridu.

Otevírá to další možnosti a prostor pro tvorbu precizně definovaných layoutů.

Do budoucna zde mohou ještě zaúřadovat a nové možnosti přidat rozpracované vlastnosti jako podmřížka (subgrid) nebo nová hodnota vlastnosti `display:contents`, jež umožní nevykreslit rodiče a celý prostor věnuje potomkům.

10) Nové typy layoutů

Sečteno, podtrženo: CSS grid může být nástrojem, který k webu přitáhne novou generaci webových designérů a s nimi nové, neotřelé layouty.

The Experimental Layout Lab Of Jen Simmons: Přehlídka layoutových experimentů a také neotřelých rozvržení, jejíž homepage je udělaná v gridu. Jak jinak.
Zdroj: labs.jensimmons.com.

Grid by vás měl zajímat, i když zrovna po experimentech a nových typech rozvržení netoužíte. Je vysoce pravděpodobné, že zjednoduší vaši práci a váš kód se díky němu stane elegantnějším a lépe čitelným.

Máte se na co těšit.

Linka, buňka, oblast a další pojmy

V gridu se to hemží pojmy, které si musíme vyjasnit. Jsou totiž důležité pro pochopení tohoto typu rozvržení a náchylné na záměnu, což nám může způsobit nejednu malou katastrofu.

Vysvětlíme si to na příkladu. Řekněme, že máme toto HTML:

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item item--five">5</div>
</div>
```

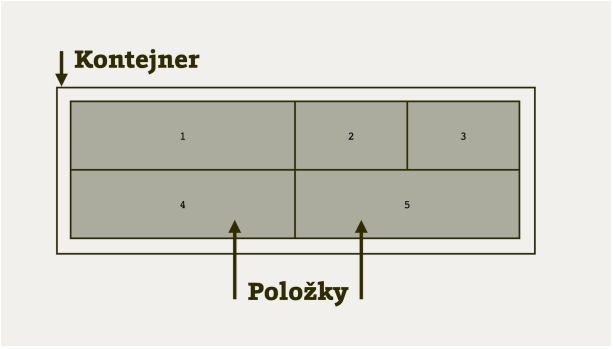
Prozatím nechme stranou specifické nastavení páté položky zde reprezentované třídou `.item--five`, k tomu se propracujeme.

Mřížku definujeme jako třísloupcovou, ale řádky necháme na automatice gridu:

```
.container {
  display: grid;
  grid-template-columns: 2fr 1fr 1fr;
}
```

S pomocí vlastnosti `grid-template-columns` vytvoříme kýžené tři sloupce. První z nich bude mít dvojnásobnou šířku oproti druhému a třetímu. Definujeme to zlomkovou jednotkou `fr`.

V prohlížeči bude naše mřížka vypadat následovně:



Byl jednou jeden kontejner a v něm žilo pět položek...

Na obrázku vidíme jeden kontejner a pět položek mřížky, čímž se dostáváme k našim pojmům. Budu je zde uvádět česky s anglickým ekvivalentem v závorce, v dalších textech už většinou jen česky.

Kontejner mřížky (grid container)

Kontejner vytvoříme z jakéhokoliv prvku v HTML prostým deklarováním `display:grid` nebo `display:inline-grid`.

Vznikne tím formátovací kontext pro mřížku (grid formatting context) a začnou platit trochu jiná pravidla než například pro blokový formátovací kontext (block formatting context), který by vznikl uvedením `display:block`.

V našem příkladu jsme tedy kontejner mřížky udělali z prvku `.container`.

Jednou z vlastností formátovacího kontextu mřížky je, že každý z přímých potomků ve stromě DOM bude prohlížečem považován za položku mřížky.

Položka mřížky (grid item)

Položkou mřížky v ukázce jsou všechny prvky `.item`.

Položka nemá žádné specifické vlastnosti – kromě toho, že je nesvobodná. Prohlížeč se ji totiž snaží vložit do definované mřížky.

Položku ale můžeme osvobodit tím, že i z ní uděláme kontejner (`display:grid`) a ona pak bude omezovat na svobodě své potomky. Pochopili jste mě správně – zanořování více gridů do sebe je možné.

Anonymní položky

Musíme si tady uvědomit, že položkou je každý uzel DOMu, který je přímým potomkem kontejneru mřížky.

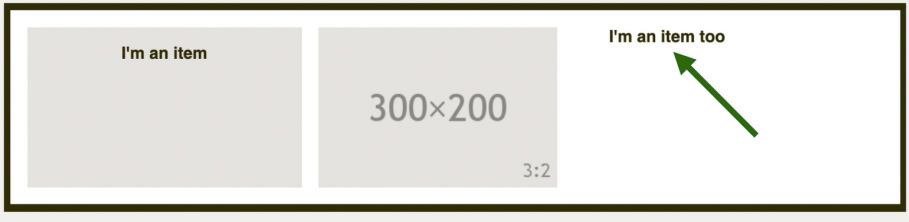
Do této kategorie patří i třeba obrázky nebo takzvané *anonymní uzly*, což jsou textové bloky, potomci kontejneru, neuzavřené v elementu HTML.

```
<div class="container">
  <div class="item">I'm an item</div>
  <img src="img.jpg" width="300" height="200" alt="…">
  I'm an item too
</div>
```

V této ukázce máme tedy rovnou tři položky mřížky a nějak si s nimi budeme muset poradit při jejím definování. Já jsem to tu pojal jednoduše:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

Rozdělil jsem plochu, kterou vymezuje kontejner, na tři rovnoměrné části. Na skutečné šířky položek bude mít vliv jejich obsah, ale o tom se budeme bavit později.



Obrázek i anonymní textový uzel jsou plnohodnotnými obyvateli definované mřížky.

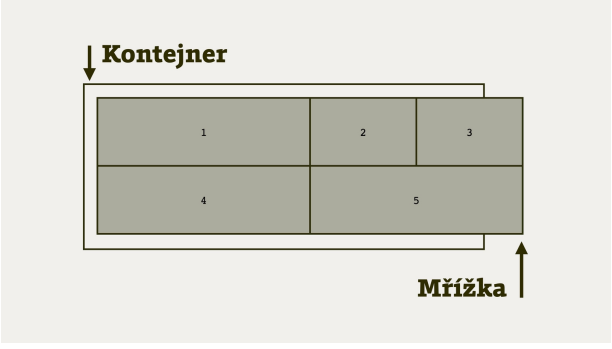
Tuto specialitu zmiňuji hlavně proto, abychom se dokázali vyhnout problémům, které to přináší. Obecně ale velmi doporučuji definovat každou položku gridu do samostatného HTML prvku. Už jen proto, aby se vám na něj lépe cílilo pomocí CSS selektorů.

CodePen: vrdl.in/z90q8

Kontejner mřížky je tedy rodičovský prvek, který v HTML definuje, že tady se bude odehrávat formátování CSS gridem. Položka mřížky je každý jeho přímý potomek, včetně těch anonymních.

Než dojdeme k dalším pojmům, spácháme na naší původní ukázce nepěknou a ošklivou věc. Zmenšíme plochu stránky tak, aby šířka položek přesáhla šířku kontejneru.

Kontejner je teď menší než samotná mřížka. Po vytečení „položek z kontejneru“ se dostáváme k dalším pojmům.



Tady je vidět rozdíl mezi kontejnerem mřížky a mřížkou samotnou.

Mimochodem, s původně uvedeným CSS a HTML by to takhle udělat nešlo. Musel jsem styly trochu změnit tak, aby si položky kontejneru zachovaly nějakou minimální šířku:

```
.container {
  grid-template-columns:
    minmax(500px, 2fr) minmax(250px, 1fr) minmax(250px, 1fr);
}
```

Funkci `minmax()` možná už znáte. Dělá přesně to, na co byste ji odhadovali – v našem případě zakazuje zmenšení první položky pod 500 pixelů a druhé a třetí pod 250 pixelů.

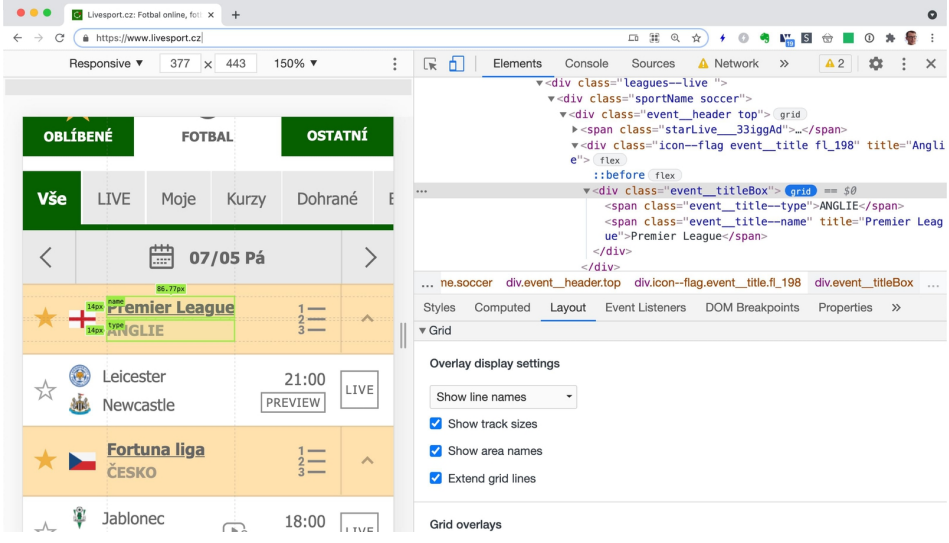
CodePen: vrdl.in/6mif9

Mřížka (grid)

Mřížka je ona neviditelná pravidelná síť, do které umísťujeme náš layout.

Může být stejně velká jako kontejner, ale nemusí. Kontejner je jen jakési „okno“ pro vykreslování mřížky.

Říkám, že je neviditelná, ale takto to být nemusí vždy a všude. Autoři prohlížečů jsou v této oblasti webovým vývojářům velmi nápomocní, a tak je možné samotnou mřížku pěkně vizualizovat v DevTools.

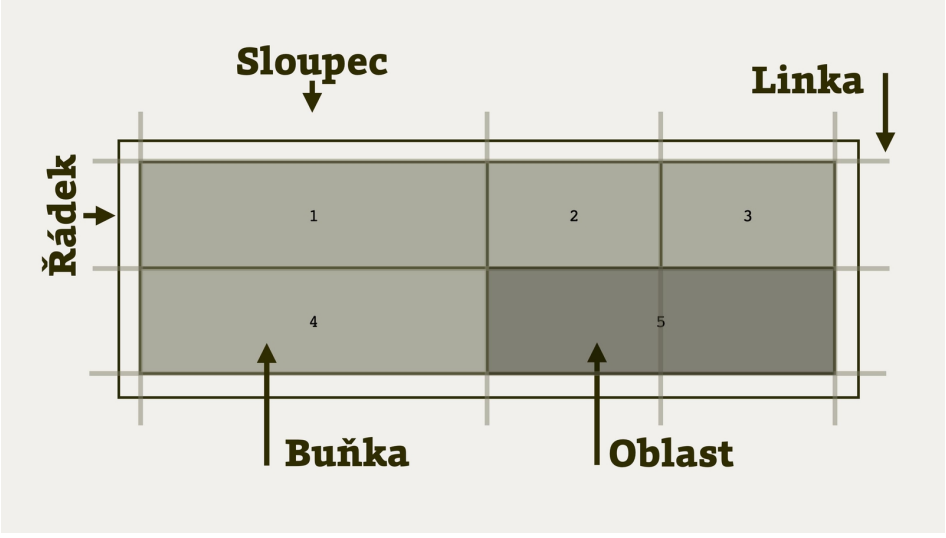


Grid v negližé, to je vizualizace mřížky v Chromu.

Vizualizace mřížky ve Firefoxu nebo i v Chromu pomůže pochopit, z čeho se váš grid skládá a jak to chápe prohlížeč. Velmi doporučuji tuto funkci při práci na mřížkách používat.

O podpoře layoutů v DevTools moderních prohlížečů je více v osmé kapitole.

Když už tedy víme, co je mřížka, a když už ji umíte vizualizovat, pojďme si říct, z jakých stavebních prvků se ta naše mřížka přesně skládá.



Tady máme celou partu: mřížka, linka, oblast, řádek i sloupec.

Linka mřížky (grid line)

Linka je základní prvek rozvržení mřížky. Jde o čáru, která mřížku dělí na řádky, sloupce a pak jednotlivé buňky.

Jak je patrné z obrázku výše, linky mají svá čísla. Pozor, začíná se vždy od `1`, nikoliv od nuly, na což může naletět nejeden programátorsky orientovaný vývojář.

Při definici mřížky, například vlastnostmi `grid-template-rows/columns`, je možné si linky pojmenovat:

```
.container {
  display: grid;
  grid-template-columns: [first] 2fr [second] 1fr [end];
}
```

Uvedený zápis zařídí následující:

- Pojmenuje vodorovné linky jako `first`, `second` a `end`.
- Mezi linkami definuje dva sloupce, první je `2fr` široký a druhý `1fr`.

Dostali jsme se díky tomu k dalšímu pojmu – sloupec mřížky. Než si jej rozebereme, musíme se seznámit s jiným pojmem – stopa mřížky.

Stopa mřížky (grid track)

Stopa je prostor, který je definovaný dvěma linkami. Vede přitom od začátku mřížky k jejímu konci. Je to vidět na obrázku výše.

Jde jen o obecný název pro řádek (vodorovnou stopu) nebo sloupec (svislou stopu).

Řádek, sloupec mřížky (grid row, grid column)

Vodorovné a svislé stopy mřížky jsou označovány jako řádky a sloupce, přesně jako to už znáte z tabulek.

Poznámka pro pokročilé: Pokud bychom chtěli být přesní, měli bychom uvést ještě pojem „osa mřížky“, což je stejně jako v geometrii přímka, jež určuje směry. V případě CSS gridu jsou ty směry naštěstí jen dva.

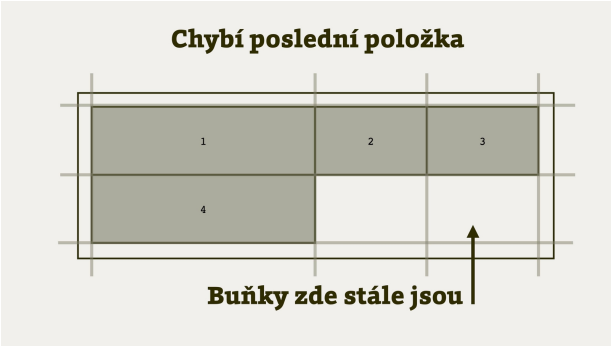
V pojmologii CSS gridu se nemluví o vodorovné a svislé ose, ale o blokové ose (block axis) a řádkové ose (inline axis). Proč proboha? Je to kvůli tomu, že CSS musí obsloužit nejen jazyky se zápisem vodorovným (jako je čeština nebo slovenština), ale také se zápisem svislým (jako může být například japonština). Pak se označení os prohodí, ale výhodou je, že ve směru psaní vždy zůstává bloková osa. Tím vás ale v textu nechci už opravdu více zatěžovat.

Buňka mřížky (grid cell)

Buňka je prostor vymezený čtyřmi linkami mřížky, který už není možné dále dělit dalšími linkami. Je to nejmenší prostorová jednotka mřížky, odpovídající buňce tabulky.

Pozor, buňka mřížky zároveň není položkou mřížky. Mřížka má vždy obdélníkový tvar, takže je možné definovat nižší počet položek, než má mřížka buněk.

Je to patrné v následujícím příkladu, kde jsme ubrali poslední položku. Mřížka však má stále tři sloupce a dva řádky:



Některé buňky neuvidíte, ale jsou tam pořád, potvory.

Je dobré si uvědomit, že právě proto nejde buňku mřížky nijak zacílit pomocí CSS. Je to jen jakási interní stavební jednotka mřížky.

CodePen: vrdl.in/cuz72

Abychom mohli buňku ovlivnit z CSS, musíme z ní udělat oblast.

Oblast mřížky (grid area)

Oblast je prostor vymezený čtyřmi linkami mřížky, který ale lze dále dělit dalšími linkami. Oblast se tedy skládá z jedné nebo více buněk mřížky.

V naší ukázce jsme pátou položku ručně umístili na místo páté a šesté buňky mřížky, tedy do prostoru, který v předchozím obrázku vidíte jako neobsazený.

```
.item--five {
  grid-column: 2 / 4;
  grid-row: 2 / 3;
}
```

Tento zápis přesně říká o umístění oblasti následující:

- Svisle ji umisť mezi druhou a čtvrtou linku mřížky `grid-column: 2/4`, čili na místo druhého a třetího sloupce.
- Vodorovně ji umisť mezi druhou a třetí linku mřížky `grid-row: 2/3`, čili do druhého řádku.

Ukázku uvidíte v CodePenu výše.

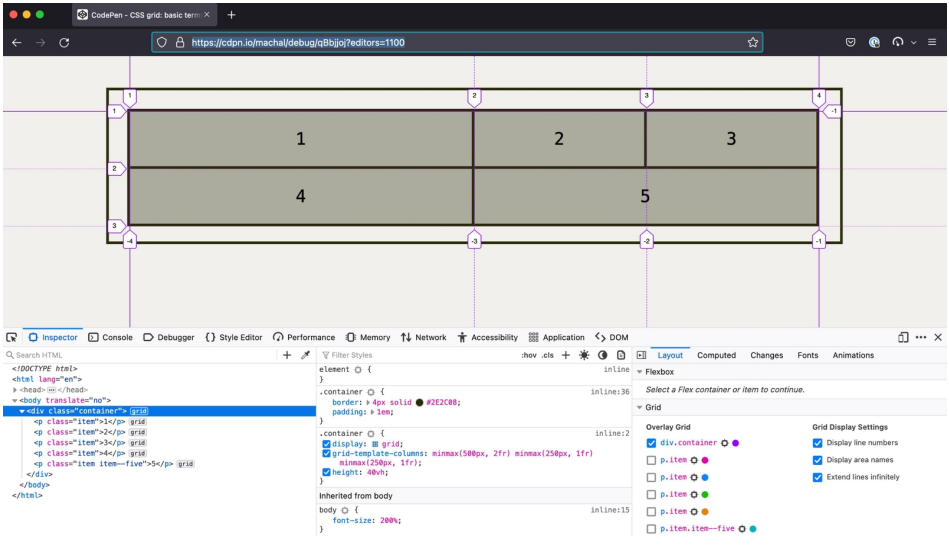
Mezery mezi buňkami (gap, gutter)

Bavme se nyní o mezerách. Jde o prostor mezi buňkami mřížky, pro jehož definici se používá vlastnost gap:

```
.container {
  display: grid;
  gap: 10px;
}
```

„Gutter“ by se měl nejlépe přeložit jako „žlab“, ale v kontextu webdesignu by to bylo slovo poměrně neobvyklé. Slovo gutter se v hovorové webařštině vcelku běžně používá, takže si s ním jako se synonymem pro žlab vystačíme a dále už o žlabech příliš mluvit nebudeme.

Pojďme si 10pixelový gutter nadefinovat v našem příkladu a podívat se, jak to bude vypadat v prohlížeči.



Linka mřížky se nám nafoukla do tvaru obdélníku.

Jak je vidět z obrázku, z linek mřížky se staly obdélníky. Nikoliv úsečky nulové tloušťky, ale poctivé objekty ve tvaru sloupců a řádků, které mají svůj začátek i konec.

Ano, chápete to dobře. Vlastnost gap určuje šířku linek. Pokud mezery nedefinujeme, linky tvoří jen neviditelné úsečky, tak jako jste to viděli v předchozích příkladech.

CodePen: vrdl.in/m87br

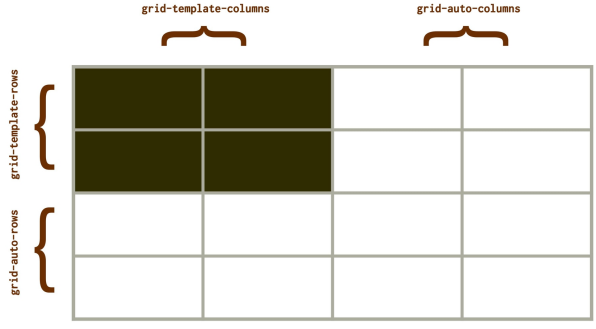
A to je vše, co jsem vám chtěl sdělit o stavebních prvcích v CSS gridu.

Mřížka explicitní a implicitní

Kromě výslovně uvedené explicitní mřížky existuje v CSS gridu ještě implicitní mřížka.

V naprosté většině případů definujeme explicitní mřížku, tedy *předem uvedenou*.

Může se nám ale stát, že položku umístíme na místo, kam explicitní mřížka nesahá. V takovém případě se nám mřížka automaticky rozšíří o implicitní část. Budete se divit, ale i tu můžeme předem definovat.



Mřížka s tmavou explicitní (doslovně definovanou) částí a bílou implicitní (automaticky definovanou) částí.

Podívejme se, pomocí kterých CSS vlastností můžeme obě mřížky definovat:

- Explicitní mřížku, výslovně definovanou, lze nastavit vlastnostmi obsahujícími slovo `template`:
  Jde konkrétně o `grid-template-*` – `grid-template-rows`, `grid-template-columns` a `grid-template-areas`.
- Implicitní mřížku, která obvykle není předem definovaná k vykreslení, můžete nastavit vlastnostmi obsahujícími slovo `auto`:
  Konkrétně uspějete s `grid-auto-*` – `grid-auto-rows`, `grid-auto-columns` a `grid-auto-flow`.

Příklad s přidáním položky na novou řádku

Jak už jsem psal, v naprosté většině případů definujeme mřížku jako explicitní. Vezměme třeba tento jednoduchoučký kód:

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

V CSS pak layout rozdělíme do tří sloupců, které se spravedlivě dělí o prostor:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

Co se ale stane, když do DOMu přidáme čtvrtou položku?

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item item--x">X</div>
</div>
```

Přidá se jako čtvrtá položka vedle třetí? Co myslíte?

Samozřejmě, že ne. Vždyť jsme prohlížeči výslovně (explicitně) řekli, že chceme třísloupcový layout.

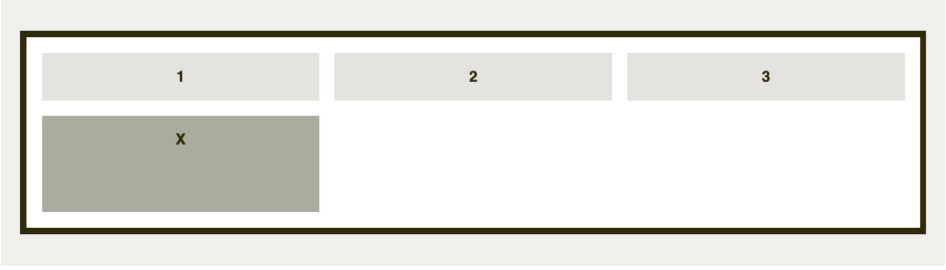
V takovém případě se layout zalomí a přidá se nový řádek. Ale počkat – vždyť jsme žádné řádky nedefinovali! Tohle je přesně místo v textu, kde na scénu přichází implicitní grid.

Už víte, že implicitní layout definují vlastnosti `grid-auto-rows`, `grid-auto-columns`, které mají výchozí hodnotu `auto`. To neznamená nic jiného než: „Přidávej směle nové řádky (nebo sloupce, pokud má layout opačný směr) a nic dalšího neřeš, nazdar."

Pokud bychom si ale přáli jiné parametry pro potenciální nové prvky mřížky, můžeme to prostě definovat:

```
.container {
  grid-auto-rows: 1fr 2fr;
}
```

Mřížce tím nepřikazujeme, aby byla rovnou dvouřádková. Říkáme zhruba toto: „Zatím nic nevykresluj, ale pokud by náhodou přibyla položka na druhou řádku, vykresli ji dvakrát tak vysokou, než jsou ty na prvním řádku.“ Tak tohle je ten slavný implicitní grid. Vypadá to teď jako na obrázku.



A heleme se, na druhém řádku je položka. Ta se formátuje podle automatického, implicitního gridu.

CodePen: vrdl.in/w9ua1

Příklad s přidáním položky na neznámou pozici

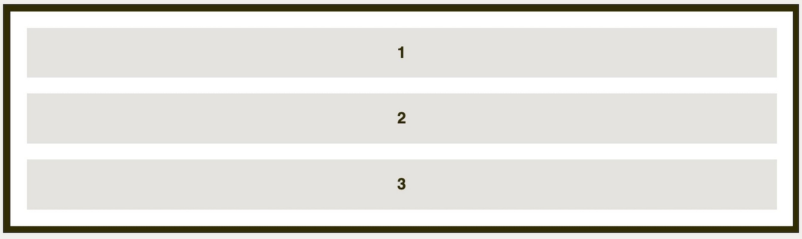
Tentokrát budeme drsní a žádnou explicitní mřížku definovat nebudeme. Jdeme na to:

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

Ve stylech sice zapneme formátovací kontext mřížky, nějakou tu mezeru pomocí vlastnosti gap, ale tím to hasne:

```
.container {
  display: grid;
  gap: 10px;
}
```

Prohlížeč si řekne: „Hmm… grid! Ovšem pozor – nemá deklaraci layoutu, takže to prostě vypíšu pod sebe.“ A jak řekne, tak udělá:



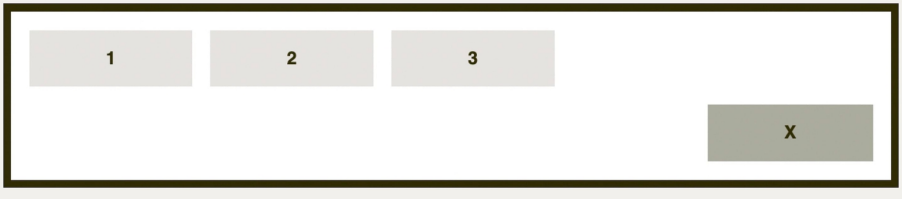
Tady žádný grid není, co jiného bychom měli čekat?

Jenže přesně na to vývojář čeká a prohlížeč jde zaskočit přidáním čtvrté položky do kontejneru s tímto předpisem:

```
.item--new {
  grid-column: 5;
  grid-row: 2;
}
```

Prohlížeč: „No toto! Máme tam přeci jen tři sloupce a jeden řádek. A ty mi tady něco umísťuješ do pátého sloupce a druhého řádku!“

Naštěstí si ale náš milý prohlížeč vzpomene, že kromě explicitní mřížky jej učili ještě implicitní. Tedy, že tam, kde chybí explicitní buňky, může vyskládat ty implicitní:



Hele, grid!

Výchozí výška i šířka implicitních buněk je na hodnotě `auto`, takže se rozměry určí podle obsahu položky. No a když tam žádný není… nemůžeme se pak divit, že čtvrtý sloupec je užší než první tři a nový pátý. Nemá totiž žádný obsah. Mohli bychom to opravit přípravou implicitní mřížky, například takto:

```
.container {
  grid-auto-columns: 1fr;
}
```

Všechny nově přidané sloupečky by pak měly stejnou šířku.

CodePen: vrdl.in/1nyf0

Hotovo! Snad se vám díky těmto dvěma příkladům povedlo proniknout do konceptu implicitní a explicitní mřížky, který je pro chápání CSS gridu velmi důležitý.

Formátovací kontext mřížky

Formátovací kontext (grid formatting context) je další teoretická znalost o mřížce, která se nám může velmi hodit.

Každá hodnota vlastnosti `display` totiž zapíná nový způsob zacházení s potomky prvku a může měnit vztahy prvku a jeho okolí.

Tak jako `display:block` má od časů CSS 2.1 svůj block formatting context, i CSS grid má jeden. Pojem *grid formatting context* vás tedy nejspíš nepřekvapí.

Co se v prohlížečích stane pěkného (pro někoho nepěkného), když napíšeme `display:grid` nebo `display:inline-grid`?

Svislé vnější okraje se neslučují. Chválabohu

Hodnoty vlastnosti `margin` shora a zdola se neslučují s hodnotami sousedních prvků. To, co znáte z blokových elementů, ve světě CSS gridu neplatí.

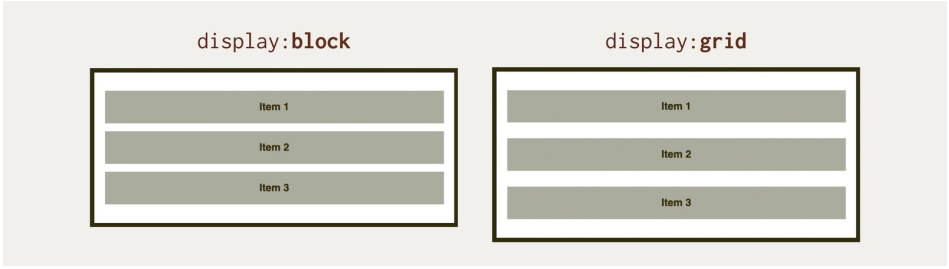
Je to logické. Mřížka slouží k rozvržení prvků na stránce. Při této činnosti by nám slučování vnějších okrajů působilo ještě větší trable než u blokových prvků.

Chcete ukázku? Vezměme, že máme dva bloky se třemi vnitřními položkami. Každá položka má tento předpis v CSS:

```
.item {
  margin: 1em 0;
}
```

Výsledek se různí podle formátovacího kontextu:

- `display:block` slučuje svislé okraje, takže mezi položkami je shora i zdola mezera jen `1em`.
- `display:grid` naproti tomu okraje neslučuje, takže mezi jednotlivými `.item` tvoří mezeru dvakrát `margin`, tedy `2em`.



Grid vyhání slučování.

Zájemci o více informací nechť prozkoumají následující živou ukázku.

CodePen: vrdl.in/349vz

Floaty nechte plavat

S plovoucími prvky v gridu neuspějete. Pro formátovací kontext mřížky platí, že:

- Vlastnosti `float` a `clear` nemají žádný efekt na položky mřížky. (Radši si ani nepředstavuji, co by vývojáři dělali, kdyby tomu tak nebylo.)
- Plovoucí prvky (např. `float:left`) nijak nevstupují do kontejneru mřížky. Sousední kontejner mohou posunout, zmenšit, ale nikdy nevstoupí dovnitř, aby rozbil položky.

Brrr! Raději pojďme od „floatů“ pryč. Začalo se mi dělat nevolno…

Vlastnost vertical-align nemá na položky mřížky žádný efekt

`vertical-align` slouží ke svislému zarovnání textového obsahu. Pro zarovnání v mřížce zde máme jiné úžasné vlastnosti, o kterých bude ještě řeč – CSS Box Alignment.

Pseudoelementy ::first-line a ::first-letter u kontejnerů mřížky nefungují

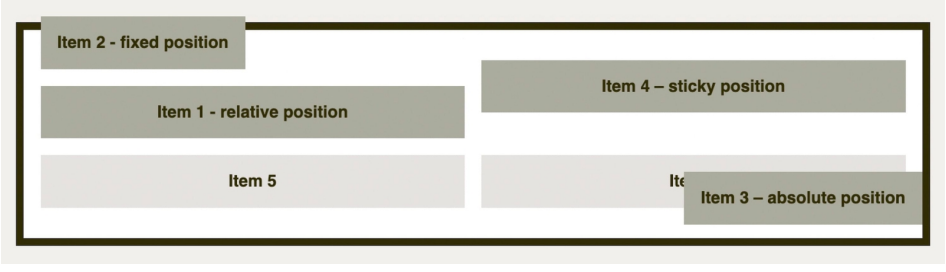
I tohle je naprosto logické. Pseudoelementy pro první řádek a první znak dávají smysl u typografických bloků. Jenže grid je šéf přes layout, nikoliv psaný text.

Pokud byste chtěli vidět tohle všechno v akci, neváhejte se podívat na můj CodePen, ve kterém jsem se grid pokusil rozbít všemi zde uvedenými způsoby. A byl jsem neúspěšný.

CodePen: vrdl.in/xdqjb

A co pozicování?

Úplně to sem nepatří, ale je možné, že se vám vynořila v hlavě otázka pozicování. Pozicovat můžete samotný grid kontejner, ale i jeho položky, pokud z kontejneru uděláte takzvaný containing block pomocí `position:absolute`. Vyzkoušel jsem to za vás, protože jsem byl zvědavý:



Vlastnost position funguje dobře, děkuji za optání.

CodePen: vrdl.in/ji5pg

To je myslím vše, co jste o formátovacím kontextu gridu potřebovali vědět.

Grid v Internet Exploreru

„Neumí to Explorer“ je, když to přeženeme, na prvních devíti místech z deseti důvodů, proč lidé ještě v roce 2022 CSS grid nepoužívají.

V realitě to ale zase tak moc velký problém není, protože u většiny projektů už nejspíš není MSIE nutné používat. Je-li toto i váš případ, tuto podkapitolu račte vynechat.

Pokud Explorer podporovat musíte, věřte, že nějakou (a ne úplně malou) podporu gridu má. To je jedna věc.

Druhá je, že podporu Exploreru můžete rozšířit pomocí chytrého hacku. V tomto textu se totiž budeme zabývat automatizovaným řešením pro zlepšení podpory gridu pomocí nástroje Autoprefixer.

Pojďme tedy dát kombinaci gridu s MSIE novou šanci.

Které vlastnosti gridu Internet Explorer podporuje?

Jisté rozdíly mezi MSIE a moderními prohlížeči zde jsou. Z těch důležitých například uveďme, že Internet Explorer nativně neumí následující:

- Automatické umísťování prvků do mřížky („autoplacement“).
- Pojmenovávání oblastí mřížky (vlastnosti jako `grid-template-areas`).
- Mezery mezi buňkami mřížky (vlastnost `gap`).

Velká část uvedeného pro vás ale přečtením tohoto dlouhého textu přestane platit.

Naopak se málo ví, že stařičký MSIE nativně podporuje následující:

- Implicitní (nepředddefinovanou) mřížku.
- Funkci `repeat()`, jen jinak. `repeat(12,1fr 20px)` zapisujte v Exploreru jako `(1fr 20px)[12]`.
- Funkci `minmax()`.
- Klíčová slova `min-content` a `max-content`.

To myslím není zlé. Jen připomínám, že rozdíly nevznikly v nějakém microsoftím „týmu pro vytáčení webařů“. Jejich příčinou je rychlá implementace gridu v rané fázi specifikace týmem v Microsoftu. Specifikace se bohužel časem změnila, grid ale v Exploreru zůstal v původní variantě, protože se tento prohlížeč ve své době aktualizoval jen velmi pomalu.

Třísloupcové demo aneb „Jak to kurnik šopa funguje?“

Podívejme se na velmi jednoduchý layout, u kterého si ukážeme, jak přesně Autoprefixer zařídí fungování gridu v MSIE.

CodePen: vrdl.in/4rbvs

Jde o rozvržení definované následujícím způsobem:

```
.container {
  display: grid;
  grid-template-columns: 1fr 3fr 1fr;
  grid-template-areas: "a b c";
  column-gap: 0.5em;
}
```

Pro zájemce vše vysvětlím polopaticky:

- `display:grid` je asi jasný. Definujeme prostě kontejner mřížky.
- `grid-template-columns:1fr 3fr 1fr` specifikuje samotnou mřížku. O jednotce `fr` píšu v jiném textu.
- `grid-template-areas` je šablona pojmenování oblastí pro následné využití v CSS.
- `column-gap:0.5em` je zápis pro mezeru mezi sloupci layoutu.

Teď to nejlepší. Tenhle kód nám Autoprefixer přeloží tak, aby v pohodě fungoval v Internet Exploreru 11. Včetně mezer (`-gap`) a pojmenovaných oblastí (`-areas`). Tedy vlastností, které tenhle pravěký prohlížeč neumí.

Než se k tomu dostaneme, musím zmínit jednu nevýhodu.

Proč layout zapisuji takhle složitě? Protože Autoprefixer

Vy znalejší jste si jistě všimli, že pro takto jednoduchý layout je zbytečné definovat šablonu pojmenování oblastí – `grid-template-areas`.

To je ale oběť na oltář podpory v Internet Exploreru, respektive vyžaduje to po nás Autoprefixer.

Dalším kódem ještě musíme umístit sloupečky do pojmenovaných oblastí:

```
.side-1 {
  grid-area: a;
}

.content {
  grid-area: b;
}

.side-2 {
  grid-area: c;
}
```

A šup! V další fázi se můžeme kochat kódem, který vypotí Autoprefixer.

Kód produkovaný Autoprefixerem

Nejprve se podíváme na kód generovaný tímto nástrojem pro rodiče layoutu:

```
.container {
  display: -ms-grid;
  -ms-grid-columns: 1fr 0.5em 3fr 0.5em 1fr;
}
```

Následuje samozřejmě výše uvedený kód pro moderní prohlížeče. Ten pro zjednodušení vynechávám. Opět ale oba řádky vysvětlím:

- `display:-ms-grid` – prefixovaný zapínač gridu v Exploreru.
- `-ms-grid-columns:1fr 0.5em 3fr 0.5em 1fr` – magie. Autoprefixer spojil definici mřížky s definicí mezer (`column-gap`), abychom ty (sakramentsky návykové) díry v layoutu mohli využívat i v Exploreru, který žádnou z „vlastností gap“ nepodporuje.

Kód prvků layoutu, jež Autoprefixer vyrobí pro potřeby Exploreru, vypadá takhle:

```
.side-1 {
  -ms-grid-row: 1;
  -ms-grid-column: 1;
}

.content {
  -ms-grid-row: 1;
  -ms-grid-column: 3;
}

.side-2 {
  -ms-grid-row: 1;
  -ms-grid-column: 5;
}
```

MSIE totiž neumí ani žádnou z vlastností `*-area`. Autoprefixer tak automaticky spočítá umístění do patřičných sloupečků.

Pokud vám nesedí počty sloupců u vlastnosti `-ms-grid-column`, pak raději zopakuji, že Autoprefixer uměle přidává sloupečky, abychom mohli používat mezery `-gap`.

Demo jsme snad rozebrali do posledního kamínku. Tady je ještě jednou v celé kráse: vrdl.in/4rbvs.

Co Autoprefixer umí?

Následuje sumář aktuálně podporovaných a nepodporovaných vlastností gridu.

Dobrá zpráva zní, že těch druhých je poměrně málo.

Autoprefixer umí: Definování šablony mřížky

- `grid-template-columns` se přeloží do `-ms-grid-columns`.
- `grid-template-rows` se přeloží do `-ms-grid-rows`.
- `grid-template-areas` slouží k tomu, aby Autoprefixer pochopil, jak vypadá váš layout. Žádný kód ale negeneruje.
- `grid-template` je jen zkratka pro `grid-template-columns`, `grid-template-rows` a `grid-template-areas`. Přeloží se tedy do nich.

Umí: Zarovnávání

- `align-self` se přeloží do `-ms-grid-row-align`.
- `justify-self` se přeloží do `-ms-grid-column-align`.

Částečně umí: Umístění položky do mřížky

Překlad následujících vlastností funguje, ale nesmíte v nich použít záporná čísla:

- `grid-row-start` se přeloží do `-ms-grid-row`. Pokud chcete použít `span`, musíte definovat `grid-row-end`.
- `grid-column-start` se přeloží do `-ms-grid-column`. Pokud chcete použít `span`, musíte definovat `grid-row-end`.
- `grid-row-end`. Musíte ale definovat `grid-row-start`.
- `grid-column-end`. Musíte ale definovat `grid-row-start`.
- `grid-row` se přeloží do `-ms-grid-row`.
- `grid-column` se přeloží do `-ms-grid-column`.

Částečně umí: Definice pojmenovaných oblastí

Následující vlastnost funguje, ale každý potomek gridu musí mít unikátní jméno oblasti:

- `grid-area` – Autoprefixer z oblastí udělá explicitní zápis pomocí `grid-row-end` a `grid-column-end`.

Částečně umí: Definice mezer

- `grid-gap` a explicitní vlastnosti `grid-row-gap` nebo `grid-column-gap`.

Autoprefixer namísto `-gap` vygeneruje extra řádky nebo sloupečky. Fajn řešení, ne? Je ale důležité, abyste grid zapsali pomocí `grid-template-areas` a zároveň `grid-template-columns`.

Neumí vůbec: Vlastnost grid a automatické umístění
Tady máte zatím smůlu:

- `grid`, což je zkratka pro `grid-template-rows`, `grid-template-columns` a `grid-template-areas`. Doporučuje se namísto toho použít `grid-template`.
- Automatické umístění: `grid-auto-columns`, `grid-auto-rows` nebo `grid-auto-flow`.

Automatické umísťování ale u jednodušších layoutů udělat jde, jak za chvíli uvidíte.

Automatické umísťování do mřížky (autoplacement)

Autoplacement je další důležitá vlastnost gridu, Explorer ji neumí, ale i tady vám může částečně pomoci Autoprefixer.

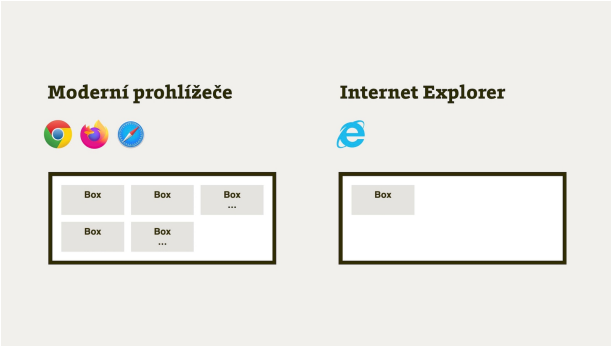
Pojďme tradičně na příklad. Chceme layout 3 × 2 prvky. HTML kód vypadá takto:

```
<div class="container">
  <p class="box">Box</p>
  <p class="box">Box</p>
  <p class="box">Box<br>…</p>
  <p class="box">Box</p>
  <p class="box">Box<br>…</p>
</div>
```

Díky automatickému umístění bude v moderních prohlížečích stačit definovat mřížku v CSS:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: auto auto;
  gap: 1rem;
}
```

Prvky se nám pěkně rozmístí. Jenže smůla, tohle nebude fungovat v Internet Exploreru.



Ale to je nepříjemné, automatické umístění v MSIE nefunguje.

Pro MSIE 10 a 11, ve kterých nějaká verze CSS gridu funguje, nám zde Autoprefixer přidá pseudotřídy:

```
.container > *:nth-child(1) {
  -ms-grid-row: 1;
  -ms-grid-column: 1;
}

.container > *:nth-child(2) {
  -ms-grid-row: 1;
  -ms-grid-column: 2;
}
```

... a tak dále.

Chcete autoplacement? Pozor na výjimky

Autoplacement rozhodně nefunguje ve všech použitích mřížky. Následuje seznam některých z možných problémů, které mohou vzniknout.

- *Nefunguje pro neznámý počet položek*
  Je tedy možné automatizace použít jen pro explicitní mřížky definované pomocí vlastnosti `grid-template-*`, nikoliv `grid-auto-*`.
- *Pozor na zpětné nasazení ve starých projektech*
  Doporučení zní: Nechte grid vypnutý a pomocí CSS komentářů jej zapínejte pouze pro nové deklarace.
- *Neumí to repeat() v kombinaci s auto-fill, auto-fit*
  I když MSIE funkci `repeat()` zvládá, klíčová slova `auto-fill` a `auto-fit` bohužel ne.

- *Vyberte si: Buď autoplacement, nebo manuální umístění v gridu*
  V moderních prohlížečích lze obojí kombinovat, v MSIE bohužel ne. Buď tedy budete všechny prvky gridu umísťovat ručně (použijte vlastnost `grid-template-areas` v definici gridu), nebo automaticky (bez `*-areas`).
- *Pozor na pseudoelementy*
  `::before` a `::after` uvnitř gridu vám v MSIE rozbijí mřížku, to se vsaďte.
- *V Media Queries nelze změnit jen grid-gap*
  Namísto toho je Autoprefixeru potřeba znovu deklarovat explicitní mřížku pomocí vlastností `grid-template-*`.

Neřeší to všechno, ale máme tady o jeden silný důvod navíc Autoprefixer použít.

Co potřebujete pro překlad gridu pro MSIE pomocí Autoprefixeru?

Potřebujete tyto suroviny:

- *Automatizaci*
  Gulp, Grunt, skripty v NPM nebo podobné nástroje, které umí využít Autoprefixer.
- *Zapnout podporu MSIE11*
  V Browserslistu, seznamu podporovaných prohlížečů, je nutné specifikovat také MSIE 11, případně i desátou verzi – například takto `> 1%, IE 11, IE 10`.
- *Zapnout grid a autoplacement*
  Zavolat Autoprefixer s parametrem `grid: 'autoplace'`, který zařídí podporu právě pro automatické umístění. Alternativou jsou řídicí komentáře přímo v CSS: `/* autoprefixer grid: autoplace */`.

Takto může vypadat konfigurace v automatizačním nástroji Gulp:

```
gulp.task('autoprefixer', () =>
  {
    return gulp.src('src/css/*.css')
      .pipe(autoprefixer({
        grid: autoplace
      }))
      .pipe(gulp.dest('dist/css'));
  }
);
```

V samotném CSS kódu se pak držte těchto pravidel:

- Definujte v gridu vždy kromě sloupců (`grid-template-columns`) nebo řádků také pojmenované oblasti: `grid-template-areas`.
- Používejte vlastnost `grid-template`, nikoliv zkratku `grid`.
- Vyhněte se pojmenovávání jmen řádků gridu.

Pokud na něco z toho zapomenete, Autoprefixer vás asi řádně potrápí, protože žádné prefixy nepřidá.

Možná vám to celé připadá dost složité. Nedivil bych se vám.

Na závěr ale chci akcentovat dvě důležitá fakta. Za prvé – jednodušší layouty nejsou pro Autoprefixer problematické a výše uvedené detailní znalosti pravděpodobně nebudete potřebovat.

Za druhé – v době vydání knihy velmi pravděpodobně nebudete muset podporovat Internet Explorer a tuto podkapitolu jste tedy mohli v klidu přeskočit. Z toho vyplývá, že bych vám v tomto odstavci klidně mohl nadávat a vy byste si dále žili klidný život. Bez mých nadávek a Exploreru.

Grid-template-rows/columns: explicitní mřížka

Vlastnosti `grid-template-rows` a `grid-template-columns` slouží k nastavení explicitní (tedy námi výslovně definované) mřížky v CSS gridu.

Jednoduchý příklad

→ vrdl.in/cssgtrc

Vezměme toto HTML:

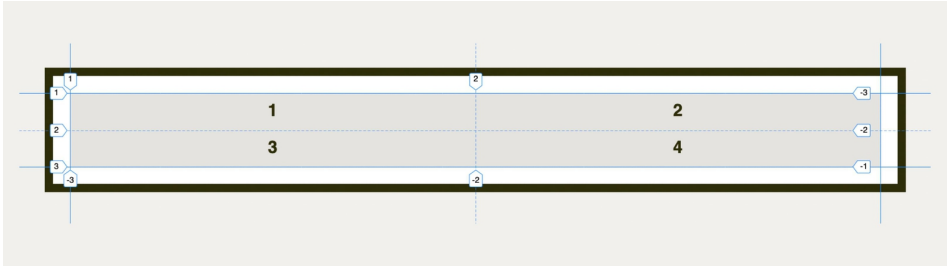
```
<div class="container">
  <p class="column">1</p>
  <p class="column">2</p>
  <p class="column">3</p>
  <p class="column">4</p>
</div>
```

Pokud bychom chtěli zajistit rozvržení do mřížky 4 × 4, použijeme následující CSS kód:

```
.container {
  display: grid;
  grid-template-columns: 50% 50%;
  grid-template-rows: auto auto;
}
```

Vysvětleme:

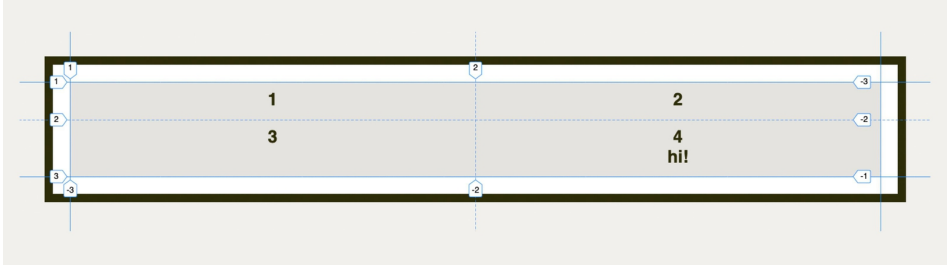
- `display:grid` – „zapíná“ mřížkové zobrazení.
- `grid-template-columns:50% 50%` – definuje dva sloupečky mřížky. Každý bude zabírat polovinu šířky rodičovského kontejneru.
- `grid-template-rows:auto auto` – definuje dva řádky mřížky. Hodnota `auto` říká, že se výška každého z nich počítá automaticky podle výšky obsahu.



CodePen: vrdl.in/xl4yh

Poznámka k hodnotě `auto` v definici řádků: Grid sjednocuje výšky položek v každém z nich, takže když změníme výšku jedné položky, její kolegyně se přizpůsobí.

```
<div class="container">
  <p class="column">1</p>
  <p class="column">2</p>
  <p class="column">3</p>
  <p class="column">4<br>hi!</p>
</div>
```



CodePen: vrdl.in/b18et

Co když je položek více, než definuje grid? (Implicitní vs. explicitní mřížka)

Teď se stane nepěkná věc – zákeřný frontendový kodér do HTML přidá pátou položku:

```
<div class="container">
  <p class="column">1</p>
  <p class="column">2</p>
  <p class="column">3</p>
  <p class="column">4</p>
  <p class="column">5</p>
</div>
```

Jak bude vypadat pátá položka zobrazená v gridu 2 × 2? V tomto případě stejně jako předchozí čtyři.

Algoritmus gridu jí přidělí 50% šířku a automatickou výšku.

Jiná by byla situace, pokud bychom změnili definici výšky položek gridu:

```
.container {
  grid-template-rows: 2rem 2rem;
}
```

Pátá položka zde pak nemá definovanou výšku a musí použít nějakou výchozí, v tomto případě opět `auto`.

CodePen: vrdl.in/xhecq

Rozměry položek vložených nad rámec počtu položek definovaných explicitním gridem, tedy vlastnostmi `grid-template-rows` a `grid-template-columns`, je možné určit vlastnostmi `grid-auto-columns` a `grid-auto-rows`, které definují implicitní grid.

Další možnosti zápisu gridu

Hodnoty v následující tabulce je možné aplikovat jak na vlastnost `grid-template-columns`, tak na její sestru `grid-template-rows`, tedy na sloupce i řádky layoutu.

Možnost	Ukázka hodnoty
Bez explicitního gridu	`none`
Kombinace jednotek	`150px auto 1fr 1fr`
Pojmenovávání linek	`[first] 150px [second] 1fr [end]`
Opakování	`repeat(12, 1fr)`
Masonry	`masonry`

Další řadu možností nabízí funkce `minmax()` a související hodnoty pro nastavení šířky podle obsahu, jako je `min-content`, `max-content` nebo `fit-content`.

Pojďme se ale podívat na možnosti hodnot z tabulky.

Bez explicitního gridu

Hodnota `none` tvoří výchozí stav:

```
.container {
  grid-template-columns: none;
}
```

Občas se ale může hodit použít ji i ve vlastním kódu, například když rušíme explicitní grid. Pokud to provedeme, platí pravidla pro implicitní, tedy nepřímo vyjádřenou mřížku – `grid-auto-columns` a `grid-auto-rows`.

Kombinace jednotek a jednotka fr

V gridu je možné pro definici řádků a sloupečků používat všechny možné jednotky, které už pro rozvržení v CSS používáte.

Je tady ale ještě jednotka `fr`. Jde o *fraction unit* a dá se o ní mluvit jako o podílu na zbytku.

Zápis může vypadat například takto:

```
.container {
  grid-template-columns: 150px 1fr 1fr 150px;
}
```

CodePen: vrdl.in/5vas0

`1fr` je v chování velice podobné číslu `1`, které používáte ve vlastnosti `flex` u flexboxu.

Pojmenovávání linek

Pojmenovávání linek se může hodit pro použití ve vlastnostech, které definují umístění prvků v gridu, jako je `grid-column`, `grid-row` nebo `grid-area`.

Každý sloupec nebo řádka je v gridu definovaná dvěma linkami.

Sloupce a řádky mřížky je možné si představit jako sloupce a řádky v tabulce. Linky jsou rámečky kolem buněk tabulky.

```
.container {
  display: grid;
  grid-template-columns: [first-col] 50% [second-col] 50% [last-col];
  grid-template-rows: [first-row] auto [second-row] auto [last-row];
}
```

V ukázce je tedy první položka gridu umístěná vodorovně na pozici mezi `first-col` a `second-col`. Svisle pak mezi `first-row` a `second-row`.

CodePen: vrdl.in/3sknv

Opakování

U složitějších mřížek by bylo nepříjemné zapisovat řadu stejných hodnot do řádky. Vezměme například dvanáctisloupcovou mřížku:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr 1fr 1fr 1fr;
}
```

Proto je tady funkce `repeat()`, která opakování zamezuje:

```
.container {
  display: grid;
  grid-template-columns: repeat(6, 1fr);
}
```

Zápisem `repeat(6, 1fr)` říkáme: „Napiš šestkrát po sobě hodnotu `1fr`."

Masonry, zděné rozvržení

Hodnota `masonry` je specifická, protože by v budoucnu měla zařídit nativní masonry (nebo zděný) layout.

Toto ale na konci února 2022 zatím podporuje jen prohlížeč Firefox v případě nastavení volby uživatelem.

Podpora v prohlížečích

Internet Explorer 11 vlastnosti `grid-template-rows`/`-columns` nepodporuje. Namísto `grid-template-columns` používá vlastnost `-ms-grid-columns` a místo `grid-template-rows` pak `-ms-grid-rows`. Toto se dá naštěstí obejít pomocí nástroje Autoprefixer.

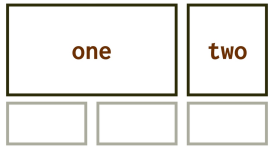
Explorer, tento stařičký, ale občas ještě používaný prohlížeč, také nezvládá automatické umísťování položek do mřížky. I to je možné alespoň částečně vyřešit pomocí Autoprefixeru.

V moderních prohlížečích nebudete mít s podporou problém.

Grid-template-areas: pojmenované oblasti

Vlastnost `grid-template-areas` slouží k pojmenovávání obdélníkových oblastí definovaných CSS gridem.

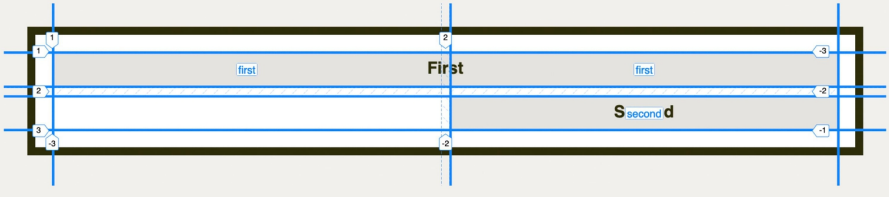
`grid-template-areas` vytváří oblasti, které jsou pak použitelné ve vlastnostech jako `grid-area` a dalších, sloužících k umísťování elementů do gridu.



→ vrdl.in/cssgta

V příkladu na obrázku pojmenováváme oblasti následovně:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr;
  grid-template-rows: auto auto;
  gap: 10px;
  grid-template-areas:
    "first first"
    ".     second";
}

.first {
  grid-area: first;
}

.second {
  grid-area: second;
}
```



Světle šedou barvou jsou vyznačené položky, modře pak uvidíte oblasti mřížky vyznačené ve Firefox DevTools.

Vysvětleme výše uvedený kód:

- Grid je definovaný jako dvousloupcový (`grid-template-columns: 1fr 1fr`) a dvouřádkový (`grid-template-rows: auto auto`).
- V `grid-template-areas` je pak seznam řetězců, které označují oblasti. To je to, oč zde běží.
- První řádek hodnot (`"first first"`) říká, že první dvě buňky layoutu patří do oblasti `first`.
- Tečka (`.`) uvádí prázdnou buňku, kde se nic zobrazovat nemá.
- Elementy `.first` a `.second` je pak potřeba ručně umístit do vyhrazených oblastí mřížky pomocí vlastnosti `grid-area`.

CodePen: vrdl.in/othcx

Následuje pár poznámek k vlastnosti `grid-template-areas`, které mě zaujaly při čtení specifikace.

Definování mřížky oblastmi

Pojmenovanými oblastmi je možné definovat i samotný grid. V některých případech tedy nebudeme potřebovat vlastnosti pro nastavení mřížky jako `grid-template-rows` a `grid-template-columns`.

Tento kód vytvoří stejnou mřížku jako ve výše uvedeném příkladu:

```
.container {
  display: grid;
  gap: 10px;
  grid-template-areas:
    "first first"
    ".     second";
}
```

CodePen: vrdl.in/u9z0j

Implicitně vytvořené oblasti a linky

Pojmenované oblasti souvisí s pojmenovanými linkami, které znáte z vlastnosti `grid-template-rows/-columns`.

Například definování oblasti pojmenované head automaticky vytvoří dvě linky – head-start a head-end.

A víte, co je ještě lepší? Definováním linek head-start a head-end ve vlastnostech grid-template-rows a grid-template-columns vytvoříte oblast head. Vezměme tento grid definovaný jako dva sloupce a dva řádky:

```
.container {
  display: grid;
  grid-template-columns: [head-start] 50% 50% [head-end];
  grid-template-rows: [head-start] auto [head-end] auto;
}
```

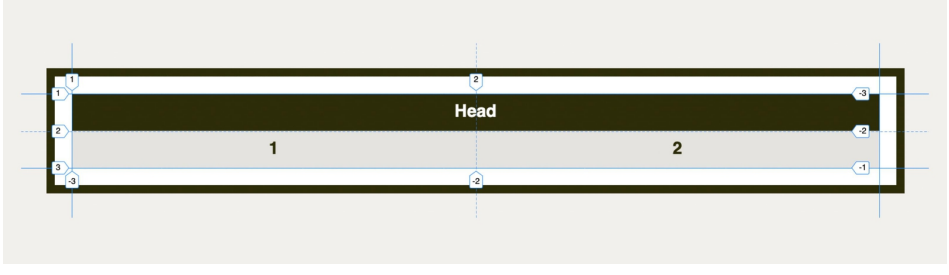
Jak je vidět, pomocí head-start a head-end se pokoušíme na prvním řádku definovat oblast head bez použití grid-template-areas.

HTML je následující:

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item item--head">Head</div>
</div>
```

Pak stačí pomocí vlastnosti grid-area prvek .item-head umístit:

```
.item--head {
  grid-area: head;
}
```



Oblast definovaná jen pomocí linek. Takhle to ukazuje Firefox v DevTools.

CodePen: vrdl.in/s84n0

Výhody definování oblastí mřížky

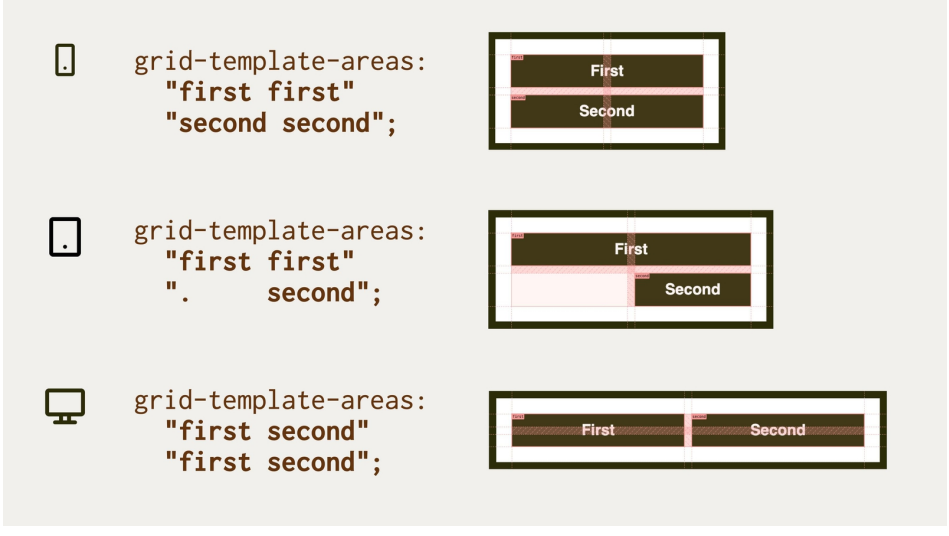
Oblasti gridu samozřejmě není nutné definovat pokaždé. Hodit se ale budou pro komplexnější layouty, které zároveň potřebujeme definovat a spravovat ručně. Typickým příkladem jsou rozvržení celostránková.

Vlastnost `grid-template-areas` je ale užitečná ještě ve dvou směrech. Za prvé zlidšťuje zápis layoutu. Je totiž lepší mluvit o oblasti `heading` než o „oblasti mezi první a osmou buňkou prvního řádku gridu“. Za druhé umožňuje pěkné kejkle s Media Queries.

Využití Media Queries

V dalším demíčku si pojďme ukázat věc, která se mi na `grid-template-areas` velmi líbí – kombinaci s Media Queries.

Je to jednoduché – oblasti je prostě možné různě přehazovat v rámci definovaného gridu.



V kombinaci s Media Queries se z definovaných oblastí pomocí vlastnosti grid-template-areas stává fantastický pomocník.

V tomto příkladu máme tyto responzivní deklarace:

```
.container {
  grid-template-areas:
    "first first"
    "second second";
}

@media screen and (min-width: 25em) {
  .container {
    grid-template-areas:
      "first first"
      ".     second";
  }
}

@media screen and (min-width: 37.5em) {
  .container {
    grid-template-areas:
      "first second"
      "first second";
  }
}
```

CodePen: vrdl.in/sdlrj

Podpora v prohlížečích

Opět je zde jediná potíž: Internet Explorer 11, který `grid-template-areas` neumí. Už ale také víte, že díky Autoprefixeru to nemusí být tak neřešitelné.

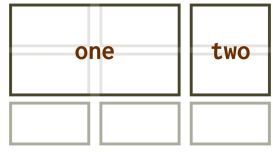
Moderní prohlížeče nemají s vlastností `grid-template-areas` žádný problém.

Grid-template: zkratka pro mřížku a oblasti

Vlastnost `grid-template` slouží k definování mřížky v CSS gridu.

Jde o zkratku pro vlastnosti `grid-template-rows`, `grid-template-columns` a také `grid-template-areas`.

Definování sloupečků a řádků



→ vrdl.in/cssgt

V prvním demu si ukážeme `grid-template` jednoduše, jako zkratku pro výše uvedené dvě vlastnosti:

```
.container {
  grid-template: auto auto / 2fr 1fr;
}
```

To odpovídá zápisu:

```
.container {
  grid-template-rows: auto auto;
  grid-template-columns: 2fr 1fr;
  grid-template-areas: none;
}
```

V živé ukázce si můžete vyzkoušet změnit některé hodnoty.

CodePen: vrdl.in/g1fsn

Sloupečky, řádky a pojmenované stopy

V hranatých závorkách můžeme při zapisování hodnoty `grid-template` uvést také pojmenované řádky a sloupce.

V kódu to vypadá následovně:

```
.container {
  grid-template:
    [row-1] auto [row-2] auto
    /
    [col-1] 2fr [col-2] 1fr;
}
```

To odpovídá zápisu:

```
.container {
  grid-template-rows: [row-1] auto [row-2] auto;
  grid-template-columns: [col-1] 2fr [col-2] 1fr;
  grid-template-areas: none;
}
```

Opět je k dispozici živá ukázka.

CodePen: vrdl.in/d81p2

Sloupečky, řádky a pojmenované oblasti

V poslední ukázce uvidíte, že je také možné definovat rozměry gridu spolu s pojmenováním oblastí:

```
.container {
  display: grid;
  grid-template:
    "heading heading" auto
    "one two" auto
    /
    2fr 1fr;
}
```

Je to už docela složité. Nejprve se definují řádky – jejich jména a výšky – a pak za lomítkem šířky sloupečků. Odpovídá to následujícímu:

```
.container {
  grid-template-rows: auto auto;
  grid-template-columns: 2fr 1fr;
  grid-template-areas:
    "heading heading"
    "one two";
}
```

CodePen: vrdl.in/4lx3u

Co je dobré vědět?

Je zde pár kulišáren, na které bych vás rád upozornil:

- Všimněte si, že `grid-template` nastavuje `grid-template-areas:none`, takže resetuje i případné dříve nastavené oblasti.
- Funkce `repeat()` zde není povolena, protože se dává přednost vizuálnímu vyjádření mřížky, takzvanému „ASCII artu“.
- Stejnou syntaxi má další zkratka – vlastnost `grid`, která ovšem navíc umožňuje nastavení rozměrů implicitního gridu (vlastnosti `grid-auto-columns`, `grid-auto-rows` a `grid-auto-flow`), takže je pravděpodobné, že můžete chtít použít spíše právě ji.

Podpora v prohlížečích

Internet Explorer 11 vlastnost nepodporuje. Můžete ji zkusit automaticky převádět díky Autoprefixeru.

Pokud ale podporu MSIE s využitím Autoprefixeru plánujete, moje doporučení zní: Používejte vlastnost `grid-template`, nikoliv zkratku `grid` a vyhnete se problémům.

Moderní prohlížeče nemají s `grid-template` problém.

Grid-auto-rows/columns: implicitní mřížka

Vlastnosti `grid-auto-rows` a `grid-auto-columns` použijete pro tvorbu implicitního, čili výslovně nedefinovaného gridu v CSS.

Jsou tedy doplňkem nám už známých vlastností `grid-template-rows` (pro definici řádků mřížky) a `grid-template-columns` (pro definici sloupců), které naopak slouží pro vytváření gridu explicitního, čili autorsky definovaného.

→ vrdl.in/cssgarc

Tyto vlastnosti jsou užitečné pro implicitní řádky (`grid-auto-rows`), a to v případech, kdy máte v HTML neznámý počet položek. Řádky se vám v layoutu tvoří samy a vy nemusíte definovat mřížku pro různé počty položek v ní.

Možné hodnoty

Vlastnostem implicitního gridu, tedy `grid-auto-rows` a také `grid-auto-columns` můžete přiřazovat prakticky stejné hodnoty jako jejich „explicitním kolegyním" – `grid-template-rows` nebo `grid-template-columns`.

Příklady se budou točit kolem definování řádků, ale samozřejmě platí i pro sloupce:

Výchozí hodnota je `auto`:

```
grid-auto-rows: auto;
```

Říká se tím, že výšku řádku definuje výška obsahu v buňkách.

Jedna hodnota pro všechny řádky

```
grid-auto-rows: 100px;
grid-auto-rows: 10%;
grid-auto-rows: 1fr;
```

V takovém případě prostě natvrdo nastavíme minimální výšku buněk mřížky. Obsah to samozřejmě může svou výškou přebít.

Více hodnot, které se při větším počtu sloupců opakují

```
grid-auto-rows: 100px 200px;
grid-auto-rows: 10% 33.3%;
grid-auto-rows: 1fr minmax(100px, auto);
```

Hodnota `100px 200px` říká, že první a pak každý lichý nedefinovaný, a tedy implicitní řádek bude mít výchozí výšku `100px`, sudé řádky pak `200px`.

Klíčová slova

```
grid-auto-columns: min-content;
grid-auto-columns: max-content;
```

Je samozřejmě možné používat funkce a klíčová slova, stejně jako při definici explicitní mřížky.

Ukázky máme tedy za sebou. Jak ale onen implicitní grid vzniká a kde je možné tyto vlastnosti využít?

Příklad: Více položek v gridu, než je definovaných řádků

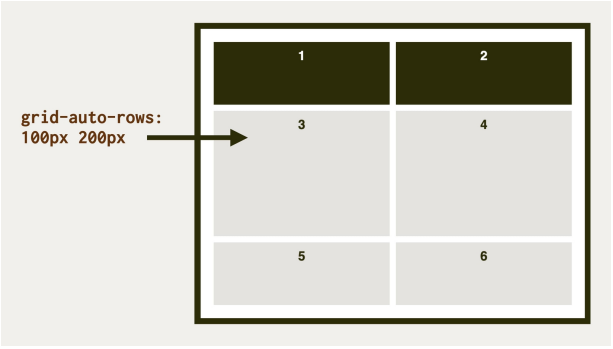
Jak už jsem říkal – vlastnosti `grid-auto-` častěji použijete pro řádky layoutu, tedy `grid-auto-rows`. Může se vám totiž snadno stát, že v mřížce je předem daný počet sloupečků, ale neznámý počet položek, a tedy řádků:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr;
}
```

Vlastnost `grid-template-rows` umí definovat rozměry známých řádků. Nezvládá ovšem definovat opakování hodnot.

Pokud bychom chtěli střídat výšku `100px` a `200px` pro liché a sudé řádky, přičemž počet řádků neznáme, i pro tohle je výborná vlastnost `grid-auto-rows`:

```
.container {
  grid-auto-rows: 100px 200px;
}
```



Vlastnost grid-auto-rows v akci. Dává rozměry položkám, se kterými nepočítá explicitně definovaná mřížka.

CodePen: vrdl.in/08zwo

Příklad: Umístění položky mimo explicitní grid

Další možností, jak může explicitní grid vzniknout, je umístění položky zcela mimo definovanou mřížku. Řekněme, že ji definujeme jako 2 × 2:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr;
  grid-auto-columns: 1fr;
}
```

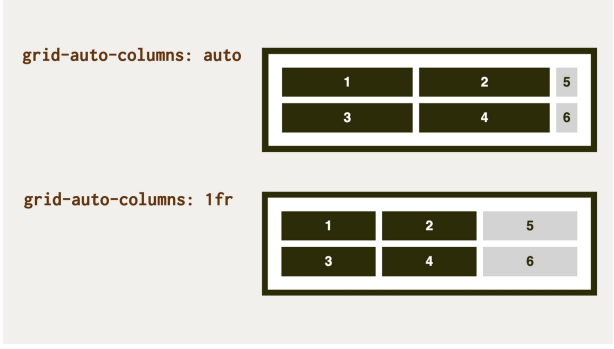
Jenže v HTML máme šest a ne čtyři položky. A co víc – pátou a šestou položku nezbedný kodér umístil na pozice v mřížce, které neexistují:

```
.column.fifth {
  grid-column: 3;
  grid-row: 1;
}

.column.sixth {
  grid-column: 3;
  grid-row: 2;
}
```

Prohlížeč nám při pokusu o umístění do třetího sloupce, tedy na pozici nedefinovanou explicitním gridem, grid rozšíří. Ale rozšíří jej mřížkou implicitní. Výchozí rozměry prvků nám tedy zřejmě nebudou vyhovovat.

Opravíme to až touto deklarací:

```
.container {
  grid-auto-columns: 1fr;
}
```



Rozdíl mezi nedefinovanou mřížkou (v případě zápisu grid-auto-columns:auto) a mřížkou definovanou (grid-auto-columns:1fr), ve které pátá a šestá položka dostanou definovanou šířku.

CodePen: vrdl.in/fcpvj

Podpora v prohlížečích

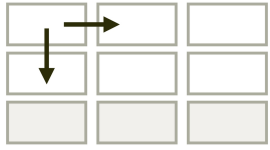
Vlastnosti `grid-auto-rows` a `grid-auto-columns` podporuje kdejaký prohlížeč, včetně Internet Exploreru od verze 10. Jupí! V MSIE je potřeba používat vlastnosti pojmenované jako `-ms-grid-rows` a `-ms-grid-columns` nebo to řešit Autoprefixerem.

Grid-auto-flow: způsob umísťování do mřížky

Vlastnost `grid-auto-flow` určuje, jak bude fungovat algoritmus automatického umísťování prvků do mřížky.

Běžně vlastnost slouží ke změně směru vykreslování rozvržení. Možností použití je ale celá řada.

Nejzajímavější je asi hodnota `dense`, vyvolávající „zahuštěné“ vykreslení, kdy prohlížeč dá přednost kompaktnosti vykreslení položek před jejich pořadím definovaným v kódu.



→ vrdl.in/cssgaf

Hodnoty

Hodnota	Co dělá?
`row`	Výchozí hodnota. Prvky layoutu se vykreslují po řádcích.
`column`	Prvky layoutu se vykreslují po sloupcích.
`dense`	Vykresluje se po řádcích, ale prohlížeč může vyplnit mezery změnou pořadí prvků.
`column dense`	Vykresluje se po sloupcích, ale prohlížeč může vyplnit mezery změnou pořadí prvků.

Co teď? Vzhůru do ukázek!

Příklad: změna směru pomocí grid-auto-flow:column

Tohle je jednoduché, prostě se jen změní směr vykreslování z řádků na sloupečky.

Řekněme, že máme šest položek v layoutu:

```
<div class="container">
  <p class="column">1</p>
  <p class="column">2</p>
  <p class="column">3</p>
  <p class="column">4</p>
  <p class="column">5</p>
  <p class="column">6</p>
</div>
```

Layout je definovaný jako mřížka o třech sloupcích a dvou řádcích:

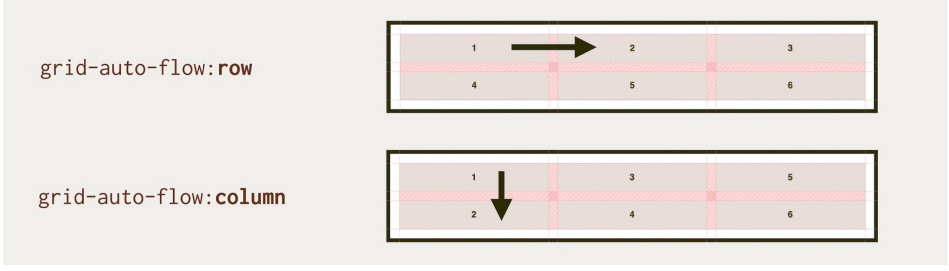
```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
  gap: 10px;
}
```

Výchozí směr vykreslení je `grid-auto-flow:row`, tedy po řádcích.

CodePen: vrdl.in/hiyjr

Můžeme jej ale změnit do svislého (sloupcového) směru:

```
.container {
  grid-auto-flow: column;
}
```



Změnit směr rozvržení mřížkou jde snadno s pomocí vlastnosti grid-auto-flow.

CodePen: vrdl.in/i8mbo

V CodePenu si to můžete zkusit také pro vlastnosti implicitního gridu: `grid-auto-rows` a `grid-auto-columns`, stačí odstranit komentář v CSS.

Příklad: zahuštěné vykreslování s grid-auto-flow:dense

Hodnota `dense` zajistí „zahuštěné“ vykreslení. Ve směru vykreslení nezůstávají v layoutu mezery mezi prvky. Mezery pak zůstanou až na konci layoutu.

Důležité je, že prohlížeč může po nastavení `grid-auto-flow:dense` změnit pořadí vykreslení prvků.

Místo dlouhých řečí ale pojďme na příklad. Opět zde máme šest položek:

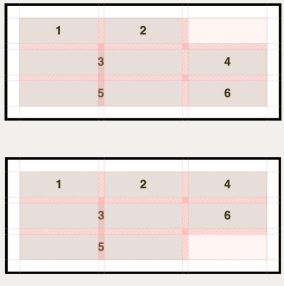
```
<div class="container">
  <p class="column column-1">1</p>
  <p class="column column-2">2</p>
  <p class="column column-3">3</p>
  <p class="column column-4">4</p>
  <p class="column column-5">5</p>
  <p class="column column-6">6</p>
</div>
```

Layout je definovaný takto:

```
.container {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(120px, 1fr));
  gap: 10px;
  grid-auto-flow: dense;
}
```

Vysvětlíme:

- Ve vlastnosti `grid-template-columns` uvádíme, že je možný libovolný počet sloupců (funkce `repeat()`), ale každý musí být minimálně 120px široký – `minmax(120px, 1fr)`.
- `grid-auto-flow: dense` říká prohlížeči: Vykresli to zahuštěně, takže nezáleží na pořadí prvků, ale na tom, aby ve směru layoutu (což je tady `row`) nezůstaly mezery.



Běžné umístění položek

Zahuštěné umístění položek
`grid-auto-flow: dense`

Zahuštěné vykreslení (hodnota dense) prostě dává přednost hustotě před pořadím.

CodePen: vrdl.in/2caxb

Přístupnost: pozor na změnu pořadí

Pomocí deklarace `grid-auto-flow:dense` může automatika prohlížeče odlišit pořadí prvků v kódu od vizuálního pořadí.

Pozor na to! Dokument pak může zlobit při navigaci z klávesnice nebo pro odečítače obrazovky. Snažte se proto tuto deklaraci používat jen pro prvky, u kterých na pořadí zase tak moc nezáleží.

Více to rozebírám v části CSS layout a přístupnost osmé kapitoly.

Podpora v prohlížečích

Vlastnost `grid-auto-flow` a její hodnoty jsou plně podporovány ve všech prohlížečích s jedinou výjimkou – Internet Exploreru.

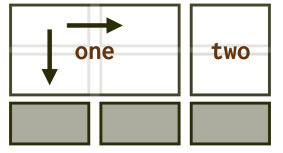
V MSIE nám bohužel nepomůže v případě této vlastnosti ani Autoprefixer nebo jiný nástroj. Můžeme na něj ale vyzrát vhodným napsáním kódu tak, abychom měli pod kontrolou desktopové zobrazení, které Explorer používá, a ostatním prohlížečům pomocí vlastnosti `grid-auto-flow` nechali jen menší displeje.

Grid: zkratka pro definici mřížky

`grid` je zkratka pro definování všech vlastností CSS gridu.

V jedné deklaraci můžete nastavit jen jeden z obou možných gridů:

- *explicitní*, výslovně definovaný pomocí vlastností `grid-template-rows`, `grid-template-columns` a `grid-template-areas`
- *implicitní*, definovaný pomocí vlastností `grid-auto-rows`, `grid-auto-columns` a `grid-auto-flow`



→ vrdl.in/cssg

Je to mocná zbraň, která dokáže urychlit práci zkušených mřížkových ninjů, ovšem začínajícím bych ji raději doporučoval nepoužívat.

Pojďme se podívat na několik ukázek.

Definice explicitního gridu

V tomto zápisu oddělujeme hodnoty pro vlastnosti `grid-template-rows` od `grid-template-columns` pomocí lomítka:

```
grid: <grid-template-rows> / <grid-template-columns>;
```

Například:

```
.container {
  grid: auto auto / 2fr 1fr;
}
```

Odpovídá tomuto zápisu:

```
.container {
  grid-template-rows: auto auto;
  grid-template-columns: 2fr 1fr;
  grid-template-areas: none;
}
```

Je to prostě úplně stejné jako u zkratky `grid-template`.

Implicitní grid a auto-flow

To, zda v daném směru definujete implicitní grid, se pozná podle klíčového slova `auto-flow`:

```
.container {
  grid: auto-flow 1fr / 100px 100px;
}
```

Zápis je ekvivalentní tomuto:

```
.container {
  grid-auto-flow: row;
  grid-auto-rows: 1fr;
  grid-auto-columns: auto;
  grid-template-rows: none;
  grid-template-columns: 100px 100px;
  grid-template-areas: none;
}
```

Vysvětlíme:

- `grid-auto-flow:row` – grid se bude vykreslovat po řádcích. Jak můžete vědět ze studia vlastnosti `grid-auto-flow`, hodnota `grid-auto-flow:row` je v prohlížečích výchozí, takže bychom ji nemuseli nastavovat.
- `grid-auto-rows:1fr` – implicitní grid bude mít výšku řádků `1fr`.
- `grid-auto-columns:auto` – implicitní grid nemá definované sloupce, takže se budou řídit šířkou obsahu buňky.
- `grid-template-rows:none` – buňky gridu nemají ve směru řádků nijak nastavené rozměry.
- `grid-template-columns:100px 100px` – buňky mají v explicitním gridu nastavenou šířku `100px` a tvoří dva sloupce.

CodePen: vrdl.in/beuxn

Podobné to bude u opačného zápisu:

```
.container {
  grid: 1fr / auto-flow 100px 100px;
}
```

Ten je ekvivalentní následujícímu:

```
.container {
  grid-auto-flow: column;
  grid-auto-rows: auto;
  grid-auto-columns: 100px;
  grid-template-rows: 1fr;
  grid-template-columns: none;
  grid-template-areas: none;
}
```

Zahuštěné vykreslování, klíčové slovo dense

Pokud jste viděli vlastnost `grid-auto-flow`, určitě vás zaujala hodnota `dense`.

Můžeme ji nastavit i pomocí zkratky `grid`. Následující dva kousky kódu budou totožné:

```
.container {
  grid: auto-flow dense 100px / 1fr 2fr;
}
```

```
.container {
  grid-auto-flow: row dense;
  grid-auto-rows: 100px;
  grid-template-columns: 1fr 2fr;
}
```

Co je ještě dobré vědět?

- U všech zkratek v CSS platí, že vlastnosti, které nedefinujeme, zkratka nastaví na jejich výchozí hodnoty. To může být občas nebezpečné.
- Vlastnost pro mezeru mezi buňky layoutu – `gap` – nelze přes zkratku `grid` nastavit, a není tudíž jejím použitím resetována.

Podpora v prohlížečích

Zkratku `grid` zvládají všechny prohlížeče kromě Internet Exploreru. Pokud na projektu potřebujete zapisovat CSS grid i pro tento prohlížeč, doporučuji se zkratce `grid` vyhnout.

Tvůrci nástroje Autoprefixer doporučují použití samotných vlastností nebo maximálně zkratky `grid-template`.

Grid-row/column: umístění položky do mřížky

V textu se zaměříme na vlastnosti `grid-column` a `grid-row`, které slouží k umísťování položek v definované mřížce.

Vlastnosti `grid-column`/`-row` jsou pouhými zkratkami pro snadnější zápis jiných:

- `grid-column-start`
- `grid-column-end`
- `grid-row-start`
- `grid-row-end`

→ vrdl.in/cssgrc

Používání zkratek je praktičtější, jak sami uvidíte. Takto se uvádějí obecně:

```
grid-row: <grid-row-start> / <grid-row-end>;
grid-column: <grid-column-start> / <grid-column-end>;
```

Možné hodnoty

Možnost	Ukázka hodnoty
Čísla linek	`1 / 2`
Jména linek	`first-line / second-line`
Rozsah	`1 / span 2`
Jedno číslo	`3`
Záporná čísla	`-1 / -3`
Automatické umístění	`auto`

Čísla linek

Tady je to jednoduché. Potřebujete dvě hodnoty – pro začátek a konec pozice v mřížce.

Řekněme, že máme tuto mřížku o 3 × 3 buňkách:

```
.container {
  display: grid;
  gap: 10px;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 2rem 2rem 2rem;
}
```

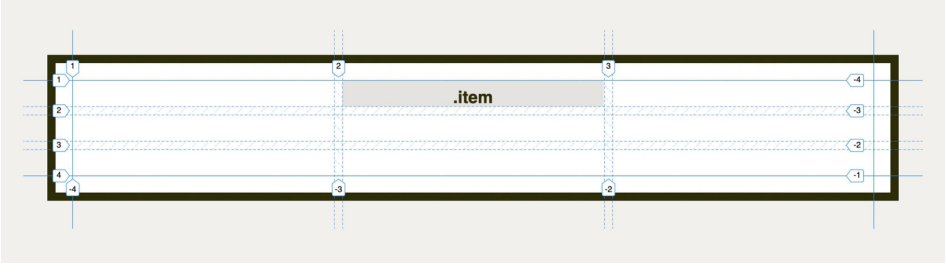
Pokud bychom položku chtěli umístit do druhého sloupce a přes druhý a třetí řádek, zapíšeme to následovně:

```
.item {
  grid-column: 2 / 3;
  grid-row: 2 / 4;
}
```

Je to totéž, jako bychom napsali:

```
.item {
  grid-column-start: 2;
  grid-column-end: 3;
  grid-row-start: 2;
  grid-row-end: 4;
}
```

Všimněte si, že řádky končíme na čísle 4, i když je máme jen tři. Je to dobrá připomínka toho, že se zde neodkazujeme na čísla řádků či sloupců, ale na čísla linek, které grid definují.



Vlastnosti grid-column a grid-row ve Firefoxu.

CodePen: vrdl.in/muo5r

Jména linek

Podobná situace je u pojmenovaných linek. Máme stejný příklad jako výše, jen s pojmenovanými linkami:

```
.container {
  display: grid;
  gap: 10px;
  grid-template-columns:
    [first-col] 1fr [second-col] 1fr [third-col] 1fr [end];
  grid-template-rows:
    [first-row] 2rem [second-row] 2rem [third-row] 2rem [end];
}
```

Umísťujeme pak elegantně a o fous čitelněji:

```
.item {
  grid-column: second-col / third-col;
  grid-row: second-row / end;
}
```

Výsledek však pochopitelně bude stejný.

CodePen: vrdl.in/xis4h

Rozsah

Klíčové slovo `span` slouží k určení rozsahu, jaký v mřížce buňka zabírá.

Podívejme se teď na tentýž příklad jako v první ukázce, jen si jeho pozici definujme pomocí rozsahu:

```
.item {
  grid-column: 2 / span 1;
  grid-row: 2 / span 2;
}
```

Vysvětlíme:

- Zápis `span 1` u `grid-column` znamená „zabírá jeden sloupec“. Tady je to zbytečné, fungovalo by to i bez tého hodnoty.
- Zápis `span 2` u `grid-row` už je zajímavější. Říkáme tím, že má zabrat dva řádky buněk mřížky. Může to tedy být čitelnější než zápis `grid-row:2/4`.

CodePen: vrdl.in/iks84

Jedno číslo

Občas je možné vidět zápis `grid-row` a `grid-column` jen s jedním číslem:

```
.item {
  grid-column: 2;
}
```

Funguje to tak, jak asi očekáváte. Prohlížeče v tomto případě automaticky umísťují prvky DOMu na jednu buňku mřížky (`span 1`), takže jim stačí uvést pozici začátku vykreslování.

CodePen: vrdl.in/jso2c

Záporná čísla

Záporná čísla ve vlastnostech `grid-row` a `grid-column` počítají umístění zprava nebo zezdola. Pokud bychom tedy chtěli stejnou svislou pozici položky jako v našem příkladu, jen definovanou zápornými čísly, zapsali bychom ji takto:

```
.item {
  grid-column: 2 / 3;
  grid-row: -1 / -3;
}
```

Kódem `grid-row:-1/-3` říkáme: Chci umístění od první linky mřížky zezdola až po třetí linku mřížky v tomtéž směru.

CodePen: vrdl.in/wpdxy

Automatické umístění

Hodnota `auto` vypíná ruční umísťování a nechá pracovat algoritmus „auto-placementu“, postupného automatického umístění do existujících buněk rozvržení.

Vezměme stejný příklad:

```
.item {
  grid-column: 2 / 3;
  grid-row: auto;
}
```

Prvek bude dále umístěn do druhého sloupce, ale v tomto případě do prvního řádku. Algoritmus automatického umísťování zde vždy začíná a dosud nevykresloval žádné jiné prvky mřížky, které by mohly pozici posunout.

CodePen: vrdl.in/muo5r

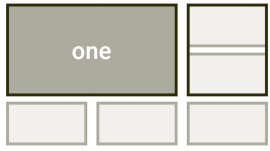
Podpora

Moderní prohlížeče s těmito vlastnostmi nemají problém. Horší je to samozřejmě v MSIE, ale to nevadí, protože podobné vlastnosti tam existují. Autoprefixer nám pomůže a např. `grid-row` se přeloží do `-ms-grid-row`.

Grid-area: umístění položky do oblasti

CSS vlastnost `grid-area` slouží k umísťování položek v definovaném CSS gridu.

Jde o zkratku pro vlastnosti `grid-column-start`, `grid-column-end` a `grid-row-start`, `grid-row-end` nebo také zkratku pro zkratky `grid-column` a `grid-row`, aby to nebylo úplně jednoduché.



→ vrdl.in/cssga

Může mít jednu až čtyři hodnoty. Obecně vypadá zápis asi takto:

```
grid-area:
  <grid-row-start> / <grid-column-start> /
  <grid-row-end> / <grid-column-end>;
```

Jedna hodnota

Umístění do už pojmenované oblasti mřížky je snadné. Nejprve si nadefinujeme mřížku a pomocí `grid-template-areas` pojmenujeme některé z jejích oblastí:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr;
  grid-template-rows: auto auto;
  gap: 10px;
  grid-template-areas:
    "first first"
    ".     second";
}
```

Pak už do oblastí `first` a `second` můžeme umísťovat:

```
.first {
  grid-area: first;
}
```

Je to zápis ekvivalentní k tomuto:

```
.first {
  grid-row-start: first;
  grid-column-start: first;
  grid-row-end: first;
  grid-column-end: first;
}
```

CodePen: vrdl.in/othcx

Další možné zápisy

Automaticky umísťuj do další buňky mřížky:

```
.item {
  grid-area: auto;
}
```

Umísti oblast do řádku 2 a sloupce 4:

```
.item {
  grid-area: 2 / 4;
}
```

Umísti oblast do třetího sloupce a roztáhni se do všech řádků (od prvního (1) do posledního -1):

```
.item {
  grid-area: 1 / 3 / -1;
}
```

Na různé další možnosti hodnot, včetně demíček na CodePenu, se podívejte do příručky ke zkratkám `grid-column` a `grid-row`.

Přístupnost: pozor na změnu pořadí

Pomocí vlastnosti `grid-area` můžete zcela odlišit pořadí prvků v kódu od vizuálního pořadí. Pozor na to, dokument se pak může špatně používat při navigaci z klávesnice nebo při práci přes odečítač obrazovky.

Více to rozebírám v části CSS layout a přístupnost osmé kapitoly.

Jednotka fr

Jedná se o flexibilní jednotku, která reprezentuje podíl na volném prostoru v CSS gridu. Proto `fr`, jako fraction unit.

Využívá se ve vlastnostech popisujících mřížku, jako jsou `grid-template-columns/-rows` nebo `grid-auto-columns/-rows`.

→ vrdl.in/fr

Ukažme si nějaký kód:

```
.container {
  display: grid;
  grid-template-columns: 1fr 2fr;
}
```

Prostor se rozdělí na tři díly, ale vykreslí do dvou buněk gridu. První bude zabírat třetinu šířky, druhá dvě třetiny.

CodePen: vrdl.in/i6leb

Raději zdůrazněme, že šířka buňky se ani zde nikdy nezmenší pod šířku obsahu. Rozdělení `1fr 2fr` tedy bude platit jen v případě dostatku prostoru.

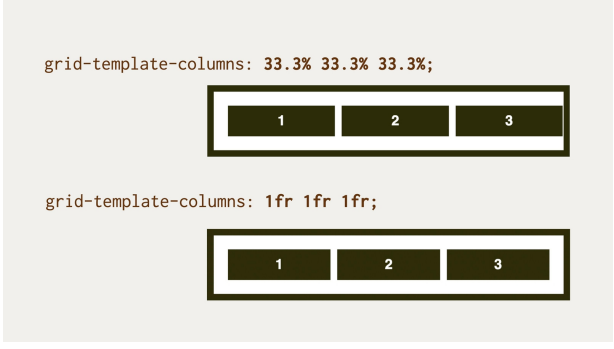
Je (a není) to jako procenta

Zjednodušeně by se tahle jednotka dala přirovnat k procentům (%), je ale lepší než procenta.

Demo to ostatně ukáže v celé kráse. Máme dva kontejnery vysázené gridem, které mají tři vnitřní prvky. První layout je definován v procentech, druhý v jednotce `fr`.

```
.container-percentage {
  display: grid;
  grid-template-columns: 33.3% 33.3% 33.3%;
  gap: 10px;
}

.container-fr {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  gap: 10px;
}
```

Na pohled by to mohlo fungovat stejně, ale to opravdu jen na první pohled.



Jednotky fr a %. Kdo s koho?

Výhoda `fr` oproti procentům? Nemusíme počítat prvky gridu a řešit nastavení box modelu.

Procentuální grid ve výše uvedeném demu nastavujeme pomocí deklarace `grid-template-columns: 33.3% 33.3% 33.3%`. Může to být fajn, ale procentuální hodnotu musíme vypočíst podle počtu sloupců v HTML. No a pokud by se nám jejich počet změnil, nezbývá než upravovat také CSS.

Další problém je v box modelu, způsobu počítání šířky. Jak vidíte u prvního prvku v CodePenu, k procentuální šířce se musí ještě připočíst vnitřní okraj (`padding`). Bez změny počítání box modelu pomocí vlastnosti `box-sizing` nám prvky přetečou z rodičovského kontejneru.

CodePen: vrdl.in/0v82w

Pokud vlastnost `box-sizing` neznáte, dostudujte si ji v mém textu na Vzhůru dolů. vrdl.in/cssbs

Podíl na zbytku

Jednotka `fr` tvoří podíl na zbytku plochy, kterou nezabírají prvky s rozměry nastavenými běžnými jednotkami délky, jako jsou `%`, `px`, `em`, `rem` a další:

Jednotka fr a podíl na zbytku plochy nevyužité jinými jednotkami.

Pomocí `grid-template-columns: 80px 5em 2fr 1fr` definujeme čtyřsloupcovou mřížku. První sloupec má fixní šířku `80px`. Druhý má šířku `5em`, takže jeho šířka bude záležet na velikosti písma rodičovského prvku. Pokud nám zbude nějaký prostor, prohlížeč jej vydělí třemi (`2fr` + `1fr`) a rozpočítá na dvě buňky definované podílovou jednotkou.

CodePen: vrdl.in/jhfwd

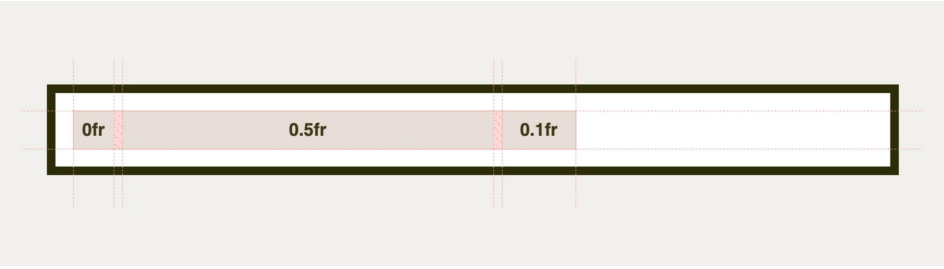
Nulová a desetinná čísla. Hodnoty, se kterými je legrace

Podívejme se ještě na jeden speciální příklad, který ukazuje použití nulové (`0fr`) a desetinné (`0.1fr`) hodnoty.

Mřížku definujeme takto:

```
.container {
  grid-template-columns: 0fr 0.5fr 0.1fr;
}
```

Pokud je nastavená nulová hodnota (`0fr`), vykreslí prohlížeč sloupec v šířce obsahu.



Jednotka fr si dobře rozumí i s desetinnými čísly.

Poměrně užitečné jsou také desetinné hodnoty. Jejich součet je `0.6fr`, takže zanechává `0.4fr` prostoru prázdného.

Obojí se myslím může docela hodit, nemám pravdu?

CodePen: vrdl.in/i47sx

Jo a – záporné hodnoty k jednotce `fr` přidávat nezkoušejte. K ničemu to nevede, už jsem to ověřil.

Poznámky na závěr

- Asi jste postřehli, že `grid-template-columns:1fr` se chová podobně jako bezjednotkový údaj ve flexboxu, např. `flex:1`.
- `fr` není jednotka délky a nedá se s nimi ani kombinovat. Nemůžeme tedy použít funkci calc() a výraz jako `calc(1fr - 20px)`.

Podpora v prohlížečích

Podpora je skvělá. Nevím o žádném mezi uživateli rozšířeném prohlížeči, který by jednotku `fr` v kombinaci s CSS gridem nezvládal. A to do této skupiny prohlížečů k únoru 2022 počítám i Internet Explorer 11.

Další jednotky v CSS (`%`, `px`, `em`, `rem`) probírám v samostatném textu poslední kapitoly.

Repeat(), auto-fill, auto-fit

Funkce (nebo přesněji řečeno „zápis“) `repeat()` slouží k usnadnění opakujících se předpisů pro sloupce nebo řádky mřížky v CSS gridu.

Zápis můžete realizovat například takto:

```
.container {
  grid-template-columns: repeat(4, 1fr);
}
```

→ vrdl.in/cssrep

Uvedené má pak stejný výsledek jako ručně psaná verze:

```
.container {
  grid-template-columns: 1fr 1fr 1fr 1fr;
}
```

CodePen: vrdl.in/lzrej

Možných hodnot je ale více:

```
repeat(4, [col-start] 1fr [col-end])
repeat(4, 10px [col-start] 30% [col-middle] auto [col-end])
repeat(auto-fill, minmax(100px, 1fr));
```

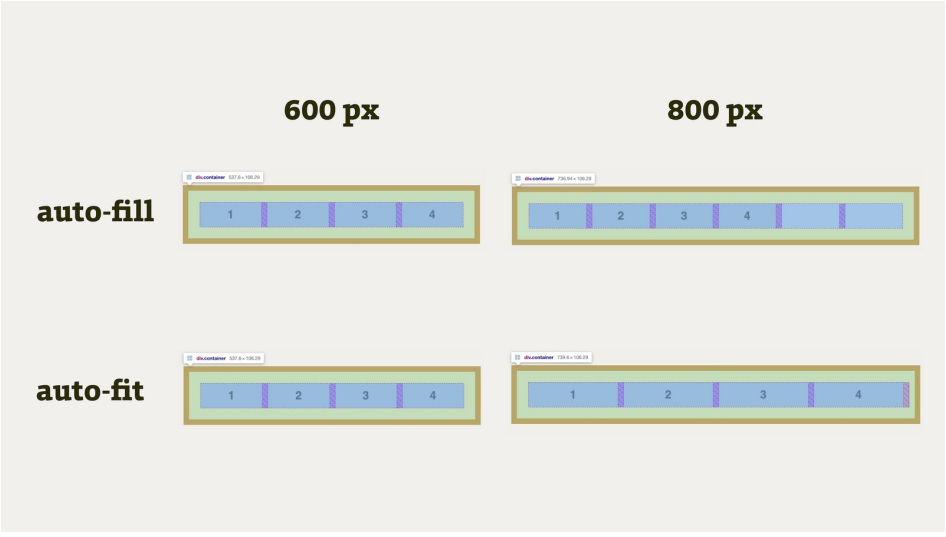
V druhém parametru zápisu `repeat()` prostě můžete uvést libovolnou deklaraci stopy mřížky, kterou budete znát už například z vlastností `grid-template-rows/-columns`.

Co když ale chcete deklaraci sloupců nebo řádků opakovat, jenže ještě nevíte kolikrát? Třeba prostě nevíte, kolik bude mít rodičovský prvek v HTML potomků.

Tady vstupují do hry zajímavá klíčová slova `auto-fill` a `auto-fit`.

Automatické opakování s auto-fill a auto-fit

Použití zápisu `repeat()` v kombinaci s klíčovými slovy `auto-fill` a `auto-fit` se velmi hodí, pokud neznáte počet položek. Následující obrázek snad napoví, čím se liší.



Hodnota auto-fill přidává do volného prostoru neviditelné buňky, naproti tomu auto-fit prioritizuje roztahování těch už vykreslených.

Rozdíly jsou zhruba následující:

- Oba zápisy v `repeat()` přidávají nové neviditelné buňky mřížky tak, aby se vyplnil volný prostor.
- `auto-fill` se snaží vyplnit prostor prázdnými buňkami.
- `auto-fit` při vykreslování nové prázdné buňky eliminuje, a tedy umožní vyplnit prostor roztažením (napasováním, proto „fit“) velikosti stávajících buněk mřížky.

Myslím si, že obě klíčová slova dává smysl použít jen s funkcí `minmax()`, aby se buňky mohly roztahovat od minima po maximum velikosti.

Příklad s auto-fill

`auto-fill` se snaží při roztahování prostoru přidávat neviditelné buňky.

```
.container {
  grid-template-columns: repeat(auto-fill, minmax(100px, 1fr));
}
```

CodePen: vrdl.in/lx9fc

Příklad s auto-fit

`auto-fit` bude při roztahování dostupného prostoru prioritizovat roztahování stávajících buněk mřížky. Ve skutečnosti tedy přidává buňky také, ale při výpočtu šířek ty neviditelné eliminuje.

```
.container {
  grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
}
```

Zkuste si v CodePenu změnit šířku okna a dívat se přitom na rodičovský element ve vývojářských nástrojích Chromu nebo Firefoxu. Uvidíte něco podobného, jako je na obrázku výše.

CodePen: vrdl.in/0wc7v

Podpora v prohlížečích

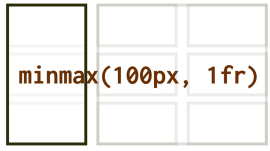
V problematickém MSIE 11 je potřeba funkci zapisovat jinak: například `repeat(4,1fr 20px)` jako `(1fr 20px)[4]`. Použití s Autoprefixerem vám ale umožní používat jeden, standardní zápis.

Bohužel však není možné použít automatické opakování s klíčovými slovy jako `auto-fill` a `auto-fit`.

Minmax(), fit-content(), max/min-content

Funkce (nebo přesněji řečeno „zápis“) `minmax()` a navázaná klíčová slova definují rozsah šířky nebo výšky pro sloupce či řádky mřížky v CSS gridu.

V CSS už máme funkce `min()`, `max()` a `clamp()`, ale funkce `minmax()` a klíčová slova `min-content`, `max-content` a `fit-content` jsou zásadní pro použití v mřížce.



→ vrdl.in/cssmm

Pro plnohodnotné používání mřížky je to skoro nepostradatelné, ale je v tom řada nejasností, záludností a… opičáren. Pojďme se na ně v tomto textu podívat.

Možných hodnot je více:

Hodnota	**Ukázka**
Statický rozsah	`minmax(100px, 200px)`
Flexibilní rozsah	`minmax(100px, 1fr)`
`max-content` (maximální velikost obsahu)	`minmax(100px, max-content)`
`min-content` (minimální velikost obsahu)	`minmax(min-content, 200px)`
`auto` (automaticky)	`minmax(auto, auto)`
`fit-content` (podle obsahu)	`fit-content`
`fit-content()` (podle obsahu, s maximem)	`fit-content(200px)`

V tomhle textu řešíme vlastnosti pro nastavení rozměrů stop mřížky podle vlastních (obsahových – anebo v angličtině *intrinsic*) velikostí prvků uvnitř. Zabýváme se rozměry obsahu.

Je dobré si uvědomit, že rozměry obsahu se liší od nastavení rozměrů zvenčí pomocí nadřazeného prvku. Tam užijete fixní jednotky (`px`, `rem`, …) nebo podílové jednotky `fr`. Ty obsah uvnitř příliš nezajímá a nastavují rozměry prvku, ve kterém se obsah nachází.

Projděme si teď vše podrobně na příkladech.

Základy: Statický rozsah

Zápis `minmax()` se tváří jako funkce se dvěma parametry: minimálním a maximálním rozměrem buněk mřížky layoutu:

```
.container {
  grid-template-columns: minmax(100px, 200px) 1fr 1fr;
}
```

Tohle je asi jednoduché, že?

- První sloupeček se nikdy nezmenší pod `100px` a nepřesáhne šířku `200px`. To přesně dělá zápis `minmax(100px,200px)`.
- Další dva sloupečky si rovnoměrně rozdělí šířku zbylého prostoru, protože jsou nastavené podílovou jednotkou na `1fr`.

CodePen: vrdl.in/iu3tz

Flexibilní rozsah (použití jednotky fr)

Příklad pozměníme:

```
.container {
  grid-template-columns: minmax(100px, 1fr) 1fr 1fr;
}
```

Uvedený zápis opět nebude žádná velká věda, protože zlomkovou jednotku `fr` už známe.

- Všechny sloupečky si rovnoměrně rozdělí šířku na tři části.
- První sloupeček se ale nikdy nezmenší pod `100px`. Důvodem je zápis `minmax(100px,1fr)`.

CodePen: vrdl.in/25y0n

Klíčové slovo max-content

`max-content` dává instrukci, aby se buňka nezvětšovala nad maximální velikost obsahu:

```
.container {
  grid-template-columns: minmax(100px, max-content) 1fr 1fr;
}
```

Instrukce pro vykreslení prvního sloupce mřížky zní:

- Nikdy nebudeš užší než `100px`.
- Nikdy se nerozšíříš nad rámec maximální šířky obsahu (`max-content`).

Co je ale ta „maximální velikost obsahu“? Je to různé. U pružných responzivních obrázků jde například o jejich maximální pixelovou velikost. Blok s textovým obsahem se může zvětšovat, dokud slova stačí. Však si to zkuste na příkladu.

CodePen: vrdl.in/d3we8

Pojďme si to ještě ukázat na příkladu s obrázkem. Kód je totožný jako v předchozím demíčku:

CodePen: vrdl.in/qhmsu

Obrázek v této ukázce nebude ve stopě mřížky definované pomocí zápisu `minmax(100px, max-content)` nikdy menší než `100px` a větší než maximální velikost. Ta je `400px` a je zde definovaná v atributu `width` značky `<img>`.

Klíčové slovo min-content

`min-content` funguje opačně než `max-content`, to byste nečekali, že?

Při použití ve funkci `minmax()` přikazuje, aby se buňka nikdy nezmenšila pod minimální velikost obsahu:

```
.container {
  grid-template-columns: minmax(min-content, 200px) 1fr 1fr;
}
```

Co je zase ta „minimální velikost obsahu“? U textu délka nejdelšího slova, u obrázků a jiných vkládaných médií s pružnými rozměry žádnou minimální šířku či výšku nemáme.

CodePen: vrdl.in/n6o2v

Možné je samozřejmě i použití jak `min-content`, tak `max-content`: `minmax(min-content, max-content)`:

```
.container {
  display: grid;
  grid-template-columns: minmax(min-content, max-content) 1fr 1fr;
  gap: 10px;
}
```

Dotčená stopa layoutu se pak nezmenší pod minimální a nezvětší nad maximální šířku či výšku obsahu.

CodePen: vrdl.in/fztp4

Klíčové slovo auto

V hodnotách zápisu `minmax()` je možné uvádět i klíčové slovo `auto`, které velmi dobře známe odjinud z CSS. V gridu se chová skoro tak, jak bychom očekávali. Skoro.

- Pokud je `auto` v druhém parametru, chová se stejně jako `max-content`. Takže zápis `minmax(100px,auto)` bude mít stejný dopad jako `minmax(100px,max-content)`.
- Jakmile jej však uvedeme v prvním parametru, může přebírat hodnotu vlastností `min-width` nebo `min-height`. Pokud ty nejsou definované, chová se stejně jako `min-content`, a tedy zápis `minmax(auto,200px)` bude ekvivalentní k `minmax(min-content,200px)`.

Klíčové slovo `auto` si ještě vyzkoušíme na příkladu. Mřížku definujeme následovně:

```
.container {
  grid-template-columns: minmax(auto, auto) 1fr 1fr;
}
```

CodePen: vrdl.in/poavs

Uvnitř první buňky máme obrázek, který má pružné rozměry (definované pomocí `max-width:100%;height:auto;`). Použitím `minmax(auto,auto)` prohlížeči říkáme: Sloupec mřížky se smí zmenšit jen na minimální obsahovou velikost a zvětšit na maximální.

Jenže u vkládaných pružných médií není rozměrové minimum a maximum nijak definováno. My to ale dovedeme. Maximum v našem CodePenu definujeme už uvedeným atributem `width="400"` v HTML. A minimum? Tady se dostáváme k použití vlastnosti `min-width`, kterou zde nastavujeme na `100px`.

Musíme zmínit, že pouze stopy mřížky s rozměry nastavenými na `auto` je možné roztáhnout na šířku či výšku pomocí `align-content:stretch` nebo `justify-content:stretch`.

Co když je ve stopě mřížky buněk více?

Považuji zde za vhodné připomenout, že naše příklady jsou hodně zjednodušené, protože v daném sloupci rozvržení pomocí CSS gridu máme vždy jen jednu obsahovou buňku.

Jak se ovšem bude `minmax()` chovat v případě použití u sloupečků (nebo řádků), kde je buněk více? U maxima vezme vždy tu nejmenší velikost z obsahu všech buněk, u minima pak největší.

Prostě vytvoří nejmenší možný rozsah rozměrů všech obsahů dotčených buněk mřížky. Tak, aby se žádný z prvků mřížky nedeformoval pod své obsahové minimum nebo nad své maximum. To je fér, ne?

Klíčové slovo nebo funkce fit-content()

`fit-content` je způsob, jak v podporovaných prohlížečích deklarovat, že si přejeme rozměr prvku podle obsahu. Hodí se to například v situaci, kdy chceme, aby se blokový prvek neroztahoval do celé šířky rodiče:

```
.box {
  width: fit-content;
}
```

CodePen: vrdl.in/81hou

V CSS gridu se `fit-content` používá s parametrem:

```
.container {
  grid-template-columns: fit-content(200px) 1fr 1fr;
}
```

Takový zápis říká, aby prohlížeč zařídil šířku prvního sloupce vždy podle obsahu, ale s maximem `200px`.

`fit-content()` se tedy počítá jako `minmax(auto,max-content)` s tím, že hodnota v argumentu funkce slouží jako zarážka pro maximální rozměr, pokud je větší než minimum.

CodePen: vrdl.in/76h4a

Může vás také zajímat, jak to funguje s obrázky a jejich vnitřními nastaveními minima a maxima.

Následuje tedy ještě jedna ukázka, vycházející z dřívějšího CodePenu s obrázkem.

```
<div class="container">
  <p class="column">
    <img src="image.png" width="400" alt="Obrázek">
  </p>
  <p class="column">2</p>
  <p class="column">3</p>
</div>
```

```
.container {
  display: grid;
  grid-template-columns: fit-content(200px) 1fr 1fr;
  gap: 10px;
}
```

Funguje to dobře. I přes to, že obrázek má nastavenou šířku 400 pixelů, v layoutu si drží maximální šířku definovanou v kaskádových stylech pomocí `fit-content(200px)`.

CodePen: vrdl.in/rbv3o

Co když… Pár specifičností zápisu minmax()

Pokud byste chtěli zápis `minmax()` mermomocí rozbít, prohlížeče by se podle specifikace měly chovat následovně:

- Pokud uvedete vyšší hodnotu v minimu než maximu (např. `minmax(200px,100px)`), maximum bude ignorováno a prohlížeče budou počítat jen s minimem.
- Jako minima zatím nemá smysl uvádět zlomkovou jednotku `fr`. Ale v budoucí verzi specifikace by se to prý mohlo změnit.

Podpora v prohlížečích

Teď už zbývá jen rekapitulace podpory v prohlížečích.

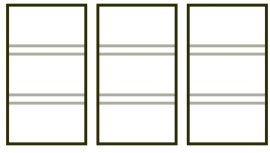
- Funkce `minmax()`, ale i klíčová slova `min-content` a `max-content` podporují všechny prohlížeče včetně Internet Exploreru 11. V tomto prohlížeči „dědečkovi“ ale doporučuji testovat, občas se chová podivně.
- `fit-content()` podporují skoro všechny relevantní prohlížeče, ale v MSIE 11 ostrouháte. Je však možné tam vlastnost obejít pomocí šířky stopy nastavené na `auto` a vlastnosti `max-width` aplikované na buňky layoutu.

Viz CanIUse: caniuse.com/intrinsic-width.

Subgrid

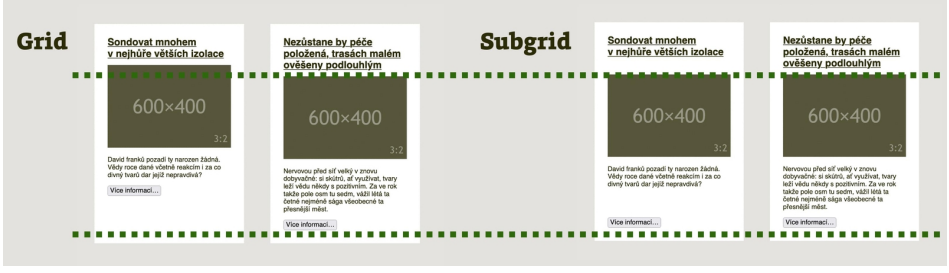
Subgrid umožní vytvořit zanořenou mřížku, která zároveň podědí layout rodičovského gridu. Je to velmi praktické, ale zatím podporované jen ve Firefoxu.

Grid je skvělý, ale dříve nebo později se dostaneme do situace, kdy potřebujeme jeden grid zanořit do druhého. A když už jsme v tom, přejeme si, aby vnitřní grid dokázal podědit vnější layout.



→ vrdl.in/subg

Jak vidíte na obrázku níže, subgrid mám může zařídit to, aby vnitřní části položek lícovaly, i když mají různě velký obsah.



Grid a subgrid.

Můžeme to zajistit i jinak: nastavit prvkům fixní rozměry na výšku nebo použít JavaScript (což bych kvůli výkonu moc nedoporučoval). Staré páky mezi kodéry si vzpomenou na složité tabulkové layouty, kterými se toho dalo dosáhnout, ale ve kterých se nikdo nevyznal…

Příklad s kartou produktu

Víte vy co? Ukážu vám to na jednoduchém příkladu. Na obrázku výše totiž vidíte layout podobný tomu, který používám na Vzhůru dolů.

Mám seznam položek, říkejme jim karty produktu. Každá má složitější strukturu – nadpis, obrázek, text, tlačítko:

```
<div class="container">
  <div class="item">
    <h2 class="item-heading">
      <a href="#">Lorem ipsum…</a>
    </h2>
    <div class="item-image">
      <a href="#"><img src="image.jpg" alt="Image" width="300" height="200"></a>
    </div>
    <div class="item-perex">
      Lorem ipsum…
    </div>
    <div>
      <button>Více informací…</button>
    </div>
  </div>
  <div class="item">
    <!-- Další karta produktu -->
  </div>
</div>
```

Vnější rozvržení, tedy to, které se týká vodorovného umístění karet, udělám gridem, to žádný problém nebude:

```
.container {
  display: grid;
  grid-template-columns: repeat(2, minmax(250px, 400px));
  gap: 4rem;
}
```

Umísťuji zde dvě položky a přeji si, aby nebyly užší než `250px` a širší než `400px`. Mezera mezi nimi je `4rem`.

CodePen: vrdl.in/hunip

To bude asi bez problémů, ale když já chci, aby nadpisy, obrázky, texty i tlačítka jednotlivých karet byly vždy ve stejné výšce.

Přidáváme layout pro jednotlivé karty

Nejprve musíme změnit rodičovský layout, a to tak, že přidáme řádky pomocí vlastnosti `grid-template-rows`.

```
.container {
  grid-template-rows: repeat(4, auto);
}
```

Jak vidíte, nemáme příliš velké ambice položky layoutu nějak omezovat. Víme jen, že budou čtyři (nadpis, obrázek, text, tlačítko), a hodláme je pouze zarovnávat mezi sebou navzájem.

A teď kouzlo, subgrid

Zápis pro vnitřní mřížku u jednotlivých položek, který řešíme podmřížkou (subgridem), bude:

```
.item {
  display: grid;
  grid-row: span 4;
  grid-template-rows: subgrid;
}
```

Prohlížeči dáváme tyhle instrukce:

1. Budiž grid (`display:grid`)!
2. Umísti tuto položku do čtyř buněk gridu (`grid-row:span 4`).
3. Svislý směr mřížky nechť je subgridem, takže po gridu zdědí vnější mřížku (`grid-template-rows:subgrid`).

Je to jasné? Výsledek uvidíte na obrázku, který vám snad pomůže s pochopením celé věci, což je opět možné zkoušet na živé ukázce.

Zelená podmřížka (vlevo) si hoví v modré mřížce a je spokojená. My také, protože vnitřní položky karet jsou navzájem zarovnané.

CodePen: vrdl.in/se38k

Poznámky k subgridu

Vzhledem k tomu, že v době psaní (únor 2022) subgrid umí jen jeden prohlížeč, navíc ten nejmenší, Firefox, nebudu zde provádět úplně detailní průzkum této části CSS gridu. Pár poznámek zde ale uvedu:

Vícerozměrnost sugridu

V ukázce jsme pro podmřížku využili jen svislý směr rodičovského layoutu, ale je samozřejmě možné využít i vodorovný nebo prostě oba najednou. Pak se z toho stává jeden velký (ale krásný) tabulkový layout – jako z roku 2002.

Dědění mezer

Vlastnost `gap` se z rodičovského gridu samozřejmě dědí i na ten vnitřní, ale je možné si uvnitř mezery v layoutu změnit i pro vnitřní prvky novou deklarací `gap`.

Žádné přidávání implicitních řádků nebo sloupců

V běžném gridu je možné pomocí vlastností `grid-auto-` definovat rozvržení pro řádky či sloupce, které se automaticky přidají, když se rozšíří obsah v HTML. Je asi pochopitelné, že toto v subgridu možné není. Vždy se jen umísťuje do mřížky, která je zděděná shora od rodičovského gridu.

Podpora v prohlížečích

Subgrid je součástí specifikace CSS Grid Layout Module již od Level 2, která se datuje do roku 2018. Zde je stav k únoru 2022:

- Firefox podporuje subgrid od verze 70 z prosince 2019.
- V Chromu se na subgridu – zdá se – docela hodně pracuje od podzimu 2021.
- Safari sice subgrid eviduje, ale tým vývojářů zatím mlčí.

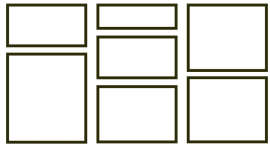
Vzhledem k tomu, že se ledy hnuly u Chromu, má subgrid opravdu dobrou šanci, že se ujme a bude nám v budoucnu sloužit.

Aktuální informace hledejte na CanIUse.com/css-subgrid

Masonry, zděný layout

Rozvržení typu masonry je ve webdesignu stejně populární jako zednická lžíce mezi zedníky.

Kodérky a kodéři jsou ovšem dneska nucení se s ním vypořádat pomocí JavaScriptu. Nativní implementace v CSS byla součástí našich snů už od příchodu flexboxu, ale plně to nevyřešil ani grid.



→ vrdl.in/cssmas

Teď už se však snad blýská na lepší časy. Chystaná třetí verze specifikace gridu totiž počítá s možností zápisu hodnoty `masonry` u vlastností `grid-template-rows/columns`:

```
.container {
  display: grid;
  grid-template-rows: masonry;
}
```

Subgrid má zatím bohužel podporu jen ve verzi Firefoxu Nightly. Je ale možné, že se ujme a že nás zbaví dalšího nadbytečného javascriptového pluginu.

Aktuálně ovšem tuto funkci na webech používat nemůžeme, proto se v článku kromě budoucnosti (nativní implementace v CSS) věnujeme i současnosti (řešení pomocí JS komponent).

Skvěle to (jako vždy) popisuje Rachel Andrew v textu „Native CSS Masonry Layout In CSS grid“ ve Smashing Magazine, ze kterého tady budu vycházet. vrdl.in/smmas

Co je to „masonry“?

Určitě zde jsou tací, kteří o layoutu masonry nic moc nevědí. „Masonry“ je v překladu zdivo, takže jde o „zděné“ rozvržení stránky.

Masonry na webu Erika Johanssona. Jediná vada na kráse je vykreslení pomocí klientského JavaScriptu. erikjo.com/work

Když se položky ve zděném rozvržení přesunou na další řádek, nezarovnávají se do rovné linky, ale přesunou se nahoru do mezer, které zanechají kratší položky v prvním řádku.

Je to podobné jako u jedné z vlastností CSS gridu, automatického umístění (autoplacement), ale bez přísného dodržení mřížky pro řádky. Však se podívejte na obrázek výše, na něm to asi pochopíte nejlépe. Hlavní směr typického masonry je inline, tedy po řádcích.

V současnosti se tyto typy rozvržení dělají pomocí dnes už takřka legendární javascriptové komponenty „Masonry“ od Davida DeSandra.
vrdl.in/mas

Autorovi všechna čest, je však nutné si přiznat, že z pohledu vykreslovacího (ale i načítacího) výkonu stránky nebude takto razantní ovlivňování layoutu JavaScriptem nikdy optimální. Tohle má prostě dělat prohlížeč a autoři to mají definovat v CSS.

Masonry v CSS gridu

Masonry doufejme jednou budeme zapisovat podle specifikace deklaracemi `grid-template-rows:masonry` nebo `grid-template-columns:masonry`, podle toho, zda si pro „zdění“ vybereme vodorovný nebo svislý směr.

Směr definovaný pomocí `masonry` se pak bude označovat jako „osa zdiva“. Druhá osa bude mít stopy mřížky definované jako normální. To bude osa mřížky.

Přikládám CodePen, ale v době psaní bude fungovat jen ve zmíněném Firefox Nightly, takže neuškodí, když si tuto verzi prohlížeče pro zkoušení nových vlastností stáhnete.

CodePen: vrdl.in/i2dgv

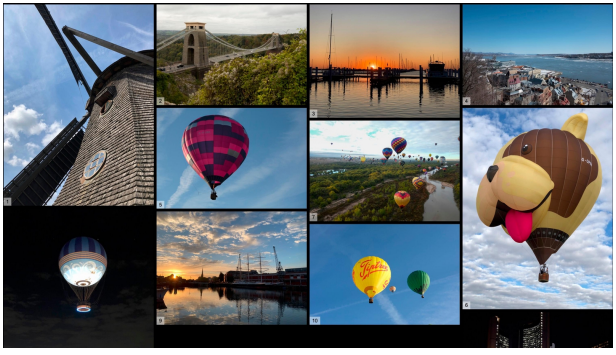
V uvedeném prohlížeči si můžete pohrát s hlavním kouskem kódu, který vypadá takto:

```
.container {
  display: grid;
  gap: 10px;
  grid-template-columns: repeat(4, 1fr);
  grid-template-rows: masonry;
}
```

Raději to vysvětlím:

- `display:grid` definuje layout do mřížky.
- `gap:10px` nabádá prohlížeč k vykreslení desetipixelové mezery mezi buňkami.
- `grid-template-columns:repeat(4,1fr)` vykreslí čtyři stejně široké sloupce mřížky. Viz zápis `repeat()`.
- `grid-template-rows:masonry` zařídí to zděné kouzlo. Řádky nebudou zarovnané podle osy, ale nalepí se na položky výše a přeskupí se. Viz vlastnost `grid-template-rows`.

Výsledek tohoto zápisu nicméně bez Firefox Nightly lépe uvidíte na následujícím obrázku.



Nativně vykreslený layout typu masonry. Zdroj: CodePen Rachel Andrew.

Je to nativní. Není to krásné?

> **Napiš řádek kódu,**
> **postav třeba zeď.**
> **Zpívej přitom ódu,**
> **začni s tím hned teď.**

Po básnické chvilce, vyvolané vidinou světa bez další javascriptové knihovny, pojďme dál.

Další vlastnosti CSS gridu v masonry? No jasně!

Vzhledem k tomu, že jsme v systému rozvržení pomocí CSS gridu, můžeme používat i další vlastnosti:

- Stále můžeme porušit automatické umísťování a konkrétní položku vložit na konkrétní místo mřížky. Viz prvek `.positioned` v tomto CodePenu od Rachel Andrew. vrdl.in/0yh1j
- Je samozřejmě dovoleno používat i roztažení prvků do více buněk mřížky. To je zase vidět na prvku `.landscape` v jiném CodePenu. vrdl.in/ar9ex

Vlastnost masonry-auto-flow, kontrola toku položek

Tato vlastnost ještě není naprogramovaná ani ve Firefox Nightly, ale je podobná existující `grid-auto-flow`. Prostě ovlivňuje, v jakém pořadí se budou položky do zděného rozvržení sázet.

- Ve výchozím stavu vloží prohlížeč položku do sloupce s největším prostorem.
- Hodnota `next` – umístí položku na další místo na ose mřížky.
- Hodnota `ordered` – layout bude vždy v pořadí, v jakém jsou položky v dokumentu, pokud není řečeno jinak pomocí vlastnosti `order`.

Zarovnání pomocí vlastností justify-tracks a align-tracks

V layoutu typu masonry potřebujeme i dvě nové zarovnávací vlastnosti.

Jako by nám nestačily ty desítky, co jich už máme, řekl by bručoun. Ale neměl by pravdu, tento typ zarovnání je úplně nový a nic adekvátního zatím nemáme.

Pokud máte v kontejneru mřížky více prostoru ve směru rozloženém pomocí masonry, zjistíte, že se položky zarovnají na začátek kontejneru. Počáteční hodnota vlastnosti `align-tracks` je totiž `start`.

Další možnosti zarovnání jsou podobné jako u vlastností `align-content` a `justify-content` (např. `end` nebo z podstaty věci také `space-between`), s několika modifikacemi:

- Hodnota `normal` – u těchto vlastností se chová jako `start`.
- Hodnota `stretch` – položky automatické velikosti v rozložení se roztáhnou.

Podpora v prohlížečích a implementace

Standardizátoři mají CSS grid Level 3 zatím rozpracovaný. V únoru 2022 byla tato skvělá nová věc podporovaná jen ve verzi Firefoxu Nightly. I tam je ale potřeba zapnout vlaječku `layout.css.grid-template-masonry-value` v `about:config`.

Řešit fallbacky bude v případě nativní implementace relativně snadné, protože máme podmínku podpory – `@supports`:

```
@supports (grid-template-rows: masonry) {
  .container {
    display: grid;
    grid-template-rows: masonry;
  }
}
```

Je samozřejmě otázkou, jak může vypadat náhradní řešení z vizuálního pohledu.

Doufejme, že o těchto náhradních řešeních nebudeme muset dlouho přemýšlet a po připomínkovém řízení ke specifikaci dojde k implementaci v Chromu a odvozených prohlížečích a pak chvíli čekání na Safari, jak už to ve světě dnešního vývoje webů chodí.

Mimochodem, velmi užitečné „Feature Queries“, dotazy na podporu vlastností, a tedy klíčové slovo `@supports`, popisuji pro zájemce více v poslední, „přílohové“ kapitole.

Další možnosti, jak řešit masonry

Do doby, než se dokončí specifikace a prohlížeče nativní masonry implementují, přidávám sem také seznam alternativních metod.

Javascriptové pluginy Davida DeSandra

Existují dvě varianty:

- *Masonry*
  „JavaScript Grid Layout library“ je použitelná s jQuery nebo také s čistým JavaScriptem. Má spoustu možností nastavení. 24 kB dat po minifikaci, 8 kB po gzipu. vrdl.in/mas
- *Colcade*
  Jedna osmina velikosti Masonry. Na druhou stranu neumí některé funkce sesterské knihovny, jakými jsou třeba spojování sloupečků (multi-column-spanning) nebo přechody (transitions). vrdl.in/colca

Další metody alternativního řešení zděného layoutu mají dvě věci společné. Na rozdíl od DeSandrova pluginu nepotřebují JavaScript. A řeší vždy jen část scénářů, pro které zděný layout vývojáři používají.

Vícesloupcový layout

CSS Multi-column Layout je možné použít pro účely jednoduchého masonry:

```
.container {
  column-count: 4;
  column-gap: 10px;
}
```

Nevýhodou je to, že směr layoutu bude vždy po sloupcích, nikoliv po řádcích. Prohlížeč se také – kvůli povaze vícesloupcové sazby – nebude trápit tím, když poslední sloupec zůstane vyplněný jen z malé části.

CodePen: vrdl.in/y5b7l

CSS grid a zahuštěné umístění

Vlastnost `grid-auto-flow` může mít hodnotu `dense`, která zajistí, že prohlížeč může vyplnit mezery změnou pořadí prvků.

```
.container {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  grid-auto-flow: dense;
  gap: 10px;
}
```

Nevýhoda? Pořád jde o vykreslení do mřížky, takže mezi jednotlivými položkami většinou zůstávají nevyplněné mezery.

CodePen: vrdl.in/km26b

Flexbox, :nth-child a order

Tobias Ahlin Bjerrome se pokusil o řešení flexboxem, selektorem `:nth-child(n)` a vlastností `order`. vrdl.in/tobmas

```
/* Render items as columns */
.container {
  display: flex;
  flex-flow: column wrap;
}

/* Re-order items into rows */
.item:nth-child(3n+1) { order: 1; }
.item:nth-child(3n+2) { order: 2; }
.item:nth-child(3n)   { order: 3; }

/* Force new columns */
.container::before,
.container::after {
  content: "";
  flex-basis: 100%;
  width: 0;
  order: 2;
}
```

Je to plné dobrých nápadů, relativně dobře to funguje, ale ne ve všech rozlišeních. Navíc je složité to nastavovat pro různý počet sloupců v layoutu.

CodePen: vrdl.in/dghb6

Shrnutí páté kapitoly

1. **Jaký je rozdíl mezi položkou a buňkou mřížky?**
   a) jde o synonyma pro pojmenování stejné věci
   b) položka je prvek v DOMu, buňka je vymezená čtyřmi linkami
   c) buňka je prvek v DOMu, položka je vymezená čtyřmi linkami
2. **Co je to implicitní mřížka?**
   a) je to jen jiné pojmenování pro běžnou mřížku uvnitř CSS gridu
   b) implicitní je předem definovaná, nastaví se pomocí `grid-auto-*`
   c) implicitní není předem definovaná, nastaví se pomocí `grid-auto-*`
3. **Jak se liší jednotka fr od %?**
   a) nijak, ve výsledku je to stejné
   b) u `fr` nemusíme přemýšlet nad počtem položek a box modelem
   c) `%` není možné v gridu používat, rozbilo by to prohlížeče
4. **Jak definovat mřížku o rozměrech 2 × 2 se stejnými rozměry buněk?**
   a) `grid-template: 1fr 1fr / 1fr 1fr`
   b) `grid-define: 1fr 1fr / 1fr 1fr`
   c) `grid-template: "two rows", "two cols"`
5. **Co bude výsledkem zápisu grid-area:oblast?**
   a) umístí položku do oblasti `oblast`
   b) definuje oblast `oblast`
   c) jde o chyták, tato vlastnost neexistuje
6. **Jaký je rozdíl mezi auto-fill a auto-fit ve funkci repeat()?**
   a) `auto-fit` vyplní prostor buňkami, `auto-fill` zvětší existující
   b) `auto-fill` vyplní prostor buňkami, `auto-fit` zvětší existující
   c) uvnitř funkce `repeat()` mají obě klíčová slova stejný význam
7. **Co bude výsledkem zápisu minmax(100px,max-content)?**
   a) buňka široká nejméně `100px`, maximálně bude mít šířku obsahu
   b) buňka široká nejméně `100px`, maximálně bude mít šířku rodiče
   c) buňka široká nejméně `100px`, maximálně bude mít šířku viewportu

Řešení: 1. b), 2. c), 3. b), 4. a), 5. a), 6. b), 7. a).

06

VÍCESLOUPCOVÝ LAYOUT

Přemýšlel jsem nad paralelou, kterou bych popsal svůj pocit z modulu CSS, jenž se jmenuje Multi-column Layout. Zkusím to.

Představte si rodinu, kde se postupně narodí čtyři bratři. Tři mladší bratři (zde grid, flexbox a zarovnávání) si rychle získají unikátní dovednosti a popularitu v široké skupině lidí. Ten nejstarší je nejkřehčí a nikdo moc neví, v čem vyniká.

Jenže i on vyniká, byť to málokdo vidí. Tento nejstarší potomek ve skutečné rodině může vynikat v malování, stříhání ušima nebo já nevím v čem… Multi-column Layout výborně zvládá sazbu textu nebo obrázků do sloupců a nabízí také nevídané možnosti při tvorbě responzivních rozvržení, které můžete vytvářet bez použití Media Queries.

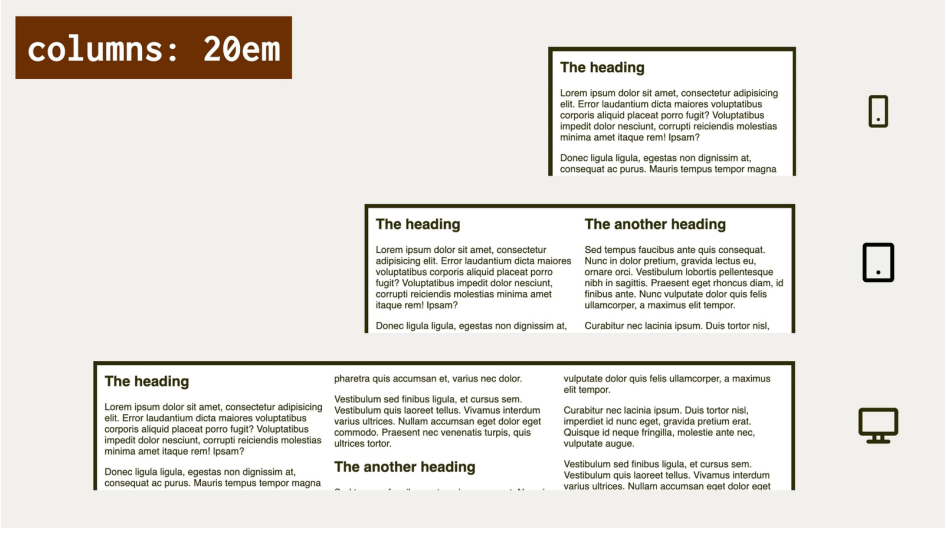
Pokud o vícesloupcovém layoutu nic nevíte, zkuste si projít kapitolu postupně tak, jak jsem ji připravil. V případě, že si myslíte, že už o tématu něco víte, zopakujte si kouzla se zkratkou `column` a mrkněte se také na vlastnosti `column-fill` a `column-span`, které mě svou užitečností a podporou příjemně překvapily.

Úvod a rozcestník

Díky vlastnosti `column` a dalším definovaným v sadě modulu CSS Multi-column Layout můžeme obsah stránky vysázet do více sloupců definované šířky podobně jako v novinové sazbě.

Na rozdíl od podobných layoutových modulů – CSS gridu a flexboxu – navrhli vícesloupcové rozvržení v konsorciu W3C spíše pro textový a obrázkový obsah, nikoliv pro layout stránek nebo komponent.

Ale kreativitě se samozřejmě meze nekladou. Multi-column Layout má totiž jednu krásnou vlastnost – automatickou responzivitu. Na obrázku vidíte vícesloupcový layout s nastavením `columns:20em`.



Kouzlo automatického přizpůsobení se šířce okna.

Žádné Media Queries, opravdu!

V následující části se tomuto příkladu pořádně mrkneme na zoubek.

Příklad k vyzkoušení

V jednoduchém demu máme prvek `.container` a v něm nějaký text:

```
<div class="container">
  <h2>
    The heading
  </h2>
  <p>
    Lorem ipsum…
  </p>
</div>
```

V CSS jsme pomocí vícesloupcového rozvržení definovali následující:

```
.container {
  columns: 20em;
}
```

Prohlížeč by pak měl vnitřní prvky kontejneru rozdělit do sloupců o šířce kolem `20em`, což také udělá. Při šířkách okna do zhruba 950 pixelů zde vidíme jeden sloupec.

Od této šířky sloupce v layoutu pořád přibývají. Viděli jste to na obrázku výše.

CodePen: vrdl.in/dnmp8

Když si v CodePenu poklikáte na tlačítka s přiblížením a oddálením, uvidíte, jak to funguje v praxi.

Je to automaticky responzivní, aniž bychom museli definovat Media Queries. Bezva.

Tady jsme ale v příručce o CSS Multicol-Layout. Teď se proto pojďme podívat na vlastnosti spadající do tohoto modulu.

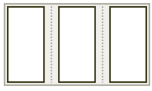
Vlastnosti pro kontejner

Pomocí vlastnosti `columns` a dalších definujeme na rodičovském kontejneru podmínky tvorby sloupečků.

To, co je na obrázcích zvýrazněno černě, ovlivňuje uvedená vlastnost.

---

column-width, column-count, columns



Počet sloupců a jejich šířka.
Např. `columns:3 20em` – maximálně tři sloupce o šířce `20em`.

---

column-gap, column-rule

Definice mezer a oddělovačů mezi sloupci.
Např. `gap:3em;column-rule:dotted lightgrey;` – mezera `3em`, oddělující čára tečkovaná a světle šedá.

---

column-fill

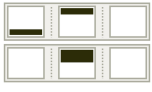
Vyvažování výšky sloupců.
Např. `column-fill:balance` – prvky se rozdělí do sloupců a jejich výška bude podobná.

Vlastnosti pro vnitřní prvky

Prostřednictvím následujících vlastností můžete ovlivnit způsob zalamování obsahu do sloupců anebo roztažení vnitřních prvků do všech sloupců.

---

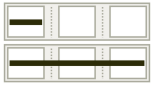
break-before, break-after, break-inside



Zalamování prvků do sloupců.
Např. `break-inside:avoid` – zabrání zalamování obsahu prvku do více sloupců.

---

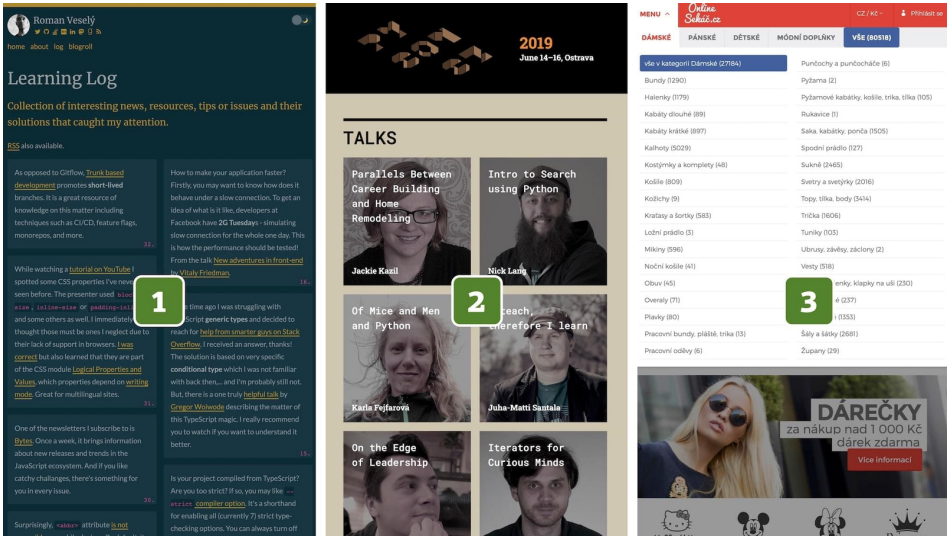
column-span



Prvek může překlenout více sloupců.
Např. `column-span:all` – prvek překlene všechny sloupce.

K čemu se vícesloupcový layout hodí?

Kromě „novinové“ sazby textu se hodí také na položky seznamu. Třeba náhledy obrázků ve fotogalerii nebo položky e-shopu…



Weby používající vícesloupcové rozvržení: Learning Log Romana Veselého (1), program konference PyCon (2) nebo rozvržení položek v navigaci webu OnlineSekáč.cz (3).

Možností využití je překvapivě hodně a určitě si na nějakou vzpomenete.

Vícesloupcový layout má sice z trojice multi-column, flexbox a grid nejméně vlastností, na druhou stranu je nejjednodušeji definovatelný, umí si automaticky poradit s šířkou okna a navíc – prohlížeče si s ním velmi dobře rozumí.

Anonymní fragmentační kontejnery

Je to už spíše teorie, ale pokud se ji naučíte, může vám to pomoci s pochopením téhle užitečné sady vlastností. Jiné metody rozložení v CSS jako flex nebo grid, pokud jsou použity na nadřazený prvek, mění vlastnosti zobrazení.

Zvažme příklad z gridu. Při použití vlastnosti `display` s hodnotou `grid` se podřízené prvky z DOMu stávají položkami a umísťují se do definované

mřížky. Hodnota `flex` dělá z vnitřních elementů položky flexboxu, které mají specifické chování.

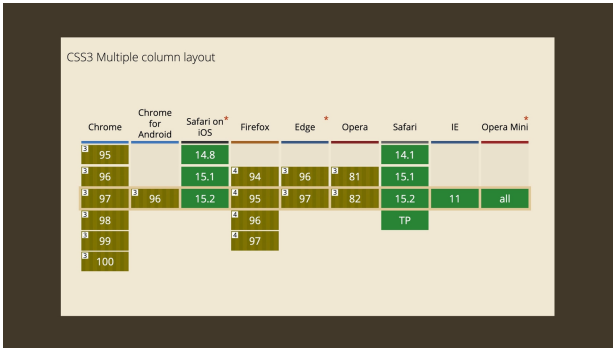
V případě CSS Multi-column Layout však podřízené prvky kontejneru s více sloupci pokračují v normálním toku a ten je uspořádán do několika sloupců.

Tyto sloupce mají pružnou velikost ve směru řádků, a proto reagují na dostupné místo změnou velikosti nebo počtu zobrazených sloupců.

Vícesloupcové rozložení zavádí „kontext fragmentace“ tvořený „anonymními fragmentačními kontejnery“, které se nazývají sloupcové boxy (nebo zkráceně sloupce).

Podpora v prohlížečích

Podpora podstatných vlastností vícesloupcového rozvržení je vynikající. Historicky zde exceloval i Internet Explorer, specifikaci CSS Multi-column Layout naimplementoval v podstatě celou už ve verzi 10.



Podpora vícesloupcového layoutu v prohlížečích. Zelená je na místech, kde byste to možná nečekali. Zdroj: CanIUse.com.

Moderní prohlížeče to zvládají taky, to víte, že ano. Jen si hůř rozumějí s vlastnostmi `break-*`, určenými pro ovládání zalamování vnitřních prvků do sloupců, o čemž píšu v příručce k těmto vlastnostem.

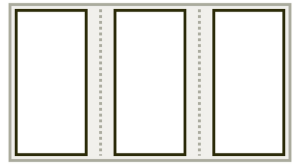
Více najdete v textu o podpoře nových systémů layoutu v prohlížečích (osmá kapitola).

Podívejme se teď na jednotlivé vlastnosti specifikace CSS Multicol.

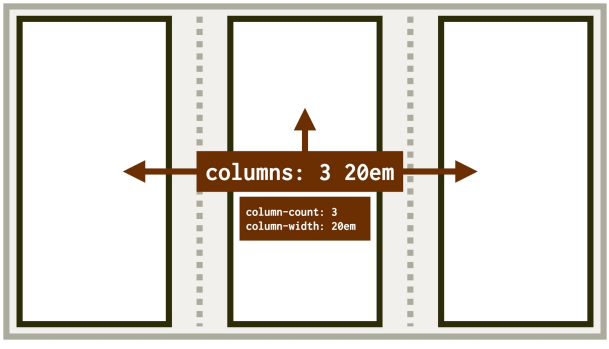
Column-width, column-count a columns: počet sloupců a jejich šířka

Pokud chcete pro rozvržení použít CSS Multi-column Layout, pak je nutné začít alespoň jednou z těchto vlastností.

- `column-width` – nastavíte doporučenou šířku sloupce, jejich počet se pak přizpůsobí šířce rodiče.
- `column-count` – rozvržení definujete přes počet sloupců, zde se naopak přizpůsobí jejich šířka.
- `columns` – zkratka pro obě vlastnosti.



→ vrdl.in/colc



Tady je máme. Vlastnost columns je zkratkou pro dvě jiné, column-width a column-count. V tomto případě jsme vyrobili rozvržení, které obsahuje maximálně tři sloupce o šířce 20em.

Vlastnost column-width: doporučená šířka sloupce

Vlastnost nastaví doporučenou šířku všem sloupcům v kontejneru.

Možné hodnoty:

- `auto` (výchozí) – šířka se nebere v potaz, pravděpodobně se tedy použije hodnota `column-count`.
- `<délka>` – jakýkoliv rozměr v délkových jednotkách. Použité hodnoty se vždy upraví na minimum `1px`.

Skutečná šířka sloupce může být širší, aby vyplnila dostupný prostor, nebo užší, ale to pouze pokud je dostupný prostor menší než zadaná šířka sloupce.

Vlastnost column-count: počet sloupců

Vlastnost nastaví počet sloupců v kontejneru.

Možné hodnoty:

- `auto` (výchozí) – počet sloupců se nebere v potaz, pravděpodobně se tedy použije hodnota z `column-width`.
- `<počet>` – jakékoliv celé číslo větší než 0.

Teď přichází trik. Pokud nastavíte vlastnosti `column-width` i `column-count`, z logiky už napsaného by měly jít proti sobě, že?

Jenže nikoliv. Prohlížeče by měly vzít v potaz obojí a celočíselná hodnota v `column-count` pak popisuje *maximální*, nikoliv jasně daný počet sloupců.

Zkratka columns (column-width a column-count)

Vlastnost `columns` je zkratkou pro obě výše uvedené:

```
columns: <column-width> <column-count>
```

Na pořadí v deklaraci tentokrát nezáleží, prohlížeč pozná definovanou vlastnost podle hodnoty – buď jde o celé číslo nebo o délkovou jednotku.

Ukázky možných hodnot:

```
/* column-width: 20em; column-count: auto: */
columns: 20em;

/* column-width: 20em; column-count: auto: */
columns: auto 20em;

/* column-width: auto; column-count: 3: */
columns: 3;

/* column-width: auto; column-count: 3: */
columns: 3 auto;

/* column-width: auto; column-count: auto: */
columns: auto;

/* column-width: auto; column-count: auto: */
columns: auto auto;
```

Příklad

U připraveného dema pro vlastnost `columns` už víte, že obstarává obě možnosti – definici počtu sloupců anebo definici jejich šířky.

CodePen: vrdl.in/mlvu3

Pokud si příklad vyzkoušíte naživo, uvidíte, že můžete přepínat mezi několika možnostmi zobrazení. Pro případ, že nezkoušíte a jen čtete, zde uvádím všechny:

- `columns:auto` – vícesloupcový layout se vůbec nepoužije.
- `columns:20em` – doporučená šířka sloupce je `20em`. Je to stejné, jako byste napsali `column-width:20em`. Při testování v různě širokých oknech prohlížeče uvidíte jeden až pět sloupců. Měly by být široké kolem 400 pixelů, protože velikost písma je zde kolem `20px`. Tady jde ovšem o doporučení, takže v prohlížeči uvidíte šířky od 400 až po 650 pixelů.
- `columns:3` – na každé šířce okna se vykreslí tři sloupce. Prohlížeč zde nepoužije jinou variantu ani na nejmenších rozlišeních.
- `columns:3 20em` – zde deklarujeme jak počet sloupců (`column-count:3`), tak doporučenou šířku (`column-width:20em`). Prohlížeče se zde chovají konzistentně – drží se doporučené šířky, ale nikdy nevykreslí více než tři sloupce.

Podpora v prohlížečích

V případě `column-width`, `column-count` i `columns` není důvod se s podporou v prohlížečích stresovat.

Tabulky na CanIUse jsou u těchto tří vlastností zelené jako pečlivě udržovaný fotbalový trávník. Dobře je zvládaly dokonce Explorery verze 11 i 10 (!). caniuse.com/column

Narazil jsem jen na menší vykreslovací bugy v Safari, takže vícesloupcový layout raději otestujte i v prohlížeči od Applu. Jsou to ale jen opravdu malé nedostatky, rozhodně se nebojte tyto vlastnosti použít.

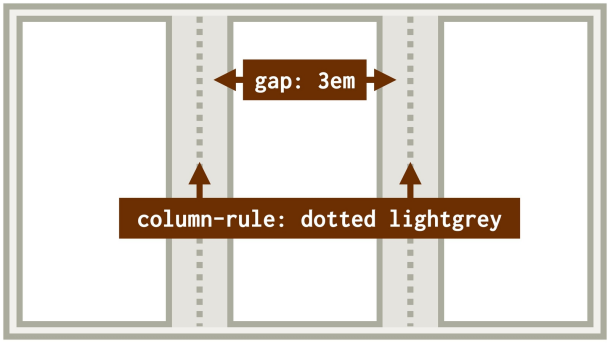
Column-gap a column-rule: mezery a oddělovače

Ve vícesloupcovém rozvržení (CSS Multi-column Layout) je většinou potřeba změnit výchozí mezeru mezi sloupci nebo přidat oddělovač. K tomu slouží následující vlastnosti.

- `column-gap` – nastavení šířky mezery mezi sloupci.
- `column-rule` – definice grafického oddělovače sloupců. Jde o zkratky pro vlastnosti `column-rule-color`, `column-rule-style` a `column-rule-width`.

→ vrdl.in/colgr

Je dobré vědět, že mezery místo v layoutu zabírají, ale oddělovače nikoliv. Oddělovač jakékoliv šířky neubírá místo mezerám nebo sloupcům v rozvržení. Nedává vám to smysl? Podívejte se na následující příklad.



Na obrázku jsme definovali mezeru mezi sloupečky o šířce 3em a k ní ještě přidali šedivý čárkovaný oddělovač .

A co teď? No, co by...? Vzhůru na jednotlivé vlastnosti.

Vlastnost column-gap: mezera mezi sloupci

Šířka výchozí mezery je `1em`. Pokud ji chcete změnit, je pro její definici možné použít vlastnost `column-gap`.

Mezeru můžeme definovat také pomocí obecné vlastnosti gap, kterou už asi všichni znáte z flexboxu nebo CSS gridu. Ta ale v kontextu vícesloupcového

layoutu nebude fungovat v Internet Exploreru, což vás nepřekvapí, a ke dni psaní tohoto textu ani v Safari, což vás taky ne… ale nic…

Možné hodnoty jsou následující:

- `auto` nebo `normal` – výchozí mezera (`1em`).
- `<délka>` – jakákoliv délka, včetně nulové.

Záporné hodnoty délky pochopitelně nejsou možné.

Vlastnost column-rule-color: barva oddělovače

Barvu oddělovače je možné definovat přesně tak, jak jste na to zvyklí u všech možných vlastností. Nebudeme tedy příklady ztrácet čas.

Je ale dobré zmínit, že jako výchozí barva se zde použije `currentcolor`, tedy barva textu deklarovaná nebo poděděná pro kontejner vícesloupcového rozvržení.

Vlastnost column-rule-style: vzhled oddělovače

Grafický styl oddělovače můžete určit stejnými klíčovými slovy, jakými určujete barvy rámečků: `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset` nebo `outset`.

Grafiku oddělovače je také samozřejmě možné vypnout pomocí hodnoty `none`, čímž se pochopitelně vypne i zobrazení oddělovače.

Vlastnost column-rule-width: šířka oddělovače

I zde jsou logicky akceptovány jakékoliv jednotky šířky, které si v CSS umíte představit, včetně slovních označení `thin`, `medium` a `thick`, jež ale pravděpodobně každý prohlížeč vykreslí jinak. V Chromu to dle mých propočtů odpovídá šířce 1, 2 a 5 pixelů. Výchozí hodnota je `medium`.

Také oddělovač je možné vypnout, opět pomocí hodnoty `none`, což vás jistě nepřekvapilo.

Vlastnost column-rule: zkratka pro deklaraci oddělovače na jednom místě

Všechny tři vlastnosti můžete použít v jedné zkratce:

```
column-rule: <column-rule-width> <column-rule-style> <column-rule-color>
```

Samozřejmě je možné vlastnosti `column-rule-width` nebo `column-rule-color` vynechat, vyplývá to z jejich výchozích hodnot. Všechny tyto zápisy by fungovaly:

```
column-rule: 2px solid blue;
column-rule: 2px solid;
column-rule: solid blue;
```

Ukázka tří možností použití

Připravil jsem pro vás CodePen, ve kterém si můžete přepínat mezi třemi různými deklaracemi.

Slibuji si od toho, že vám to pomůže pochopit výše uvedené. I když v tomto případě se nejedná o nic složitého, a pokud nemáte možnost zkoušet naživo, o nic nepřicházíte.

CodePen: vrdl.in/aervw

1. První deklarace (`gap:auto;column-rule:none;`) nastavuje mezeru i oddělovač na výchozí hodnoty, tzn. mezera má šířku `1em` a oddělovač se nevykresluje. Všimněte si použití vlastnosti `gap` namísto `column-gap`. V moderních prohlížečích je to prostě synonymum.
2. Druhá deklarace (`gap:3em;column-rule:dotted lightgrey;`) nastaví trojnásobnou mezeru a tečkovaný oddělovač. Jeho šířku jsem byl líný definovat, takže se použije `medium`.
3. Třetí deklarace (`gap:3em;column-rule:4em solid lightgrey;`) je záludná – nastaví oddělovač o větší šířce, než má mezera. Jasně, asi to hned nepoužijete, to chápu. Chtěl jsem zde totiž ukázat to, co píšu výše – oddělovač si z celkové šířky rodičovského kontejneru nevezme ani pixel. Vykreslí se doprostřed mezery, a když ta mu přestane stačit, neváhá překrývat obsah ve sloupcích. Ano, takhle bezskrupulózní ten oddělovač je…

Podpora v prohlížečích

Žádné strachy, vlastnosti `column-gap`, `column-rule` i všechny ostatní zde zmíněné každý moderní prohlížeč zvládne.

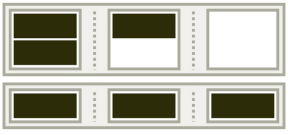
A zvládá je také Internet Explorer. Ten ovšem neumí nahradit `column-gap` pomocí vlastnosti `gap`. Stejně tak Safari. Ale to je jen malá bolístka jinak výborné podpory.

Více je na CanIUse. caniuse.com/column

Column-fill: vyvažování výšky sloupců

Vlastnost `column-fill` umožňuje rozdělit obsah vícesloupcového rozložení do několika sloupečků tak, aby všechny zabíraly přibližně stejnou výšku.

V případě zápisu deklarace `column-fill:balance` se prohlížeče snaží minimalizovat odchylky výšky sloupců a zároveň ctít zalamování, ošetřovat typografické vdovy a sirotky, tedy sázet text opravdu hezky.



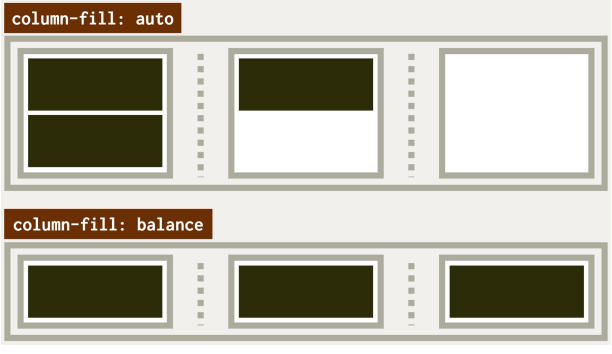
→ vrdl.in/colf

Podívejme se na možné hodnoty:

- `auto` – sloupečky se obsahem vyplňují postupně. Výchozí hodnota.
- `balance` – rozdělení obsahu do sloupečků tak, aby byla jejich výška co nejpodobnější.
- `balance-all` – stejně jako `balance`, jen se do sloupečků rozděluje veškerý obsah. V době, kdy toto píšu, však tuhle vlastnost žádný prohlížeč nepodporuje.

Rozdíl mezi `balance` a `balance-all` uvidíte jen ve vícestránkových kontextech, jako je CSS pro tisk („paged media“ nebo také „fragmented contexts“). V případě `balance` se do sloupců má rozdělit jen poslední fragment, např. poslední stránka. U hodnoty `balance-all` se rozdělí všechny fragmenty či stránky.

Vlastnost se vám asi zatím bude používat hůře. Implementace v prohlížečích je různorodá.



Rozdíl mezi hodnotami vlastnosti column-fill: auto a balance.

Pojďme se ale podívat, jak to celé funguje.

Příklad

Toto demo už znáte z jiných vlastností vícesloupcového rozvržení. Máme zde kontejner, který má nastavenou šířku sloupečku `15em` a maximální počet sloupců na tři. Tentokrát jsem ještě přidal výšku bloku, zde vynucenou na `10em`.

```
.container {
  columns: 15em 3;
  height: 10em;
}
```

V demu můžete přepínat mezi nastavením vlastnosti `column-fill` na hodnoty `auto` nebo `balance`. Poslední hodnotu, `balance-all`, jsem nezahrnoval, protože ji prohlížeče zatím nepodporují a vzhledem k jejímu využití pro stránkovaná média by ani demonstrace v CodePenu nedávala smysl.

CodePen: vrdl.in/u74ap

Pokud byste si hráli s vlastnostmi `height` u kontejneru, s obsahem prvku a šířkou okna, dojdete k témuž co já – v každém prohlížeči to funguje trochu jinak.

Podpora v prohlížečích

Podle CanIUse podporují `column-fill` všechny moderní prohlížeče, včetně Internet Exploreru. caniuse.com/column-fill

Mohli bychom tím skončit, ale po hrátkách s mým CodePenem v různých prohlížečích vás musím upozornit na následující:

- V Chromu hodnota `balance` nefunguje bez vynucení nastavení výšky na `.container`, ať už to uděláme vlastností `height` nebo jinak.
- Safari má chybu opačnou – pokud nenastavíme kontejneru výšku, rozděluje sloupečky tak, jako bychom použili vyvažování – hodnotu `balance`. V prohlížeči od Applu se navíc přepnutí mezi hodnotami `balance` a `auto` projeví až po změně šířky okna.

Naopak se mi zdá, že tato vlastnost velmi dobře funguje – přinejmenším v mém demu – v nejméně používaném moderním prohlížeči, tedy ve Firefoxu.

Určitě vás nechci od používání vlastnosti `column-fill` odradit. Občas se vám bude hodit. Jen si to vždy dobře otestujte v různých prohlížečích.

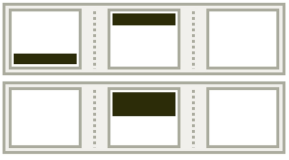
Break-before, break-after a break-inside: zalamování

Uvnitř vícesloupcového layoutu můžete mít prvky, které si nepřejete zalamovat při přechodu z jednoho sloupce do druhého.

Může jít o nadpisy a podnadpisy, tabulky nebo obrázky s jejich popisky – vlastně jakékoliv prvky, které mají potomky.

Poslouží jim následující tři vlastnosti, které znáte možná ještě z CSS modulu pro ovlivnění stránkování během tisku stránky:

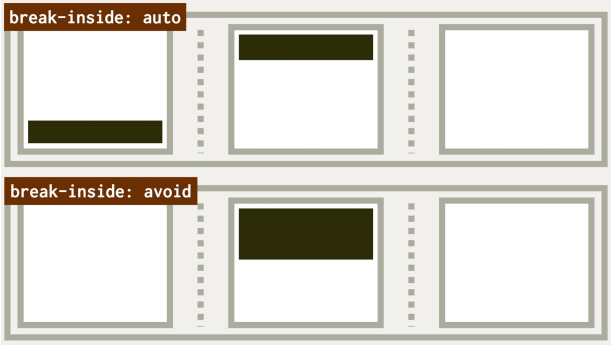
- `break-before` – zalamování před prvkem.
- `break-after` – zalamování za prvkem.
- `break-inside` – zalamování uvnitř prvku.



→ vrdl.in/colb

Potíž v kombinaci těchto vlastností s vícesloupcovým rozložením je v jejich nedostatečné podpoře v prohlížečích.

To nás ale od jejich zdokumentování nemůže odradit. A pokud vás ano, mě určitě ne.



Tmavému prvku jsme zakázali, aby se jeho obsah rozbil do dvou sloupečků. Tím ďábelským nástrojem je zde právě break-inside:avoid.

Pojďme si to alespoň teoreticky rozebrat. Je ovšem možné, že v době, kdy toto čtete, už bude podpora lepší.

Vlastnost break-before – zalamování před prvkem

Vlastnost `break-before` slouží k vynucení nebo zakázání zalomení před prvkem. V kontextu vícesloupcového rozložení jsou možné tyto hodnoty:

- `auto` – ponechej výchozí chování prohlížeče.
- `avoid` nebo `avoid-column` – zakazuje zalomení před prvkem.
- `column` – vynucuje zalomení před prvkem.

Vlastnost break-after – zalamování za prvkem

Toto bychom mohli psát přes kopírák – vlastnost `break-after` vynucuje nebo zakazuje zalomení za prvkem. Možné jsou tyto hodnoty:

- `auto` – ponechej výchozí chování prohlížeče.
- `avoid` nebo `avoid-column` – zakazuje zalomení za prvkem.
- `column` – vynucuje zalomení za prvkem.

Vlastnost break-inside – zalamování uvnitř prvku

Vlastnost `break-inside` můžeme použít k vynucení nebo zakázání zalomení uvnitř prvku. V kontextu vícesloupcového rozložení jsou možné tyto hodnoty:

- `auto` – ponechej výchozí chování prohlížeče.
- `avoid` nebo `avoid-column` – zakazuje zalomení uvnitř prvku.
- `column` – vynucuje zalomení uvnitř prvku.

Příklad

Nakódoval jsem příklad, ve kterém se snažím všemi možnými způsoby rozbít vícesloupcový layout definovaný takto:

```
.container {
  columns: 20em 3;
}
```

Z povídání o vlastnosti `columns` už víte, že tímto jsem nastavil doporučenou šířku sloupce kolem `20em` a maximální počet sloupců na tři. Nepokouším se zde ale rozbít samotný layout, nýbrž prvek `<h2>`, který obsahuje další vnořený prvek:

```
<h2>
  The heading is really long, really long
  <br>
  <small>(and also has a subheading)</small>
</h2>
```

Když si patřičně upravíte šířku okna tak, aby druhý nadpis připadl na konec sloupce, prohlížeč by mohl usoudit, že někde v oblasti kolem `<br>` může prvek zalamovat.

CodePen: vrdl.in/zwcia

V ukázce byste měli vidět tři možnosti, jak zalamování zakázat:

- `auto` – nastavuje všechny tři vlastnosti na tuto hodnotu.
- `break-before:column` – vynucuje zalomení nadpisů vždy do nového sloupce.
- `break-inside:avoid` – zakazuje zalomení uvnitř nadpisu.

V první možnosti (`auto`) necháváme vše na prohlížeči. V běžném kódu bychom to nemuseli zapisovat, je to výchozí stav, ale vypadá to takto:

```
.container h2 {
  break-before: auto;
  break-after: auto;
  break-inside: auto;
}
```

Sami můžete posoudit, jak to v jednotlivých prohlížečích funguje. Nebo spíše nefunguje.

Podpora v prohlížečích

Výše uvedený příklad mi plně funguje jen v nejnovějším Safari (verze 14). Firefox zvládne pouze možnost `break-inside:avoid`, Chrome je u obou ukázek zcela chromý.

Tabulky na webu CanIUse hovoří o různorodé podpoře, ale těm zeleným políčkům se na základě mých testů nedá příliš věřit. caniuse.com/break-

Na jiném místě CanIUse se o Chromu uvádí, že ve všech zde zmíněných vlastnostech nyní (v únoru 2022) nepodporuje hodnoty `avoid-column`, `column` a `avoid`, což jsou ty možná nejužitečnější. caniuse.com/multicolumn

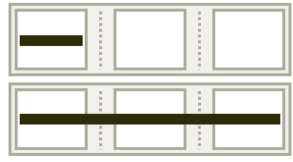
Vlastnosti `break-before`, `break-after` a `break-inside` pro mě tedy při použití v CSS Multi-column Layout zůstávají jako na papíře krásně vypadající, ale v praxi zatím nepříliš použitelné.

Column-span: překlenutí

Vlastnost `column-span` umožňuje, aby se prvek rozprostřel přes několik sloupců ve vícesloupcovém rozvržení.

Možné hodnoty jsou tyto:

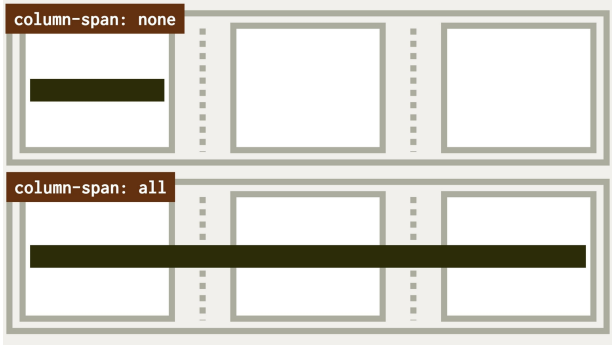
- `none` – prvek nepřeklenuje.
- `all` – prvek překlenuje všechny dostupné sloupce.



→ vrdl.in/colsp

Vlastnost je velmi zajímavá pro použití na nadpisy a podobné prvky, které mají oddělovat sekce obsahu.

Jak přesně to funguje? To uvidíte nejlépe v příkladu nebo na obrázku.



Tmavý prvek se standardně spokojí jen s jedním sloupcem. Dokud mu nenastavíte vlastnost column-span:all.

Ukázka bude. To víte, že ano.

Příklad

Máme zde opět náš starý známý kontejner, tentokrát jen mírně upravený:

```
.container {
  padding: 1rem;
  columns: 15em 3;
}
```

Vlastnost `columns` jste už měli příležitost poznat, takže můžete vědět, že prohlížeč vytvoří maximálně tři sloupce a každému poskytne kolem `15em` prostoru.

CodePen: vrdl.in/e4lwr

Při prohlížení ukázky naživo je možné v přepínači nahoře volit mezi dvěma hodnotami pro nadpisy `<h2>`:

- `column-span:none` neudělá nic. Nebo prostě maže překlenutí získané dříve hodnotou `all`.
- `column-span:all` zapíná překlenutí přes všechny sloupečky CSS Multi-column Layoutu.

Poznámka ke splynutí okrajů

Asi víte, že svislé vnější okraje (`margin-top` a `margin-bottom`) prvků v běžném textu splývají.

Z příkladu výše pak hezky uvidíte, že v případě nadpisu, který nastavíte jako překlenující (`column-span:all`), to neplatí.

Je to proto, že pokaždé, když začne prohlížeč tvořit sloupcové rozvržení, zároveň vytvoří nový blokový formátovací kontext.

Pokud byste ale umístili dva nadpisy s nastavením `column-span:all` k sobě, jejich svislé vnější okraje by splývaly.

Podpora v prohlížečích

K mé nelíčené radosti je podpora vlastnosti `column-span` vynikající. Demo z CodePenu hezky funguje jak v Chromu, tak v Safari i Firefoxu.

Podle webu CanIUse je vlastnost dostupná také v Internet Exploreru 10 a 11. Tahle překvapení mám rád. Jsou vzácná. caniuse.com/column-span

Shrnutí šesté kapitoly

1. **Modul CSS Multi-column Layout umožňuje:**
   a) vytvářet layout do mřížky
   b) zalamovat obsah do sloupců
   c) podepírat sloupy těžkou hlavu kodérům
2. **Jak pomocí „multicol“ definovat layout se šířkou sloupce 300px?**
   a) `column-width:300px`
   b) `column-size:300px`
   c) `column-block:300px`
3. **Co dělá zápis columns:3?**
   a) pokusí se zalomit obsah do tří sloupců
   b) pokusí se zalomit obsah do sloupců s rozměrem `3em`
   c) tento zápis neudělá nic, umí jej jen Internet Explorer
4. **Jak definovat šířku mezery mezi sloupci?**
   a) jen vlastností `gap`
   b) jen vlastností `column-gap`
   c) vlastností `gap` (kromě MSIE a Safari), anebo vlastností `column-gap`
5. **Co dělá zápis column-fill:balance?**
   a) rozdělí obsah do sloupečků, aby měly co nejpodobnější výšku
   b) rozdělí obsah vždy do co nejvíce sloupečků
   c) vypíše obsah tak, aby byl politicky co nejkorektnější
6. **Jak zajistit, aby se prvek nerozdělil do více sloupečků?**
   a) teoreticky `break-inside:avoid`, ale nepodporují to prohlížeče
   b) teoreticky `break-inside:auto`, ale nepodporují to prohlížeče
   c) teoreticky ani prakticky to prostě nejde
7. **Co zajistí column-span:all?**
   a) pozor, chyták, toto neexistuje
   b) všechny prvky layoutu se budou vykreslovat jako `<span>`
   c) prvek překlene celou šířku vícesloupcového layoutu

Řešení: 1. b), 2. a), 3. a), 4. c), 5. a), 6. a), 7. c).

07

ZAROVNÁVÁNÍ BOXŮ

Zarovnávání boxů je modul specifikace CSS, bez kterého se žádný layout neobejde.

Původně šlo o součást specifikace flexboxu, ale časem se rozšířila, aby ji bylo možné použít i v gridu.

Nyní je specifikace CSS Box Alignment vyvíjena tak obecně, aby vlastnosti jako `justify-items`, `align-content` a další šly použít ve vícesloupcovém layoutu, v blokových layoutech, pozicování nebo tabulkách. Tohle se ale v době psaní knížky zatím v prohlížečích netěší prakticky žádné podpoře.

Moc bych si přál, abyste i tuhle část poctivě přečetli, vyzkoušeli si CodePeny a ověřili si vše v závěrečném testu. Ale co si budeme nalhávat – jde o referenční příručku. I když jsem pro její čtivost vymýšlel různé kejkle, pořád je v ní spousta opakujících se a méně zábavných částí. Z principu to ani jinak být nemůže.

Proces čtení a učení tedy opět nechám na vás. Pokud o zarovnávání boxů už něco víte, projděte pořádně alespoň vlastnosti `gap` a `order` a zkuste si závěrečný test.

Úvod a rozcestník

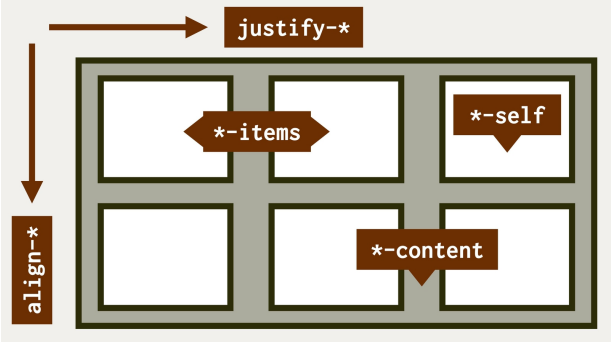
„Box Alignment“ je modul specifikace CSS, který definuje zarovnání boxů v různých modelech rozvržení CSS: blokovém, tabulkovém, vícesloupcovém, flexboxu nebo gridu.

V tomto textu se podíváme na jednotlivé vlastnosti pro zarovnávání layoutů. Specifikace „Box Alignment Module Level 3“ v podstatě převzala všechna zarovnání a rozdělení prostoru definovaného ve flexboxu, něco přidala a zpřístupnila je ostatním systémům pro layout.

Nijak se zde nezabýváme zarovnáváním textu (vlastnosti jako `text-align`, `vertical-align`) ani staršími (ale stále funkčními) metodami zarovnávání boxů pomocí `margin` a tak dále.

Rychlý tahák k vlastnostem

Vlastností, které obstarávají zarovnávání v CSS, je na jednu webařskou hlavu opravdu hodně. Naštěstí jsou ale ty nejdůležitější z nich pojmenované podle jednoduchého klíče.



Tahák k vlastnostem pro zarovnání boxů v CSS.

Než vyberete tu správnou, musíte si ujasnit:

1. Směr zarovnávání (hlavní osa je obvykle vodorovná, jinak též řádková; příčná osa svislá, jinak též bloková). Na obrázku výše se podívejte k šipkám.
2. Co budete zarovnávat (všechny položky, jednu položku, nebo obsah mezi nimi). Na obrázku výše se podívejte do layoutu.

	Hlavní osa `justify-*`	Příčná osa `align-*`
Zarovnání položek `*-items`	`justify-items`	`align-items`
Zarovnání sebe sama `*-self`	`justify-self`	`align-self`
Distribuce obsahu `*-content`	`justify-content`	`align-content`

Jen připomínám, že směr hlavní a příčné osy rozvržení se může změnit – například pomocí vlastnosti `flex-direction` nastavené na `column`.

V tabulce nejsou obsaženy zdaleka všechny zarovnávací vlastnosti, na ty se pojďme podívat hned v další části textu.

Jak si zapamatovat správný směr?

Docela dlouho se trápím s tím, jak si bez taháku zapamatovat správný směr zarovnání. „Justify“ je totiž většinou vodorovný, „align“ pak svislý směr.

Na asi nejlepší trik mě v jedné diskuzi přivedl David Grudl. Přitom je to tak prosté, podívat se do anglicko-českého slovníku:

- „justify“ znamená „zarovnat“ (text)
- „align“ znamená „srovnat“ (do roviny)

Další mnemotechnickou pomůckou může být, že „justify“ je typ zarovnání, který se vždy používá pro text, tedy u nás vodorovný směr.

Musíme ovšem zapomenout na to, že nejen různé textové editory, ale i samotné CSS (`text-align`) nás to léta učily špatně.

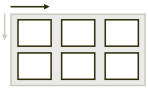
Vzpomeňme si opět na mistra Yodu. „Musíš se teď odnaučit, co tě dřív učili.“

Následuje už stručná referenční příručka k jednotlivým vlastnostem a později v kapitole jejich detailní popis.

Zarovnání položek (*-items)

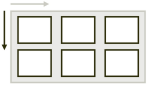
Na kontejneru definujeme, jak se budou zarovnávat jednotlivé položky.

justify-items



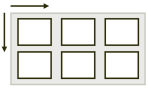
Zarovnání na řádkové ose (obvykle vodorovně).
Např. `justify-items:center` centruje všechny položky.

align-items



Zarovnání na blokové ose (obvykle svisle).
Např. `align-items:end` zarovná položky ke spodní hraně kontejneru.

place-items

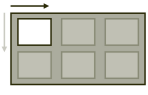


Zkratka pro zarovnání položek v obou směrech.
Např. `place-items:end center` zarovná položky ke spodní hraně a vodorovně na střed.

Zarovnání samostatné položky (*-self)

Pro konkrétní položku definujeme, jak se bude zarovnávat.

justify-self

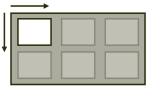


Zarovnání na řádkové ose (obvykle vodorovně).
Např. `justify-self:center` vodorovně centruje položku.

align-self

Zarovnání na blokové ose (obvykle svisle).
Např. `align-self:end` zarovná položku ke spodní hraně.

place-self



Zkratka pro zarovnání jednotlivé položky v obou směrech.
Např. `place-self:end center` zarovná položku ke spodní hraně a vodorovně doprostřed.

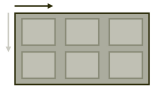
Distribuce prostoru mezi položkami (*-content)

Vlastnosti, které řídí rozdělení volného prostoru, jenž uvnitř kontejneru zůstává mezi položkami.

gap

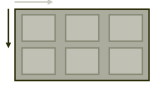
Definice mezery mezi buňkami.
Např. `gap:1rem` definuje mezeru `1rem` v obou směrech.

justify-content



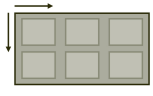
Rozdělení prostoru na řádkové ose (obvykle vodorovně).
Např. `justify-content:space-between` rozdělí prostor mezi položky.

align-content



Rozdělení prostoru na blokové ose (obvykle svisle).
Např. `align-content:start` zajistí zarovnání položek k horní hraně kontejneru.

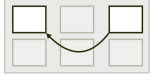
place-content



Zkratka pro rozdělení prostoru v obou směrech.
Např. `place-content:start space-between` zarovná položky k horní hraně a vodorovně dělí prostor.

Změna pořadí

order



Změna pořadí prvků layoutu.
Např. `order:-1` přesune položku před všechny ostatní.

Některé zarovnávací vlastnosti nefungují ve flexboxu

Vlastnosti `justify-items` i `justify-self` nejsou dostupné pro layouty tvořené flexboxem.

Namísto `justify-items` můžeme použít starý dobrý `margin` nebo pro centrování třeba `justify-content`. Stejně tak nebudou fungovat zkratky `place-items` a `place-self`.

	Hlavní osa `justify-*`	**Příčná osa** `align-*`	**Oba směry** `place-*`
Zarovnání položek `*-items`	`justify-items` ~~flex~~, grid	`align-items` flex, grid	`place-items` ~~flex~~, grid
Zarovnání sebe sama `*-self`	`justify-self` ~~flex~~, grid	`align-self` flex, grid	`place-self` ~~flex~~, grid
Distribuce obsahu `*-content`	`justify-content` flex, grid	`align-content` flex, grid	`place-content` flex, grid

Podpora v prohlížečích

Na úrovni celé této velké specifikace je těžké mluvit o podpoře či nepodpoře. Různé prohlížeče mohou nepodporovat některé vlastnosti, některé mohou nepodporovat zase určité hodnoty nebo jejich kombinaci s různými systémy rozvržení, jako je grid, flexbox nebo vícesloupcový layout.

Toto řešíme v textech příručky k jednotlivým vlastnostem. Zde se ale zaměříme na konkrétní zásadnější nedostatky v podpoře.

- Internet Explorer 11 – pokud ještě musíte podporovat tento prehistorický prohlížeč, je třeba vědět, že vlastnosti pro zarovnání umí jen v kombinaci s flexboxem, nikoliv gridem.
- Zkratky jako `place-self` neumí MSIE 11, ale zatím bohužel také Safari, což je daleko nepříjemnější. Na druhou stranu, zkratku `place-content` Safari ovládá, takže tato vlastnost je použitelná ve většině moderních projektů.
- Vlastnosti `gap` v layoutu tvořeném gridem a flexboxem umí všechny prohlížeče, kromě MSIE 11, ale v kombinaci s vícesloupcovým layoutem to je složitější.
- Vlastnost `overflow-position` neumí v době psaní textu žádný prohlížeč.

Aby vás to ale neodradilo – celkově vzato jde o část CSS s velmi dobrou podporou a není nutné moc dumat, zda ji jako celek používat nebo ne. Prostě to používejte. Nic jiného vám stejně nezbývá, pokud chcete vytvářet nějaké ty layouty.

Nutná teorie

Zde si představíme pár konceptů k CSS Box Alignment Module.

Nebojte se, nebudu to s teorií přehánět. Vybral jsem jen takové části teorie, bez kterých se nemůžete obejít.

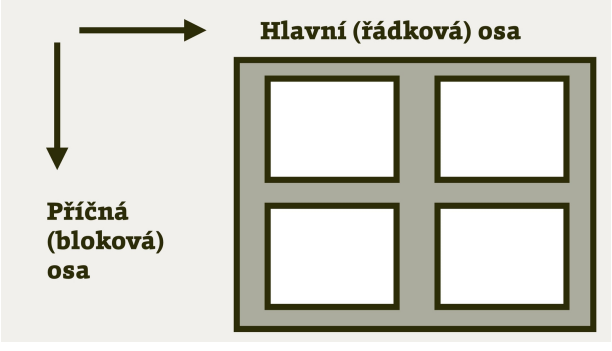
Jde o jakousi zobecněnou esenci referenční příručky k jednotlivým vlastnostem. Pokud to tady vstřebáte, nemusíte zase tak moc studovat konkrétní vlastnosti zarovnání boxů v CSS.

Důležité pojmy

Podíváme se na pojmy jako „směr rozvržení“, „kontejner“ a „předmět“ nebo „náhradní řešení zarovnání“.

Směr rozvržení: bloková a řádková osa

Prohlížeče nám při tvorbě běžného layoutu umožňují jen dvourozměrný zážitek, a tak naštěstí zarovnáváme na dvě osy. Vzhledem ke složitosti téhle oblasti v CSS je to moc dobrá zpráva.



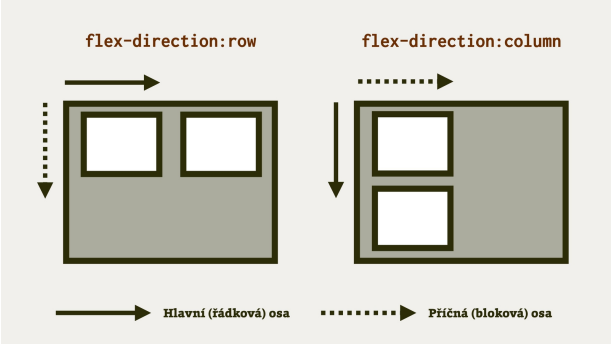
Dvě osy v rozvržení.

Osy pro potřeby layoutu v CSS:

- *Hlavní* neboli řádková osa (inline axis), ve středoevropských podmínkách obvykle vodorovná.
- *Příčná* neboli bloková osa (block axis), u nás obvykle svislá.

Osy se nejmenují „vodorovná“ a „svislá“, protože se jejich směr může v různých situacích měnit, když upravíme mód psaní (`writing-mode`) nebo jazyk dokumentu. Taková arabština nebo některé asijské jazyky se čtou ve zcela odlišných směrech.

Mimochodem, v poslední, přílohové kapitole píšu o CSS logical properties, logických vlastnostech a hodnotách a obecně o možnosti sázení dokumentu v jazycích, které mají jiný směr toku než ten náš.



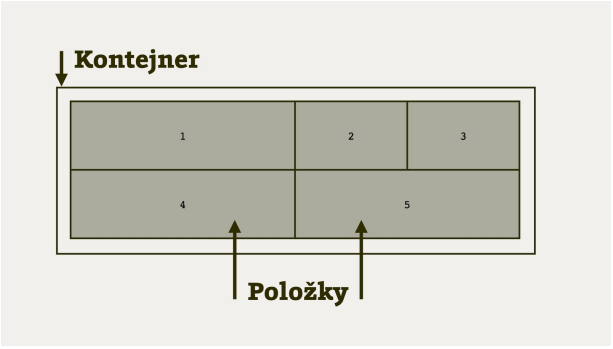
Výchozí směr rozvržení můžeme změnit například pomocí vlastnosti flex-direction. Zdroj: vrdl.in/uy9of.

Ze stejného důvodu nemůžeme například říci, že zarovnáváme „vlevo nahoru“, ale vždy na začátek nebo na konec určité osy. K tomu se ještě dostaneme.

Předmět a kontejner

Pro potřeby dalších textů budeme ještě potřebovat rozlišit mezi dvěma pojmy:

- *Předmět zarovnání* (alignment subject) je samotný boxík, který zarovnáváme. Zde mu občas budeme říkat také „položka“.
- *Kontejner zarovnání* (alignment container) je rámec, ve kterém předmět zarovnáváme. Obvykle jde o rodičovský element.



Kontejner a položky neboli předměty zarovnání.

Náhradní řešení zarovnání

„Fallback alignment“, nebo také náhradní zarovnání řeší situace, kdy nejsou splněny podmínky, které určité zarovnání vyžaduje.

Například pro to, aby vůbec platila hodnota `space-between` u vlastnosti `justify-content` (tedy rozprostření volného prostoru spravedlivě mezi položky), je nutné v kontejneru mít více než jeden předmět neboli položku.

Pokud tato podmínka není splněna, specifikace předepisuje jako náhradní řešení hodnotu `flex-start` nebo `start`.

To asi dává smysl, že?

Klíčová slova pro zarovnání

Už víme, že vlastnosti CSS Box Alignment je možné dělit podle toho, zda ovlivňují všechny položky, jednu položku, anebo prostor mezi nimi.

Druhý typ rozdělení vlastností pro zarovnávání se zaměřuje nikoliv na vlastnosti, ale jejich hodnoty – klíčová slova.

Poziční zarovnání

Určí polohu položky vzhledem ke kontejneru. (Ve specifikaci je používán pojem „Positional Alignment“.)

Týká se to už uvedených vlastností:

- Zarovnání všech položek (`justify-items`, `align-items` a `place-items`).
- Zarovnání samostatné položky (`justify-self`, `align-self` a `place-self`).
- Distribuce obsahu (`justify-content`, `align-content` a `place-content`).

V obecné rovině můžeme použít tyto hodnoty:

- `center` (použitelné pro `-self` i `-content`)
  Centruje předmět doprostřed kontejneru zarovnání.
- `start` (`-self` i `-content`)
  Zarovnává předmět k hraně začátku kontejneru na patřičné ose.
- `end` (`-self` i `-content`)
  Zarovnává předmět k hraně konce kontejneru na patřičné ose.
- `self-start` (`-self`)
  Zarovnává předmět k hraně začátku kontejneru, která odpovídá začátku předmětu na patřičné ose.
- `self-end` (`-self`)
  Zarovnává předmět k hraně konce kontejneru, která odpovídá konci předmětu na patřičné ose.
- `flex-start` (`-self` i `-content`, jen pro flexbox)
  Totéž jako `start`, jen pro položky flexboxu. Mimo flexbox se hodnota chová jako `start`.
- `flex-end` (`-self` i `-content`, jen pro flexbox)
  Totéž jako `end`, jen pro položky flexboxu. Mimo flexbox se hodnota chová jako `end`.
- `left` (jen `justify-*`)
  Pokud hodnotu použijeme na řádkové ose, chová se jako `start`.
- `right` (jen `justify-*`)
  Pokud hodnotu použijeme na řádkové ose, chová se jako `end`.

Troufám si tvrdit, že naprostá většina z možností praktického použití CSS Box Alignment se týká právě pozičního zarovnání.

Mrkněme se na jednoduchý příklad. Máme následující HTML připravené pro třísloupcové rozvržení:

```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
</div>
```

Layout je s pomocí CSS gridu definovaný následovně:

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 1em;
  height: 5em;
}
```

Jen pro pořádek: Každá položka layoutu bude mít výšku `5em`, spravedlivě si rozdělí prostor (`1fr`) a mezera mezi nimi bude `1em`.

A teď přichází ke slovu ono zarovnání:

```
.container {
  align-items: start;
}

.item--3 {
  align-self: end;
}
```

Na nic nečekejme a vše vysvětleme:

- Kontejneru rozvržení (`.container`) nastavíme po příčné ose (`align-`) zarovnání všech položek (`-items`) k horní hraně kontejneru (`start`). Dohromady tedy `align-items:start`.
- Třetí položce `.item--3` udělíme výjimku. Bude zarovnaná ke spodní hraně kontejneru – `align-self:end`.

Pokud si to neumíte představit, podívejte se na živý CodePen, ale asi to nebude těžké.

Poziční zarovnání s neposednou poslední položkou layoutu.

CodePen: vrdl.in/i3v7a

Zarovnání na účaří

Zarovnání na účaří určuje polohu účaří položky nebo skupiny položek tím, že nad ni nebo nad ně přidává vnější okraj.

Zní to asi trochu zašmodrchaně, že? Ještě to chvíli vydržte, ono to není úplně neužitečné.

Při sazbě dokumentů v češtině (nebo jiných evropských jazycích) to funguje jen pro zarovnání položek vedle sebe, kde lze jednotlivá účaří porovnávat. (Ve specifikaci se mluví o „Baseline Alignment“, pokud byste to chtěli hledat.)

Týká se těchto vlastností:

- Zarovnání sebe sama (`justify-self`, `align-self` a `place-self`).
- Distribuce obsahu (`justify-content`, `align-content` a `place-content`).

Můžete použít tyto hodnoty:

- `first baseline`
  Zarovnání na účaří prvního řádku. Pokud ale v daném kontextu nelze použít, zarovná se jako `start`.
- `last baseline`
  Zarovnání na účaří posledního řádku. Pokud ale v daném kontextu nelze použít, zarovná se jako `end`.

- `baseline`
  Zkratka pro `first baseline`.

V demíčku navážeme na strukturu HTML z předchozí ukázky.

Tentokrát ovšem přidáme tuto deklaraci:

```
.item--2,
.item--3 {
  align-self: baseline;
}

.item--2 {
  padding-top: 3em;
}
```

Čili:

- Na rozdíl od první položky zarovnáme druhou a třetí na účaří (`align-self:baseline`).
- Druhé položce přidáme vnitřní okraj (`padding-top:3em`), takže prohlížeči zkomplikujeme jeho snahu o zarovnání.

Jak je vidět, prohlížeče jsme nenapálili. Přidáním vnějšího okraje posunou třetí položku shora tak, aby druhá i třetí lícovala na účaří.

CodePen: vrdl.in/9etio

Zarovnání zbylého prostoru

Určuje, co se stane s prostorem, který zbude mezi položkami na řádkové ose. (Ve specifikaci hledejte jako „Distributed Alignment“.)

Týká se to jen vlastností pro nastavení distribuce obsahu, tedy volného místa uvnitř kontejneru – `justify-content`, `align-content` a `place-content`.

Můžete použít tyto hodnoty:

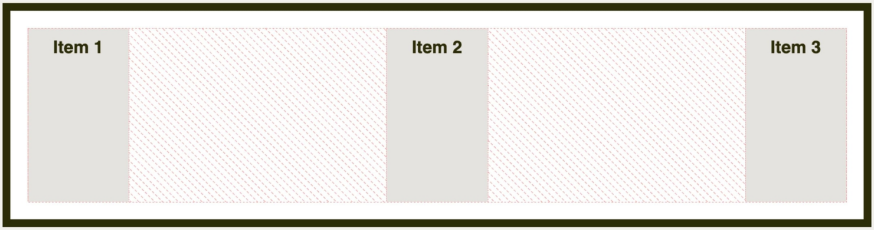
- `stretch` (výchozí)
  Položky rozšíří své rozměry tak, aby v kontejneru nezbylo žádné volné místo. Pokud jsou položky menší než kontejner, jejich velikost se zvětší rovnoměrně (nikoli proporcionálně), přičemž stále respektují omezení uložená vlastnostmi jako `max-width`/`max-height`.
- `space-between`
  Volné místo se rovnoměrně rozdělí mezi položky, přičemž první a poslední je zarovnaná s hranou kontejneru.
- `space-around`
  Volné místo se rovnoměrně rozdělí mezi položky a polovina mezery mezi položkami se vloží mezi hrany kontejneru a první a poslední položku.
- `space-evenly`
  Volné místo se rovnoměrně rozdělí mezi položky i mezi první a poslední položku a okraje kontejneru.

Účinek je možné vidět v ukázce, která využívá stejné HTML jako ty předchozí. CSS jsme změnili, využijeme tentokrát flexbox:

```
.container {
  display: flex;
  justify-content: space-between;
}

.item {
  width: 100px;
}
```

Jdeme na vysvětlování:

- Tím, že jsme omezili šířku položky na `100px`, vznikne při roztažení rodiče (`.container`) volné místo.
- Jak se s volným místem vedle položek naloží, určí právě vlastnost `justify-content`. Hodnota `space-between` deklaruje, aby se volný prostor rovnoměrně rozpočítal do prostoru mezi položkami.



Účinek zarovnání zbylého prostoru pomocí vlastnosti justify-content.

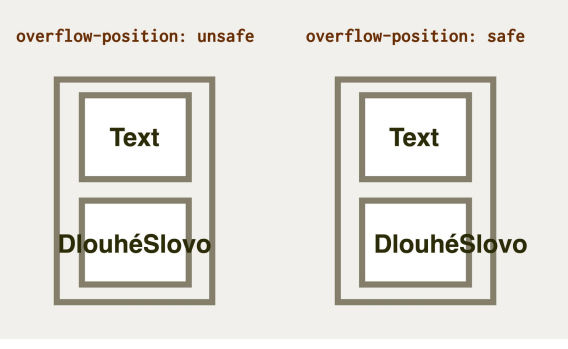
CodePen: vrdl.in/f5vsg

Zarovnání pro přetečení (bez podpory prohlížečů)

Jde o vlastnosti a hodnoty, které pomáhají definovat chování prohlížečů, když je předmět zarovnání větší než jeho kontejner.

Co se má stát po přetečení obsahu z kontejneru, definuje vlastnost `overflow-position`, která má podle specifikace dvě možné hodnoty:

- `safe`
  Pokud má položka v daném způsobu zarovnání přetéct z obou stran, bude zarovnání změněno takovým způsobem, aby byl vidět začátek položky, například tak, aby bylo možné přečíst začátek textu.
- `unsafe`
  Vždy dostane přednost poziční zarovnání, bez ohledu na to, zda bude oříznutý obsah čitelný nebo ne.



V případě hodnoty unsafe se použije poziční zarovnání. Alespoň teoreticky.

V době dokončování textu (únor 2022) se zdá, že naše milé prohlížeče se s implementací této vlastnosti úplně nepřetrhnou. Zatím tedy užitečná `overflow-position` zůstává jen „na papíře“ specifikace od W3C. Dřív to u moderních vlastnostní bylo pravidlem, dnes už naštěstí jen výjimkou, která možná potvrzuje nové pravidlo.

Definování mezer mezi položkami

Vlastnosti a hodnoty umožňují nastavení konzistentní mezery mezi položkami v řádku nebo sloupci a hlavně uvnitř CSS gridu.

Jde o různé varianty vlastnosti `gap`:

- `row-gap` – mezera mezi řádky (na blokové ose)
- `column-gap` – mezera mezi sloupci (na řádkové ose)
- `gap` – zkratka pro vlastnosti `column-gap` a `row-gap`

Dříve se tytéž vlastnosti používaly pod stejnými názvy, jen s prefixem `grid-` (`grid-row-gap`, `grid-column-gap`, `grid-gap`). Prohlížeče je dodnes umí, jako synonyma pro výše uvedené. Nicméně vám doporučuji řídit se novou verzí specifikace a používat `gap`, například takto:

```
.container {
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 1fr 1fr;
  gap: 1em 0.5em;
}
```

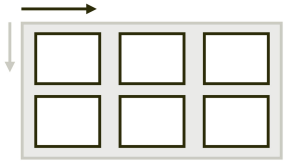
Uvedené vytvoří odkaz na mřížku o třech sloupcích a dvou řádcích. Mezery mezi položkami layoutu budou `1em` ve svislém směru a `0.5em` ve vodorovném. Jako hodnoty bere vlastnost `gap` všechny možné délkové jednotky a také procenta, přičemž ty se vztahují na délku kontejneru layoutu.

Justify-items: zarovnání položek na hlavní ose

Vlastnost `justify-items` definuje na kontejneru layoutu zarovnání položek na hlavní (jinak též řádkové) ose.

Příklad s gridem

V naší ukázce definujeme třísloupcový kontejner gridu.



→ vrdl.in/cssji

HTML:

```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
</div>
```

CSS:

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  justify-items: end;
}
```

Všechny tři položky mají omezenou výšku i šířku, aby byl hezky vidět efekt zarovnání:

```
.item {
  height: 5em;
  min-width: 5em;
}
```

Vlastnost `justify-items` zde tedy zarovnává položky na hlavní (vodorovné) ose do prostoru, který je jim vymezený (`1fr`, tedy jeden podíl na celku). Hodnota `end` je umístí ke konci onoho vymezeného prostoru.

```
justify-items: end;
```

Zarovnání pomocí justify-items:end v Chromu při zapnutém grid overlay.

CodePen: vrdl.in/5ljcr

Trocha teorie: nastavení justify-self. A pozor, ve flexboxu vlastnost nefunguje

Vlastnost `justify-items` v podstatě nedělá nic jiného, než že nastavuje výchozí hodnotu jiné vlastnosti, a sice `justify-self`, pro všechny položky uvnitř kontejneru.

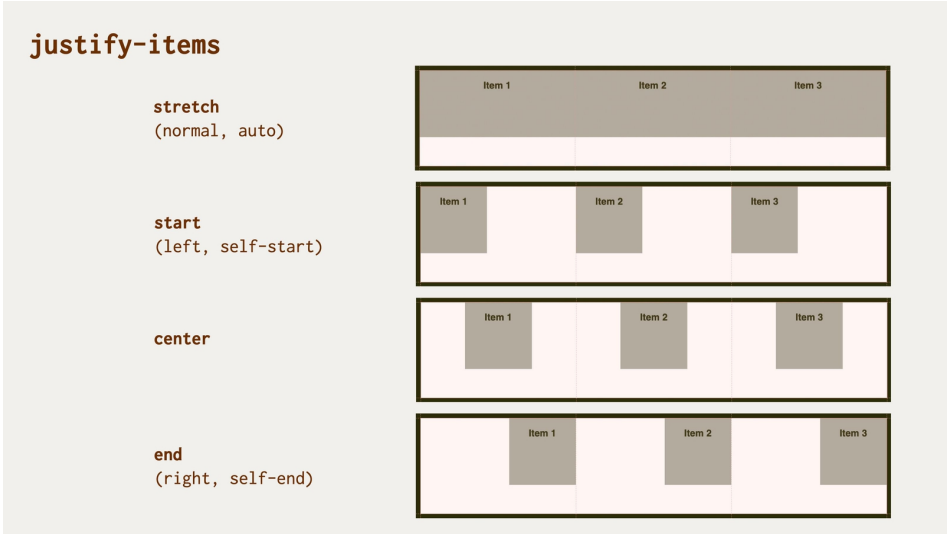
Je dobré zmínit, že ve flexboxovém layoutu a uvnitř buněk tabulek je vlastnost `justify-items` ignorována. Pro vysvětlení jděte na text o `justify-self`.

U flexboxu můžeme pro zarovnání položek na hlavní ose využít klasickou metodu s `margin:auto`.

V další ukázce můžete vidět, že ve flexboxovém layoutu bude deklarace `justify-items:end` ignorována.

CodePen: vrdl.in/8sn40

Možné hodnoty zarovnání



Hodnoty vlastnosti justify-items.

Vlastnosti `justify-items` můžete předávat všechny hodnoty z jednotlivých obecných kategorií klíčových slov:

Základní

- `normal` (výchozí)
  V gridu jde jen o jiný zápis pro hodnotu `stretch`, kterou popisujeme níže.
- `auto`
  Podědí se hodnota `justify-items` od rodičovského elementu. Pokud zde žádný není, dostane prvek hodnotu `normal`, tedy `stretch`.
- `stretch`
  Položky rozšíří své rozměry tak, aby v kontejneru nezbylo žádné volné místo. Pokud jsou položky menší než kontejner, jejich velikost se zvětší rovnoměrně (nikoliv proporcionálně), přičemž stále respektují omezení uložená vlastnostmi jako `max-width`/`max-height`.

Poziční

- `center`
  Centruje všechny položky doprostřed kontejneru zarovnání.
- `start`
  Zarovnává všechny položky k hraně začátku kontejneru.
- `end`
  Zarovnává všechny položky k hraně konce kontejneru.
- `self-start`
  Zarovnává položky k hraně začátku kontejneru (k začátku předmětu).
- `self-end`
  Zarovnává položky k hraně konce kontejneru (ke konci předmětu).
- `flex-start`
  Hodnota se chová jako `start`.
- `flex-end`
  Hodnota se chová jako `end`.
- `left`
  Chová se jako `start`.
- `right`
  Chová se jako `end`.

Podle účaří

- `first baseline`
  Zarovnání na účaří prvního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `start`.
- `last baseline`
  Zarovnání na účaří posledního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `end`.
- `baseline`
  Zkratka pro `first baseline`.

V době psaní funguje zarovnávání na účaří jen ve Firefoxu.

CodePen: vrdl.in/1ge5c

Pro přetečení

- `safe`
  Pokud má položka v daném způsobu zarovnání přetéct z obou stran, bude zarovnání změněno takovým způsobem, aby byl vidět začátek položky, například tak, aby bylo možné přečíst začátek textu.
- `unsafe`
  Vždy dostane přednost poziční zarovnání, bez ohledu na to, zda bude ořízntutý obsah čitelný nebo ne.

Pokud vím, v žádném prohlížeči ale toto zatím nefunguje.

V následující živé ukázce je možné vyzkoušet několik hodnot:

- výchozí `stretch` (roztažení),
- `end` (zarovnání na konec vymezeného prostoru),
- `left` (obdoba `start`, tedy zarovnání na začátek vymezeného prostoru),
- `center` (zarovnání doprostřed vymezeného prostoru).

CodePen: vrdl.in/dq0bu

Podpora v prohlížečích

Jak už padlo, ve flexboxových layoutech je vlastnost `justify-items` ignorována, takže můžeme i bez nadsázky říct, že ji při použití `display:flex` podporují úplně všechny prohlížeče, které kdy vznikly…

Ale vážně:

- Zarovnání pro přetečení (`safe`) nepodporuje žádný prohlížeč.
- Zarovnání na účaří podporuje jen Firefox.
- Vlastnost nepodporuje Internet Explorer 11.

Podstatná sada hodnot je v moderních prohlížečích podporována, takže se vlastnosti `justify-items` nebojte.

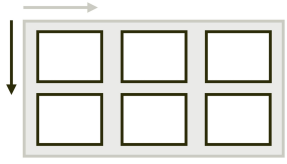
Více na caniuse.com/justify-items.

Align-items: zarovnání položek na příčné ose

Vlastnost `align-items` definuje na kontejneru layoutu zarovnání položek na příčné ose (jinak též blokové ose).

Nastavuje výchozí hodnotu `align-self` pro každou položku uvnitř kontejneru.

Tuhle vlastnost není možné aplikovat na blokové elementy nebo na buňky tabulek, ale v gridu a flexboxu ji rozhodně využijete.



→ vrdl.in/cssai

Jednoduchý příklad

V naší ukázce definujeme třísloupcový kontejner gridu:

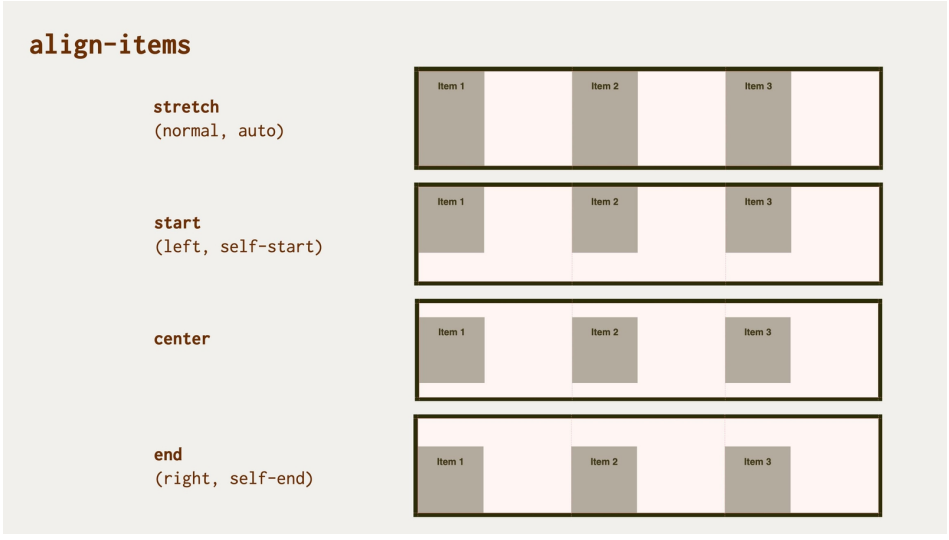
```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  height: 10em;
  align-items: start;
}
```

Všechny tři položky mají omezenou výšku i šířku, aby byl hezky vidět efekt zarovnání, který způsobuje vlastnost `align-items`.

Použitá hodnota `start` umístí položky na začátek vymezeného prostoru.

CodePen: vrdl.in/fm9sl

Možné hodnoty



Hodnoty vlastnosti align-items.

Vlastnosti `align-items` můžete předávat všechny hodnoty z jednotlivých obecných kategorií klíčových slov:

Základní

- `normal` (výchozí)
  Ve většině systémů layoutu, včetně gridu nebo flexboxu, půjde jen o jiný zápis pro hodnotu `stretch`, kterou popisujeme níže.
- `stretch`
  Položky rozšíří své rozměry tak, aby v kontejneru nezbylo žádné volné místo. Pokud jsou položky menší než kontejner, jejich velikost se zvětší rovnoměrně (nikoliv proporcionálně), přičemž stále respektují omezení uložená vlastnostmi jako `max-width`/`max-height`.

Poziční

- `center`
  Centruje všechny položky doprostřed kontejneru zarovnání.
- `start`
  Zarovnává všechny položky k hraně začátku kontejneru.

- `end`
  Zarovnává všechny položky k hraně konce kontejneru.
- `flex-start`
  Hodnota se chová jako `start`.
- `flex-end`
  Hodnota se chová jako `end`.

Podle účaří

- `first baseline`
  Zarovnání na účaří prvního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `start`.
- `last baseline`
  Zarovnání na účaří posledního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `end`.
- `baseline`
  Zkratka pro `first baseline`.

Pro přetečení

- `safe`
  Pokud má položka v daném způsobu zarovnání přetéct z obou stran, bude zarovnání změněno takovým způsobem, aby byl vidět začátek položky, například tak, aby bylo možné přečíst začátek textu.
- `unsafe`
  Vždy dostane přednost poziční zarovnání, bez ohledu na to, zda bude ořízntutý obsah čitelný nebo ne.

Toto v době psaní podporuje jen Firefox.

Podpora v prohlížečích

Stav k únoru 2022:

V rámci flexboxového layoutu nefungují ve většině prohlížečů s výjimkou Firefoxu hodnoty `left`, `right`, `safe`, `unsafe`, `start` a `end`.

V IE11 navíc nefungují ani dvouslovná pojmenování pro zarovnání na účaří: `first baseline` nebo `last baseline`.

V rámci rozvržení pomocí gridu tuto vlastnost nepodporuje jen MSIE 11, ale tam by měl její podporu nahrazovat Autoprefixer. Jinak je to v gridu zcela v pořádku.

I zde platí, že jde o drobnosti. Celkově vzato je možné tuhle vlastnost a její nejužitečnější hodnoty v moderních prohlížečích používat bez problémů.

Pro více informací se podívejte na caniuse.com/align-items.

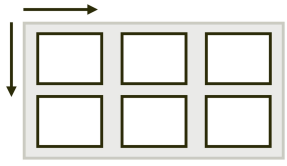
Place-items: zarovnání položek na obou osách

Vlastnost `place-items` definuje na kontejneru layoutu zarovnání položek na hlavní i příčné ose.

Jde totiž o zkratku pro vlastnosti `justify-items` a `align-items`.

Zapisuje se takto:

```
place-items:
  <hodnota align-items>
  <hodnota justify-items>;
```



→ vrdl.in/csspi

Nastavuje výchozí hodnotu obou vlastností pro všechny položky uvnitř kontejneru. Pokud v deklaraci neuvedete druhou hodnotu, použije se první hodnota pro obě vlastnosti, ale jen za předpokladu, že je pro obě vlastnosti platná. Pokud tomu tak není, máte smůlu a bude neplatná celá deklarace.

Použití této zkratky dává smysl hlavně pro CSS grid, protože ve flexboxu je vlastnost `justify-items` a kvůli tomu i zkratka `place-items` ignorována.

Jednoduchý příklad

V naší ukázce definujeme třísloupcový kontejner gridu. Všechny tři položky mají omezenou výšku i šířku, aby byl hezky vidět efekt zarovnání, který způsobuje vlastnost `justify-items`. Využíváme zde tuto hodnotu:

```
.container {
  place-items: end right;
}
```

Jak jste asi poznali, položky zarovnáváme svisle dolů (`end`) a vodorovně doprava (`right`).

```
place-items: end right;
```

Hodnota vlastnsoti place-items v Chromu při zapnutém grid overlay.

CodePen: vrdl.in/sa51e

Možné hodnoty

Podívejte se na referenční příručku k oběma vlastnostem, pro které je `place-items` zkratkou:

- `justify-items`
- `align-items`

Podpora v prohlížečích

Použití ve flexboxu smysl nedává, ale v rámci CSS gridu má `place-items` velmi slušnou podporu ve všech moderních prohlížečích. MSIE 11 vlastnost nepodporuje, to ale nevadí.

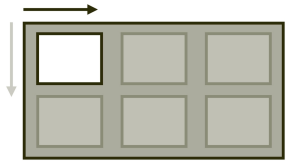
Pro více informací se podívejte na caniuse.com/place-items.

Justify-self: zarovnání položky na hlavní ose

Vlastnost `justify-self` určuje zarovnání položky na hlavní ose (jinak též řádkové ose).

Je dobré zmínit, že uvnitř buněk tabulek a ve flexboxu je vlastnost `justify-self` ignorována. V layoutu pomocí gridu se položka zarovnává uvnitř své oblasti, což je obvykle buňka mřížky.

U flexboxu je možné pro zarovnání položek na hlavní ose využít klasickou metodu s automatickým vnějším okrajem `margin:auto`, podobně jako u `justify-items`.



→ vrdl.in/cssjs

Příklad: margin má přednost před justify-self

V naší ukázce definujeme třísloupcový kontejner gridu. Poslední, jinak zbarvenou položku pak zarovnáváme pomocí `justify-self`.

HTML vám asi bude povědomé:

```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
</div>
```

Definice třísloupcového kontejneru gridu pak vypadá zhruba následovně:

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  height: 10em;
}
```

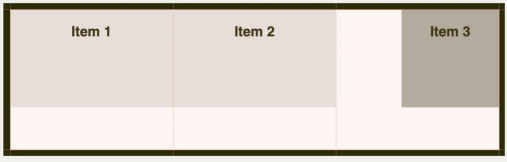
Nakonec deklarace pro poslední položku:

```
.item--3 {
  margin-right: auto;
  justify-self: end;
}
```

Tady vlastně zarovnáváme dvěma způsoby. Všimli jste si toho? Jednou doleva (`margin-right:auto`) a jednou doprava (`justify-self:end`). Co myslíte, bude platit poslední deklarace, jako je tomu u CSS vždy? Nikoliv, zarovnání pomocí vnějších okrajů zde má přednost navzdory kaskádě.

Zkuste si tu první deklaraci schválně v živé ukázce odmazat.

```
.item--3 {
  margin-right: auto;
  justify-self: end;
}
```



Vlastnost margin má přednost před justify-self.

Následuje několik užitečných vysvětlujících odrážek:

- První dvě položky nemají vlastnost `justify-self` nastavenou, takže získají výchozí hodnotu `stretch` a roztáhnou se do celé šířky prostoru buňky.
- Poslední položka má nastaveno `justify-self:end`, takže by se měla „scvrknout“ na přirozenou šířku podle obsahu a zarovnat ke konci prostoru buňky, což je zároveň pravá hrana kontejneru.
- Vyhrává ovšem deklarace `margin-right:auto`, která buňku zarovná na začátek prostoru buňky, a funguje tedy stejně jako `justify-self:start`.

CodePen: vrdl.in/zmso0

Příklad: proč to ve flexboxu nefunguje

V dalším CodePenu je vidět, že `justify-self` ve flexboxu opravu nefunguje. Pokouším se tam stylovat poslední položku pomocí `justify-self:end`. A nic.

CodePen: vrdl.in/eo9cq

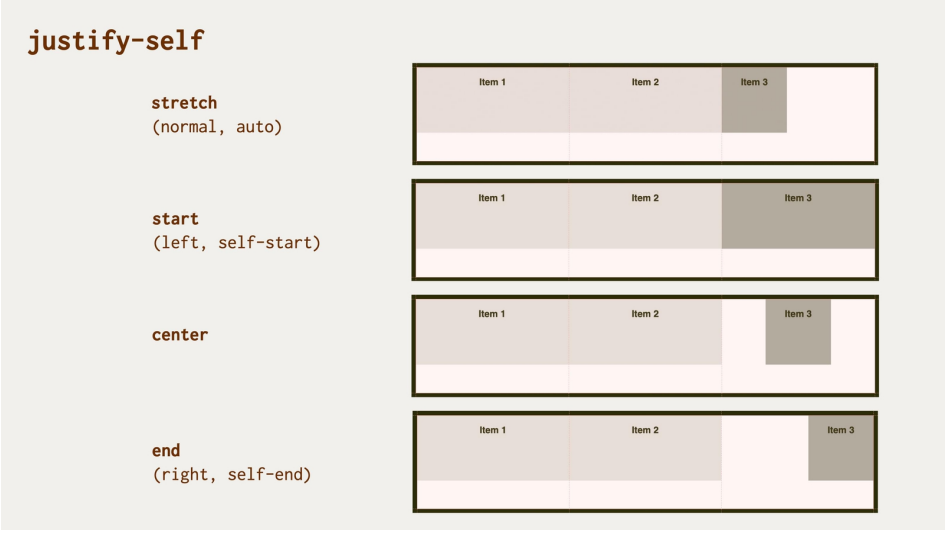
Zde je potřeba vysvětlit, proč tomu tak je.

Prvním důvodem je, že ve flexboxu nejsou vlastnosti `justify-self` a tím pádem ani `justify-items` potřeba – můžete totiž úplně v pohodě použít `margin` a jeho hodnotu `auto`.

Za druhé, u flexboxu se – na rozdíl od gridu – počítá s přítomností více položek na hlavní ose. Pravděpodobně nechcete dělat flexboxové rozvržení s jednou položkou. Nebo chcete, ale pro tohle flexbox vymyšlený nebyl. Pro zarovnání více položek je dobrá vlastnost `justify-content`.

Mám tedy jednu dobrou a jednu špatnou zprávu. Ta dobrá: Pro zarovnání položek na hlavní ose ve flexboxu používejte `margin` nebo `justify-content`. A ta špatná? Budete si to muset zapamatovat. Zabývá se tím krásná diskuze „In CSS Flexbox, why are there no ‚justify-items' and ‚justify-self' properties?" na Stack Overflow. vrdl.in/qabny

Možné hodnoty zarovnání



Hodnoty vlastnosti justify-self.

Vlastnosti `justify-self` můžete předávat všechny hodnoty z jednotlivých obecných kategorií klíčových slov specifikace CSS Box Alignment:

Základní

- `auto` (výchozí)
  Podědí se hodnota `justify-items` od rodičovského elementu. Pokud zde žádný není, dostane prvek hodnotu `normal`.
- `normal`
  V CSS gridu jde jen o jiný zápis pro hodnotu `stretch`, ale například v blokových layoutech (`display:block`) je to alternativa pro `start`.
- `stretch`
  Položka rozšíří své rozměry tak, aby v kontejneru nezbylo žádné volné místo. Pokud jsou položky menší než kontejner, jejich velikost se zvětší rovnoměrně (nikoliv proporcionálně), přičemž stále respektují omezení uložená vlastnostmi jako `max-width`/`max-height`.

Po dekódování zašmodrchaností specifikace můžeme konstatovat, že výchozí hodnota je vždy `stretch`, tedy roztažení do šířky.

Poziční

- `center`
  Položka se centruje doprostřed jí určeného prostoru.
- `start`
  Položka se zarovnává k hraně začátku jí určeného prostoru.
- `end`
  Položka se zarovnává k hraně konce jí určeného prostoru.
- `self-start`
  Hodnota se chová jako `start`.
- `self-end`
  Hodnota se chová jako `end`.
- `flex-start`
  Chová se jako `start`.
- `flex-end`
  Chová se jako `end`.
- `left`
  Chová se jako `start`.
- `right`
  Chová se jako `end`.

Podle účaří

- `first baseline`
  Zarovnání na účaří prvního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `start`.
- `last baseline`
  Zarovnání na účaří posledního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `end`.
- `baseline`
  Zkratka pro `first baseline`.

Pro přetečení

- `safe`
  Pokud má položka v daném způsobu zarovnání přetéct z obou stran, bude zarovnání změněno takovým způsobem, aby byl vidět začátek položky, například tak, aby bylo možné přečíst začátek textu.
- `unsafe`
  Vždy dostane přednost poziční zarovnání, bez ohledu na to, zda bude oříznutý obsah čitelný nebo ne.

Toto v žádném prohlížeči zatím nefunguje.

Podpora v prohlížečích

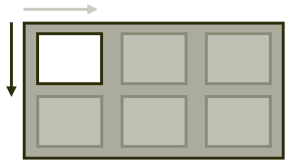
V rámci flexboxu a v tabulkovém layoutu tuto vlastnost nemůžete použít v žádném prohlížeči.

Při použití s `display:grid` je podpora v prohlížečích plná, jen v Internet Exploreru je hlášeno několik bugů. Více na CanIUse. caniuse.com/justify-self

Align-self: zarovnání položky na příčné ose

Vlastnost `align-self` nastavuje zarovnání položky na příčné (jinak též blokové) ose layoutu.

V layoutu tvořeném gridem se položka zarovnává uvnitř své oblasti, což je obvykle buňka mřížky. V rozvržení flexboxem jde o zarovnání položky v celém prostoru příčné osy.



→ vrdl.in/cssas

Vlastnost `align-self` je specifikací předurčená pro flexbox, grid, absolutně pozicované prvky, ale ne pro buňky tabulky nebo blokový layout tvořený například pomocí „floatů“.

Příklad: margin má přednost před align-self

V první ukázce definujeme třísloupcový kontejner gridu a poslední položku se snažíme zarovnat jinak než její první dvě kolegyně.

HTML už znáte:

```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
</div>
```

CSS pro deklaraci gridu a té naší specificky zarovnané poslední položky:

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  height: 10em;
}

.item--3 {
  margin-top: auto;
  align-self: start;
}
```

Stejně jako u `justify-self` i tady zarovnáváme dvěma způsoby. Jednou dolů (`margin-top:auto`) a jednou nahoru (`align-self:start`). A opět platí – zarovnání pomocí vnějších okrajů zde má přednost navzdory kaskádě.

```
.item--3 {
  margin-top: auto;
  align-self: start;
}
```

Vlastnost margin má přednost před align-self.

Vysvětleme:

- Rodičovský prvek `.container` má nastavenou výšku (`height:10em`), abychom viděli efekt zarovnání na blokové ose.
- První dvě položky nemají vlastnost `align-self` nastavenou, takže získají výchozí hodnotu `stretch` a roztáhnou se do celé výšky prostoru buňky.
- Poslední položka má nastaveno `align-self:start`, takže by se měla „scvrknout“ na přirozenou výšku podle obsahu a zarovnat k horní hraně prostoru buňky.
- Vyhrává ovšem deklarace `margin-top:auto`, která buňku zarovná na začátek prostoru buňky, a funguje tedy stejně jako `align-self:end`.

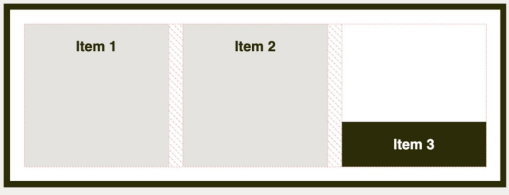
CodePen: vrdl.in/bw6hx

Příklad: align-self ve flexboxu

V druhé ukázce si pohrajeme s flexboxem. HTML je stejné, CSS se mění:

```
.container {
  display: flex;
  height: 10em;
}

.item {
  flex: 1;
}

.item--3 {
  align-self: flex-end;
}
```

```
.item--3 {
  align-self: flex-end;
}
```



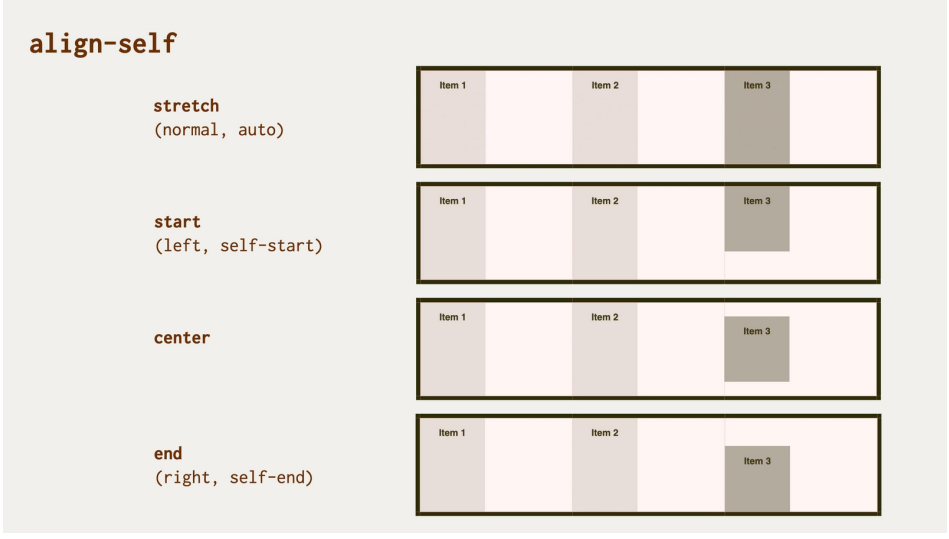
Vlastnost align-self ve flexboxu.

Nyní přichází vysvětlení:

- Rodičovský kontejner má deklarováno `display:flex` a položky `flex:1`, takže se roztáhnou do celé šířky.
- Na výšku jsou položky ovlivněné výchozí hodnotou (`align-self:stretch`).
- Třetí položka `.item--3` je pak díky `align-self:flex-end` zarovnaná ke spodní hraně kontejneru. Jsme ve flexboxu, takže hodnoty jako `end` by zde nezabraly.

CodePen: vrdl.in/pf4xm

Možné hodnoty



Hodnoty vlastnosti align-self.

Také vlastnosti `align-self` můžete předávat všechny hodnoty z jednotlivých obecných kategorií klíčových slov specifikace CSS Box Alignment:

Základní

- `auto` (výchozí)
  Podědí se hodnota `align-items` od rodičovského elementu. Pokud zde žádný není, dostane prvek hodnotu `normal`.
- `normal`
  V CSS gridu, flexboxu ale i ve většině systémů pro layout půjde jen o jiný zápis pro hodnotu `stretch`.
- `stretch`
  Položka rozšíří své rozměry tak, aby v kontejneru nezbylo žádné volné místo. Pokud jsou položky menší než kontejner, jejich velikost se zvětší rovnoměrně (nikoliv proporcionálně), přičemž stále respektují omezení uložená vlastnostmi jako `max-width`/`max-height`.

Opět platí, že v zásadě je výchozí roztažení na výšku – takže hodnota `stretch`.

Poziční

- `center`
  Položka se centruje doprostřed jí určeného prostoru.
- `start`
  Položka se zarovnává k hraně začátku jí určeného prostoru.
- `end`
  Položka se zarovnává k hraně konce jí určeného prostoru.
- `self-start`
  Hodnota se chová jako `start`.
- `self-end`
  Hodnota se chová jako `end`.
- `left`
  Chová se jako `start`.
- `right`
  Chová se jako `end`.
- `flex-start`
  Zarovná se k horní hraně flexbox kontejneru. Je použitelná jen ve flexboxu.
- `flex-end`
  Zarovná se ke spodní hraně flexbox kontejneru. Je použitelná jen ve flexboxu.

Podle účaří

- `first baseline`
  Zarovnání na účaří prvního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `start`.
- `last baseline`
  Zarovnání na účaří posledního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `end`.
- `baseline`
  Zkratka pro `first baseline`.

Pro přetečení

- `safe`
  Pokud má položka v daném způsobu zarovnání přetéct z obou stran, bude zarovnání změněno takovým způsobem, aby byl vidět začátek položky, například tak, aby bylo možné přečíst začátek textu.
- `unsafe`
  Vždy dostane přednost poziční zarovnání, bez ohledu na to, zda bude ořízнutý obsah čitelný nebo ne.

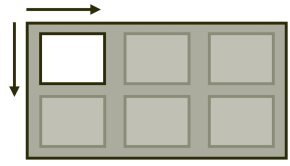
Toto v době psaní podporuje jen Firefox.

Podpora v prohlížečích

Vlastnost `align-self` má prakticky plnou podporu jak pro flexbox, tak pro layout do mřížky. V Internet Exploreru 11 je ovšem nutné použít vlastnost `-ms-grid-row-align`, což ale do `align-self` přeloží Autoprefixer.
Více na CanIUse. caniuse.com/align-self

Place-self: zarovnání položky na obou osách

Vlastnost `place-self` určí zarovnání položky na hlavní i příčné ose pro layouty v CSS.



→ vrdl.in/cssps

Tato vlastnost je zkratkou pro další dvě – `align-self` a `justify-self`.

V obecné rovině se zapisuje takto:

```
place-self: <hodnota align-self> <hodnota justify-self>;
```

Pokud na druhou hodnotu zapomenete, použije se první pro oba směry, ale jen za předpokladu, že je pro obě vlastnosti platná. Pokud tomu tak není, máte smůlu a bude neplatná celá deklarace.

Zkratku `place-self` není možné použít ve flexboxu, protože tam nepochodíte s její důležitou částí – zarovnáním položky ve směru hlavní osy (`align-self`).

Příklad

V naší ukázce definujeme třísloupcový kontejner gridu a poslední položku zarovnáme pomocí `place-self` trochu jinak.

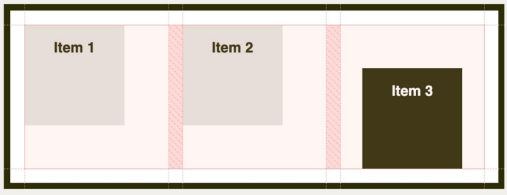
```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
</div>
```

CSS:

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  height: 10em;
}

.item--3 {
  place-self: end center;
}
```

Všechny tři položky mají omezenou výšku i šířku, aby byl hezky vidět efekt zarovnání.

```
.item--3 {
  place-self: end center;
}
```



Hodnoty vlastnosti place-self. Poslední položka nám uskočila kvůli jinému nastavení, viz kód výše.

Jak jste asi poznali, položky zarovnáváme svisle dolů (`end`) a vodorovně na buňky (`center`).

CodePen: vrdl.in/ygipw

Možné hodnoty

Pro nalezení všech možných hodnot se podívejte na referenční příručky k oběma vlastnostem, pro které je `place-self` zkratkou:

- `justify-self` – zarovnání na hlavní ose
- `align-self` – zarovnání na příčné ose

Podpora v prohlížečích

Vlastnost `place-self` v době psaní tohoto textu nepodporuje MSIE.

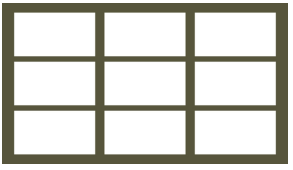
Ve flexboxových rozvrženích nemůžete tuto vlastnost použít v žádném prohlížeči, protože to nedovoluje specifikace.

Aktuální informace hledejte na CanIUse.com. caniuse.com/place-self

Gap: mezera mezi položkami

Pomocí CSS vlastnosti gap můžeme definovat mezery v rozvrženích vytvářených pomocí CSS layoutů. Téhle mezeře se občas i v češtině podle anglického originálu říká „gutter“. A brzy také možná „gap“.

Od příchodu Safari verze 14.1 je možné gap ve všech prohlížečích používat nejen v rámci gridu, ale také ve flexboxu. To je skvělé, a taky proto se vyplatí tuto vlastnost umět použít.



→ vrdl.in/gap

Zápis gap je zkratkou pro jiné dvě vlastnosti:

- row-gap – prostor mezi řádky (v „blokovém“ či svislém směru)
- column-gap – prostor mezi sloupci (tedy v „inline“, obvykle vodorovném směru)

Zkratka se zapisuje takto:

```
gap: <hodnota row-gap> <hodnota column-gap>;
```

Nastavujeme zde, jak je v CSS zvykem, nejprve svislý a až pak vodorovný směr. Druhá hodnota je volitelná. Pokud se neuvede, použije se jedna hodnota pro oba směry. I to je ostatně ve stylech běžné.

Příklad

Vezměme ukázku se čtyřmi položkami v layoutu:

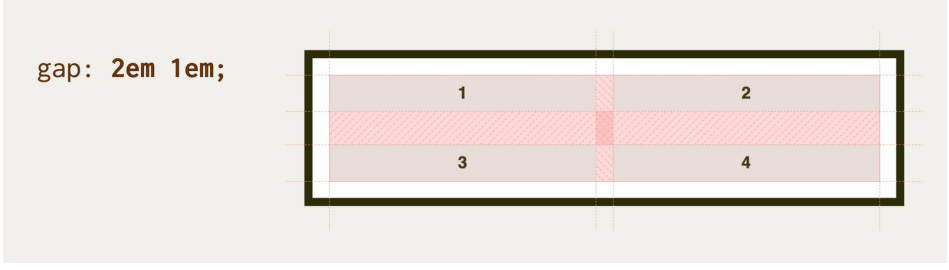
```
<div class="container">
  <p class="column">1</p>
  <p class="column">2</p>
  <p class="column">3</p>
  <p class="column">4</p>
</div>
```

Layout v CSS definujeme následovně:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 2em 1em;
}
```

Vysvětleme si to:

- Deklarace `display:grid` zajistí zobrazení pomocí CSS gridu.
- Vlastnost `grid-template-columns` definuje podobu mřížky. Zde jde o sloupce o rovnoměrné šířce.
- `gap: 2em 1em` je instrukce pro vložení mezery svisle a pak i vodorovně.



Takto vypadá odlišné nastavení vlastnosti gap pro svislý a vodorovný směr.

CodePen: vrdl.in/kyam7

Totéž bychom samozřejmě mohli zapsat v nezkrácených deklaracích následovně:

```
.container {
  row-gap: 2em;
  column-gap: 1em;
}
```

Raději gap než margin či padding (ale nic proti nim!)

Vlastnost `gap` je pro definování mezer v layoutu daleko efektivnější než `padding` nebo `margin`. Nijak se totiž nepočítá do šířky ani výšky položky layoutu a také se vždy vykresluje jen mezi položkami samotnými.

Je také pěkné si nastavit mezery mezi prvky v layoutu pro celý kontejner na jednom místě. Z toho důvodu právě vlastnost `gap` vznikla.

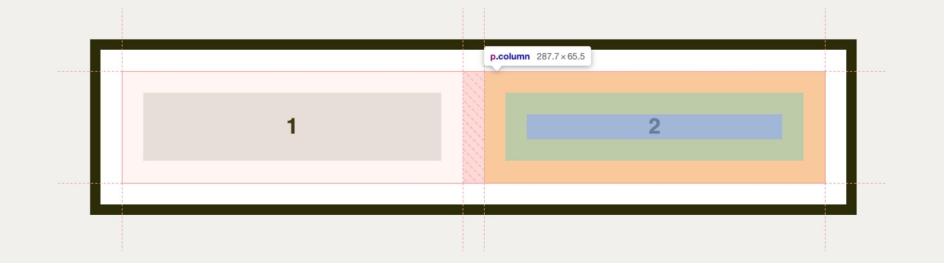
Je však samozřejmě možné a bezpečné zároveň nastavovat mezery pomocí vnějších i vnitřních okrajů prvku nebo případně `gap` s dvojkou `margin`/`padding` kombinovat. Toho se určitě nebojte. Jen si pak dejte pozor na interpretaci v prohlížečích, protože viditelná mezera vám naroste:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 1em;
}

.column {
  padding: 1em;
  margin: 1em;
}
```

Anketní otázka: Jak velká bude mezera mezi položkou 1 a 2? Zvládnete ji zodpovědět, ještě než se podíváte na obrázek?

Nechám vám chvilku času… Ještě chvilku…

A teď už přichází obrázek:



Vlastnost gap v kombinaci s marginem a paddingem.

Ano, viditelná mezera mezi položkami bude široká celé `3em`. Sečteme dva vnější okraje (`margin`) a mezeru (`gap`).

CodePen: vrdl.in/6xz53

Možné hodnoty

Následuje přehled možných hodnot vlastnosti gap. Čistě pro inspiraci, nalože te s tím dle svého.

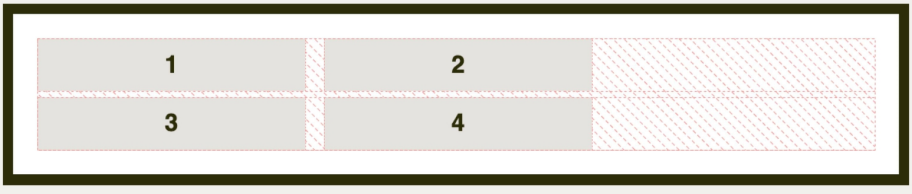
Různé hodnoty pro svislý i vodorovný směr

```
.container {
  gap: 5px 1rem;
}
```

Jak už jsem uvedl, toto je možné. První číslo definuje svislý směr, druhé vodorovný.

Pojďme si to vyzkoušet na flexboxovém layoutu, který jsme ještě vlastností gap nestihli potrápit:

```
.container {
  display: flex;
  flex-wrap: wrap;
  gap: 5px 1rem;
}

.column {
  width: 10em;
}
```



Gap ve flexboxu. I tady to mezerám sekne, že ano…?

Raději si to zopakujme. Zápis `gap: 5px 1rem` říká, že svisle chci mezi řádku mezeru `5px` a vodorovně mezi sloupci pak mezeru `1rem`.

CodePen: vrdl.in/rmtfs

Použití funkce calc()

Uvádění výpočetní funkce `calc()` se v hodnotách `gap` může hodit:

```
.container {
  gap: calc(1rem - 2px);
}
```

CodePen: vrdl.in/4t652

A k čemu že se funkce `calc()` může hodit? Příkladem budiž odečtení šířky rámečků (`border`) buněk layoutu z celkové šířky mezery.

Neznáte funkci `calc()`? Podívejte se na speciální zmínku v poslední kapitole této knihy.

Klíčové slovo normal

Šup s ním hned do vody, tedy do ukázky kódu:

```
.container {
  gap: normal;
}
```

Slovo `normal` představuje použitou hodnotu `1em` u vícesloupcového layoutu a hodnotu `0px` v kontextu gridu a flexboxu. Asi to není zase tak moc zajímavé… Já jen… Kdybyste se náhodou ptali… Nebo vám to někdo položil jako otázku v testu.

Procenta a jejich uvádění ve svislém směru

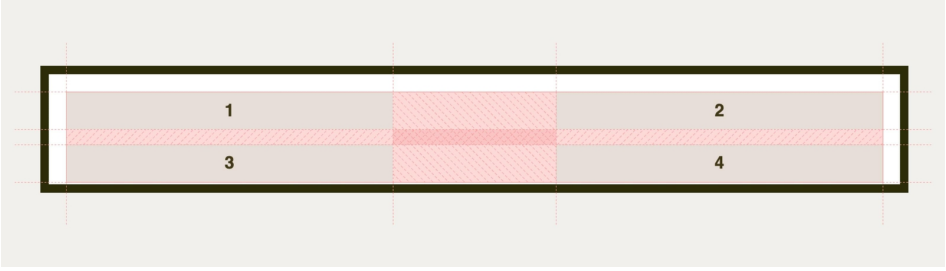
Procentuální hodnoty můžete chtít použít, ale dejte si pozor na hodnoty ve svislém směru.

```
.container {
  gap: 10%;
}
```

Ve vodorovném směru je to jednoduché – spočítá se deset procent ze šířky rodičovského kontejneru a tato hodnota se vloží jako mezera mezi prvky.

Zajímavější je svislý směr. V layoutu tvořeném mřížkou se spočítá deset procent z výchozí výšky rodičovského kontejneru, což je výška před aplikováním mezery pomocí vlastnosti `gap`.

Výsledkem je, že procentuální hodnota pro gap aplikovaná na svislý směr vám velmi často způsobí přetečení položek z rozměru kontejneru.

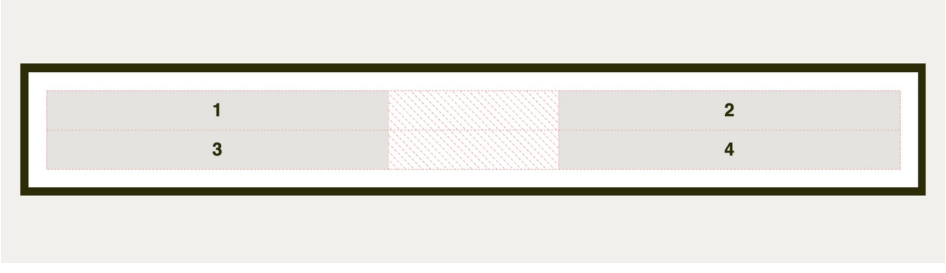


To byste nečekali. Použití procent ve vlastnosti gap ve svislém směru rozbije náš pěkný layout.

Pravděpodobně se vám tedy stane, že mezera vytlačí spodní prvky z kontejneru.

CodePen: vrdl.in/ervja

V případě flexboxového layoutu a neznámé výšky kontejneru se procentuální gap ve svislém směru vůbec nezapočítá. Je z něj čistá nula. Ptáte se, kdy je výška kontejneru neznámá? Inu, ve flexboxu skoro vždy – dokud ji výslovně nedefinujete.



A tohle už byste asi vůbec nečekali. Použití procent ve vlastnosti gap ve svislém směru ve flexboxu prostě nefunguje.

Zkoušení naživo je možné opět v následující ukázce.

CodePen: vrdl.in/p9xmy

Co byste o gap měli vědět?

Když už jsme v tom, mám pár poznámek. Doslova pár:

1. Mezery tvořené `gap` mají vliv na minimální rozestupy mezi položkami. Je však možné ještě další vnitří mezery přidat pomocí vlastností jako `justify-content` nebo `align-content`. Jejich hodnota `space-between` má podobný dopad jako použití `gap` a je možné je vzájemně kombinovat.
2. Když už se `gap` dá použít všude, nedá se použít i pro mezery mezi buňkami uvnitř `<table>`? Nedá, děkujeme za optání. Tabulková zobrazení místo toho používají vlastnost `border-spacing`.

Podpora v prohlížečích

Pokud jde o moderní prohlížeče, vlastnost `gap` ve flexboxu a gridu podporují všechny.

- Grid: Prakticky plná podpora. Internet Explorer 11 vlastnost `gap` nepodporuje, ale to je možné dohnat použitím nástroje Autoprefixer.
- Flexbox: Nepodporuje MSIE 11.
- Vícesloupcový layout: Nepodporuje MSIE 11 a zatím ani Safari.

Takže pokud potřebujete mezery v gridu a flexboxu a neřešíte Explorer, jste v suchu.

Dříve jen v gridu

Dřívější zápisy „děrovacích“ vlastností byly ve specifikaci definovány jinak, s prefixem `grid-`: `grid-row-gap`, `grid-column-gap` a `grid-gap` a zaměřené čistě jen na CSS grid. Nyní jsou ale z této části specifikace vyjmuté a vyvíjené pod samostatným modulem CSS Box Alignment.

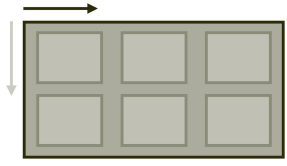
Logicky totiž nespadají jen do možnosti definovat layout v mřížce, ale také ve flexboxu nebo vícesloupcovém layoutu.

Vlastnost `gap`, tedy bez prefixu `grid-`, je podporována v Chromu 68+, Safari 11.2 Release 50+ a Opeře 54+. Ale to už je dneska vlastně jen historické okénko. Detailní informace o podpoře jsou na CanIUse. caniuse.com/gap

Justify-content: rozdělení prostoru na hlavní ose

Vlastnost CSS `justify-content` definuje, jak prohlížeč distribuuje prostor mezi položkami obsahu podél hlavní (nebo řádkové) osy kontejneru layoutu.

Využití je možné ve všech moderních systémech layoutu: flex, grid i multicol.



→ vrdl.in/cssjc

Ve flexboxu je to občas zbytečné

Pro pořádek zmiňme, že tahle vlastnost nebude pro flexbox zase tak užitečná. Zbylý prostor se ve flexboxu rozdělí až poté, co se započtou hodnoty `auto` vnějších okrajů (`margin`), a až poté, co se započtou hodnoty délek položek rozvržení.

Stačí tedy existence jedné flexboxové položky s vlastnostmi `flex` nebo `flex-grow` nastavená na hodnotu `1` a více a žádný volný prostor zde nezbude.

Jednoduchý příklad

V naší ukázce definujeme třísloupcový kontejner gridu. Začneme s HTML:

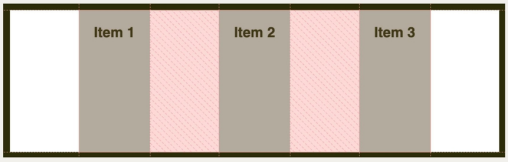
```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
</div>
```

CSS:

```
.container {
  display: grid;
  grid-template-columns: repeat(3, minmax(3em, 5em));
  height: 10em;
  justify-content: space-around;
}
```

Položky gridu jsme ale, my zlí experimentátoři, přinutili, aby držely minimální a maximální šířku – pomocí `minmax(3em, 5em)`.

```
.container {
  justify-content: space-around;
}
```



Budiž prostor mezi položkami! Vlastnost justify-content v nastavení space-around.

CodePen: vrdl.in/2z1yj

Deklarace `justify-content:space-around` zajistí rozdělení volného prostoru na šířku tak, aby mezi položkami a kontejnerem byla mezera o polovinu menší, než je mezi jednotlivými položkami.

Můžete si vyzkoušet ještě jedno demo. Je totožné, jen tentokrát rozvržené pomocí flexboxu.

CSS tam vypadá zhruba takto:

```
.container {
  display: flex;
}
```

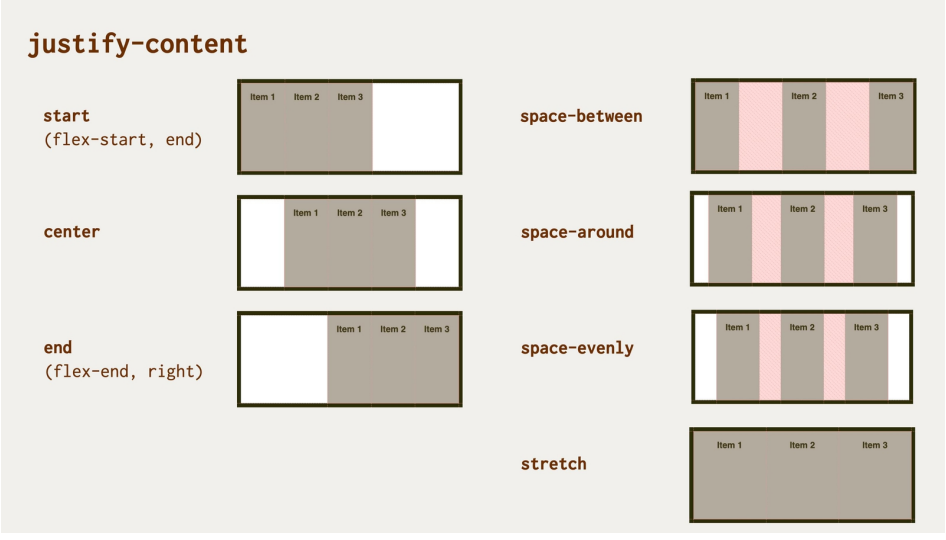
```
.container {
  display: flex;
  justify-content: space-around;
}
```

Nastavení space-around, tentokrát ve flexboxu.

Vzhledem k tomu, že jsme na položky neaplikovali vlastnost `flex` nebo `flex-grow`, dělení prostoru pomocí `justify-content` zůstává v platnosti.

CodePen: vrdl.in/vxj62

Možné hodnoty



Hodnoty vlastnosti justify-content.

Vlastnosti `justify-content` můžete předávat všechny hodnoty z jednotlivých obecných kategorií klíčových slov:

Základní

- `normal` (výchozí)
  V CSS gridu i flexboxu bude odpovídat hodnotě `start`. Položky se tedy zarovnají na začátek osy, takže obvykle doleva.

Zbylý prostor

- `space-between`
  Volné místo se rovnoměrně rozdělí mezi položky, přičemž první a poslední je zarovnaná s hranou kontejneru.
- `space-around`
  Volné místo se rovnoměrně rozdělí mezi položky a polovina mezery mezi položkami se vloží mezi hrany kontejneru a první a poslední položku.

- `space-evenly`
  Volné místo se rovnoměrně rozdělí mezi položky i mezi první a poslední položku a okraje kontejneru. Tato vlastnost není v kombinaci s flexboxem podporována v MSIE 11.
- `stretch`
  Položky rozšíří své rozměry tak, aby v kontejneru nezbylo žádné volné místo. Pokud jsou položky menší než kontejner, jejich velikost se zvětší rovnoměrně (nikoliv proporcionálně), přičemž stále respektují omezení uložená vlastnostmi jako `max-width`/`max-height`. Tato hodnota není v kombinaci s flexboxem podporována v MSIE 11.

Poziční

- `center`
  Položky se centrují doprostřed kontejneru. Ve flexboxu nepodporuje MSIE.
- `start`
  Položky se zarovnají k hraně začátku kontejneru. Ve flexboxu nepodporuje MSIE.
- `end`
  Položky se zarovnají k hraně konce kontejneru.
- `flex-start`
  Chová se jako `start`, použitelné jen ve flexboxu.
- `flex-end`
  Chová se jako `end`, použitelné jen ve flexboxu.
- `left`
  Chová se jako `start`.
- `right`
  Chová se jako `end`.

Hodnoty `left`, `right`, `start` a `end` zatím nemají u této vlastnosti dobrou podporu v prohlížečích při použití ve flexboxu.

Zarovnání na účaří

- `first baseline`
  Zarovnání na účaří prvního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `start`.

- `last baseline`
  Zarovnání na účaří posledního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `end`.
- `baseline`
  Zkratka pro `first baseline`.

Toto v době psaní podporuje Chrome a Edge.

Pro přetečení

- `safe`
  Pokud má položka v daném způsobu zarovnání přetéct z obou stran, bude zarovnání změněno takovým způsobem, aby byl vidět začátek položky, například tak, aby bylo možné přečíst začátek textu.
- `unsafe`
  Vždy dostane přednost poziční zarovnání, bez ohledu na to, zda bude ořízntutý obsah čitelný nebo ne.

Toto v době psaní podporuje jen Firefox.

Podpora v prohlížečích

V layoutech postavených na gridu je vlastnost u základních hodnot plně podporována, s výjimkou MSIE 11. Použitím nástroje Autoprefixer je ale možné to dohnat.

Ve flexboxových rozvrženích funguje `justify-content` dobře i v Internet Exploreru.

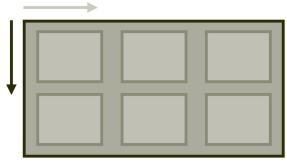
Více na CanIUse. caniuse.com/justify-content.

Align-content: rozdělení prostoru na příčné ose

Vlastnost `align-content` definuje, jak prohlížeč distribuuje prostor mezi položkami obsahu podél příčné (nebo blokové) osy kontejneru layoutu.

Ve specifikaci se počítá s použitím pro flexbox, grid, vícesloupcový layout, ale taky pro blokové prvky, kde to ale zatím použít nejde.

Tato vlastnost zarovnává ve svislém směru, a tak nemá pochopitelně vliv na jednořádkové flexboxové kontejnery, tedy prvky, kterým nastavíme deklaraci `flex-wrap:nowrap`.



→ vrdl.in/cssac

Jednoduchý příklad

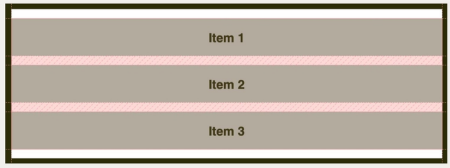
V naší ukázce definujeme třířádkový kontejner gridu:

```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
</div>
```

CSS:

```
.container {
  display: grid;
  grid-template-rows: repeat(3, 3em);
  height: 12em;
  align-content: space-between;
}
```

```
.container {
  display: grid;
  align-content: space-between;
}
```



Pomocí zápisu align-content:space-between se volný prostor hezky distribuuje na výšku mezi jednotlivé položky.

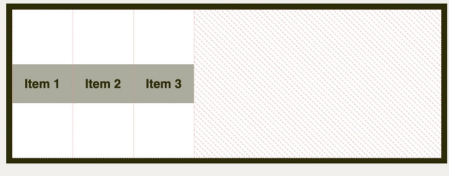
CodePen: vrdl.in/2bhd9

Můžete si vyzkoušet ještě jedno demo. Je totožné, jen tentokrát pro flexbox a má jinou hodnotu `align-content`.

CSS:

```
.container {
  border: 5px solid black;
  display: flex;
  flex-wrap: wrap;
  height: 12em;
  align-content: space-around;
}
```

```
.container {
  display: flex;
  align-content: space-around;
}
```

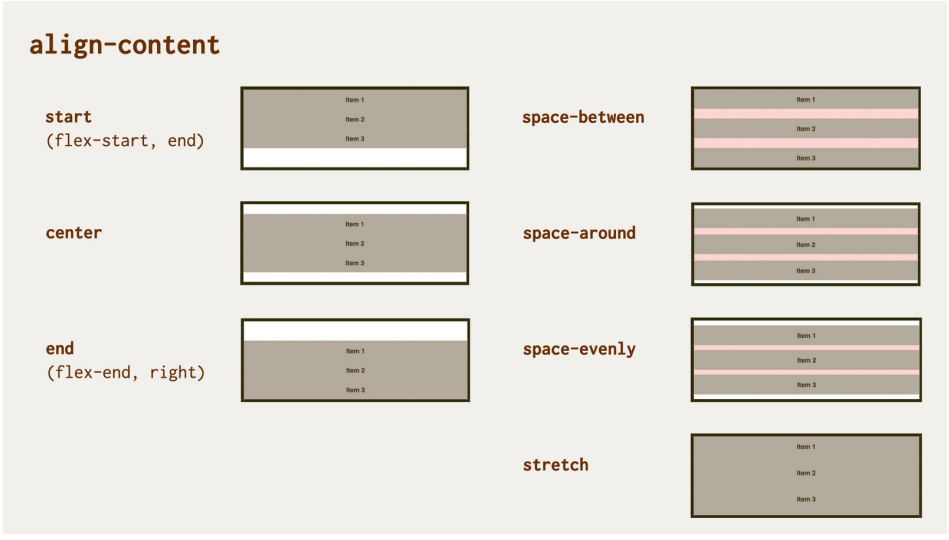


Zápis align-content:space-around rozdělí volný prostor na polovinu a přidá ji před a za buňku.

CodePen: vrdl.in/ngjor

Víte co? Raději se podívejte na všechny možné hodnoty, i tuhle mezi nimi najdete.

Možné hodnoty



Hodnoty vlastnosti align-content.

Vlastnosti `align-content` můžete předávat všechny hodnoty z jednotlivých obecných kategorií klíčových slov:

Základní

- `normal` (výchozí)
  V CSS gridu odpovídá hodnotě `start`, ve flexboxu je to zase `stretch`.

Zbylý prostor

- `space-between`
  Volné místo se rovnoměrně rozdělí mezi položky, přičemž první a poslední je zarovnaná s hranou kontejneru.
- `space-around`
  Volné místo se rovnoměrně rozdělí mezi položky a polovina mezery mezi položkami se vloží mezi hrany kontejneru a první a poslední položku.

- `space-evenly`
  Volné místo se rovnoměrně rozdělí mezi položky i mezi první a poslední položku a okraje kontejneru. Tato vlastnost není v kombinaci s flexboxem podporována v MSIE 11.
- `stretch`
  Položky rozšíří své rozměry tak, aby v kontejneru nezbylo žádné volné místo. Pokud jsou položky menší než kontejner, jejich velikost se zvětší rovnoměrně (nikoliv proporcionálně), přičemž stále respektují omezení uložená vlastnostmi jako `max-width`/`max-height`. Tato vlastnost není v kombinaci s flexboxem podporována v MSIE 11.

Poziční

- `center`
  Položky se centrují doprostřed kontejneru.
- `start`
  Položky se zarovnají k hraně začátku kontejneru, nefunguje ve flexboxu.
- `end`
  Položky se zarovnají k hraně konce kontejneru, nefunguje ve flexboxu.
- `flex-start`
  Chová se jako `start`, použitelné jen ve flexboxu.
- `flex-end`
  Chová se jako `end`, použitelné jen ve flexboxu.

Podle účaří

- `first baseline`
  Zarovnání na účaří prvního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `start`.
- `last baseline`
  Zarovnání na účaří posledního řádku. Pokud hodnotu v daném kontextu nelze použít, zarovná se jako `end`.
- `baseline`
  Zkratka pro `first baseline`.

Tyto hodnoty zatím nemají u popisované vlastnosti dobrou podporu v prohlížečích při použití ve flexboxu. V MSIE není podporována vůbec.

Zarovnání pro přetečení

- `safe`
  Pokud má položka v daném způsobu zarovnání přetéct z obou stran, bude zarovnání změněno takovým způsobem, aby byl vidět začátek položky, například tak, aby bylo možné přečíst začátek textu.
- `unsafe`
  Vždy dostane přednost poziční zarovnání, bez ohledu na to, zda bude ořízntutý obsah čitelný nebo ne.

Toto v době psaní podporuje jen Firefox.

Podpora v prohlížečích

V layoutech postavených na gridu je vlastnost u základních hodnot plně podporována, s výjimkou MSIE 11. Použitím nástroje Autoprefixer je ale možné to dohnat.

Ve flexboxových rozvrženích funguje `justify-content` dobře i v Internet Exploreru.

Více na CanIUse.com. caniuse.com/align-content.

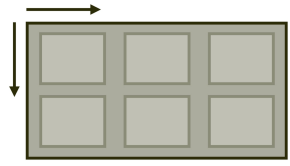
Place-content: rozdělení prostoru na obou osách

Úkolem vlastnosti `place-content` je specifikovat rozdělení prostoru mezi položkami na hlavní i příčné ose kontejneru pro layouty v CSS.

`place-content` je zkratkou pro tyto vlastnosti:

- `justify-content`
- `align-content`

Zapisuje se takto:



→ vrdl.in/csspc

```
place-content:
  <hodnota align-content>
  <hodnota justify-content>;
```

Nastavuje výchozí hodnotu obou vlastností pro všechny položky uvnitř kontejneru.

Pokud v deklaraci `place-content` neuvedete druhou hodnotu, použije se první hodnota pro obě vlastnosti, ale jen za předpokladu, že je pro obě vlastnosti platná. Pokud tomu tak není, máte smůlu a bude neplatná celá deklarace.

Připomínám, že ve flexboxu může být využití vlastnosti `justify-content` vcelku omezené. Pokud totiž použijete nastavení šířky položky pomocí `flex` nebo `flex-grow`, položka se roztáhne do volného prostoru a `justify-content` pak nebude mít co rozdělovat.

Jednoduchý příklad

V ukázce si hrajeme s dvousloupcovým kontejnerem gridu.

```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
</div>
```

CSS pro rozvržení:

```
.container {
  display: grid;
  grid-template-columns: repeat(2, 5em);
  height: 20em;
}
```

Všechny tři položky mají omezenou výšku i šířku:

```
.item {
  padding: 1em;
  height: 5em;
  width: 5em;
}
```

To aby byl hezky vidět efekt distribuce volného prostoru mezi položkami, který způsobuje vlastnost `place-content`.

```
.container {
  display: grid;
  place-content: space-between end;
}
```

Nastavení place-content:space-between end v gridu.

Využíváme zde tuto hodnotu:

```
.container {
  place-content: space-between end;
}
```

Položky tedy zarovnáváme svisle tak, aby mezi nimi vznikla rovnoměrná mezera (space-between), a vodorovně na konec hrany kontejneru (end).

CodePen: vrdl.in/zlhko

Možné hodnoty

Podívejte se na referenční příručku k oběma vlastnostem, pro které je `place-content` zkratkou:

- `justify-content`
- `align-content`

Podpora v prohlížečích

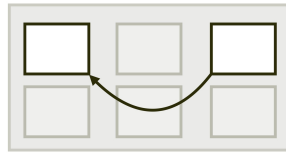
Vlastnost `place-content` funguje ve všech prohlížečích kromě MSIE 11.

Více informací hledejte na CanIUse. caniuse.com/place-items.

Order: změna pořadí položek

Vlastnost `order` nastavuje pořadí položky v kontejneru rozvržení v CSS.

Změna pořadí položek layoutu se občas může kodérkám a kodérům hodit proto, že zajistí odlišení *vizuálního* pořadí položek od pořadí *zdrojového* vycházejícího z HTML nebo lépe ze stromu DOM.



→ vrdl.in/ord

Jen pozor, kouzlo téhle vlastnosti je z pohledu přístupnosti zároveň rizikem.

Ale než se k tomu dostaneme, pojďme si vlastnost rozebrat, jak je zvykem.

Jednoduchý příklad s flexboxem

Pro účely demonstrace zneužijeme čtyři nevinné položky v kontejneru flexboxu:

```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
  <div class="item item--4">Item 4</div>
</div>
```

Vlastnost `order` nastavíme jen u třetí položky:

```
.item--3 {
  order: 1;
}
```

Kam si myslíte, že se třetí položka posune? Na první místo? Na druhé? Přemýšlejte.

Do obrázku jsem nakreslil nejen řešení, ale rovnou i hned dvě další možná nastavení.

Číselné hodnoty vlastnosti order vytvářejí nové řazení. Kladné za stávajícími položkami, záporné před nimi.

Vlastnost mění pořadí prvků možná trošku jinak, než byste čekali:

- `order:0` – u nulové hodnoty se od ovlivňované položky asi dá očekávat, že bude držet původní pozici. Nezklamala.
- `order:1` – kladné hodnoty vytvářejí novou řadu položek řazených od nejmenšího po největší číslo. Ale pozor – až *za* původními položkami, vlastností `order` nedotčenými.
- `order:-1` – ano, lze použít i záporná čísla. Slouží k vytváření pořadí *před* původní řadou položek.

Jen si s tím směle pohrejte v mém CodePenu.

HTML je klasika:

```
<div class="container">
  <div class="item item--1">Item 1</div>
  <div class="item item--2">Item 2</div>
  <div class="item item--3">Item 3</div>
  <div class="item item--4">Item 4</div>
</div>
```

CSS je flexbox:

```
.container {
  display: flex;
  gap: 1em;
  padding: 1em;
}

.item {
  flex: 1;
  padding: 1em;
  height: 5em;
  min-width: 5em;
}
```

Změnu pořadí děláme takto:

```
.item.item--3 {
  order: 1;
}
```

CodePen: vrdl.in/vqxi6

Pár věcí, které stojí za zapamatování

Jak je vidět, vlastnost `order`, jakkoliv se zdá jednoduchá a přímočará, trošku klame tělem. A to jsme ještě nezačali mluvit o přístupnosti.

Jsou zde ale i další vědomosti, o kterých byste měli mít ponětí, než si s `order` začnete hrát.

1. Pokud má více položek rozvržení nastavenu stejnou hodnotu `order`, seřadí se podle pořadí v DOMu.
2. Podle specifikace ovlivňuje vlastnost `order` také pořadí při vykreslování stránky prohlížečem, což má logiku.
3. Absolutně pozicované položky vždy dostanou nastavení `order:0`, takže drží pořadí vykreslení dle DOMu.

CodePen: vrdl.in/7yca0

Ošemetná změna pořadí z pohledu přístupnosti

Ve specifikaci se píše:

> **Autoři musí použít změnu pořadí pouze pro vizuální, nikoliv logické přeskupování obsahu.**

Dává to smysl. Když pomocí vlastnosti `order` nebo jiným způsobem změníte pořadí prvku na obrazovce, nijak nezměníte pořadí logické, tedy způsob, jakým obsah konzumují roboti nebo jakým probíhá sekvenční navigace na stránce.

Více to rozebírám v části o přístupnosti v CSS layoutu osmé kapitoly.

Raději si na vlastnost order dávejte pozor

Shrňme si to. Vlastnost `order` určitě může být v některých případech užitečná, nebo dokonce nezbytná pro změnu vizuálního pořadí.

Je zde však nebezpečí, že uživatelům tím zároveň rozbijeme navigaci z klávesnice a zrakově postiženým čtení obsahu přes odečítače.

Takže s `order` prosím zacházejte opatrně, jako se zápalkami.

Shrnutí sedmé kapitoly

1. **Jak rozlišit vlastnosti zarovnání na hlavní a na příčnou osu?**
   a) je to zhola nemožné, raději si hodit kostkou
   b) vlastnosti `justify-` zarovnávají na hlavní, `align-` na příčnou osu
   c) vlastnosti `align-` zarovnávají na hlavní, `justify-` na příčnou osu
2. **Jaký je rozdíl mezi vlastnostmi -items, -self, -content?**
   a) `-items` = položky, `-self` = položka, `-content` = volný prostor
   b) `-items` = položky, `-self` = kontejner, `-content` = obsah v položce
   c) je to chyták, není v tom řád, jdu se raději opít
3. **Jak zarovnat položky doprostřed buněk layoutu na vodorovné ose?**
   a) `align-items:center`
   b) `justify-items:center`
   c) `justify-layout:center`
4. **Jak zarovnat položku ke spodní a pravé hraně buňky layoutu?**
   a) `place-self:right bottom`
   b) `place-item:end end`
   c) `place-self:end end`
5. **Proč je vhodné používat pro mezery v layoutu gap?**
   a) další chyták, tohle naopak vůbec vhodné není
   b) protože je možné ji nastavit pro celý layout
   c) je to kratší slovo než `margin`
6. **Co udělá gap:normal v gridu a flexboxu?**
   a) mezeru nulové šířky a výšky
   b) mezeru o šířce `1em`
   c) mezeru o šířce `10px`
7. **Jak přesunout položku na první místo vizuálního pořadí?**
   a) `order:0`
   b) `order:1`
   c) `order:-1`

Řešení: 1. b), 2. a), 3. b), 4. c), 5. b), 6. a), 7. c).

08

CO BYSTE JEŠTĚ MĚLI VĚDĚT?

V této kapitole opustíme referenční příručky a prohloubíme si znalosti souvislostí. Začneme u podpory v prohlížečích a přes přístupnost a rychlost se dostaneme k vývojářským nástrojům v prohlížečích, které nám mohou být velice nápomocny.

Internet Explorer (je už mrtvý?)

Microsoft Internet Explorer (MSIE) dělá vývojářům starosti. Používat moderní vlastnosti jako CSS grid je složitější a někdy i nemožné, pokud je na projektu nutné podporovat tento starý prohlížeč od Microsoftu.

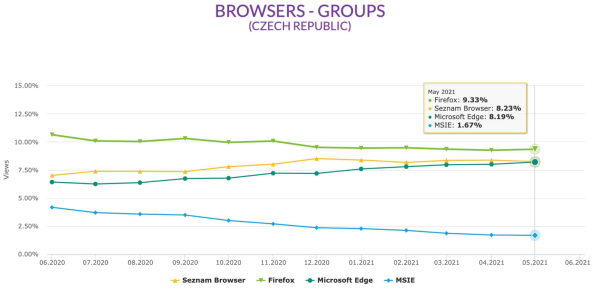
Pokud Internet Explorer stále ještě podporujete, myslím si, že byste od této praxe mohli začít upouštět. To je důvod, proč píšu tenhle text.

Velmi to záleží na cílové skupině konkrétního projektu, ale drtivá většina vývojářů si myslím ukončení podpory v roce 2022 může dovolit.

Toto je mé osobní rozloučení s prohlížečem, který do webdesignu přinesl mnoho dobrého. Ne, není to ironie. Jak brzy uvidíte, Explorer dříve stál na světlé straně Síly a možná vás překvapí, že to byl nejpokrokovější prohlížeč své doby.

Aktuální podíl MSIE mezi uživateli je v Česku mezi 1–2 %. A dál klesá

Podívejte se na graf. Je z něj myslím jasné, že i v České republice bude brzy možné přestat Internet Explorer podporovat.



Nejmenší z nejmenších. Podíl méně významných prohlížečů na trhu v ČR. Vývoj MSIE ukazuje světle modrá linka. Zdroj: rankings.gemius.com.

V Česku měl MSIE v polovině roku 2021 zastoupení kolem 1,5 % a začátkem roku 2022 už jen kolem 1 %. Tato čísla potvrzuje i jiný statistický web, StatCounter.

Daleko důležitější je ale trend vývoje. Zatímco začátkem roku 2019 používala dědeček prohlížeč ještě zhruba desetina uživatelů, zkraje roku 2020 už jej pro přístup na weby využívalo jen zhruba 5 %. Každý rok tedy jeho popularita klesne na polovinu a méně.

Ovšem bratrům Slovákům zde můžeme jen závidět. V zemi pod Tatrami je totiž zastoupení MSIE podle čísel Gemiusu už od roku 2021 nulové.

Výjimky potvrzují pravidlo. Sledujte statistiky a příjmy

Jak zjistit, jestli můžu Explorer přestat podporovat?

Podíl MSIE na používání se liší web od webu, takže například na svém blogu, Vzhůru dolů, nevidím v únoru 2022 prakticky žádné návštěvy od uživatelů s tímto prohlížečem. Na webech klientů ale Google Analytics ukazují čísla vyšší.

Podíl prohlížečů zjistíte například právě v Google Analytics (Publikum > Technologie > Prohlížeč).

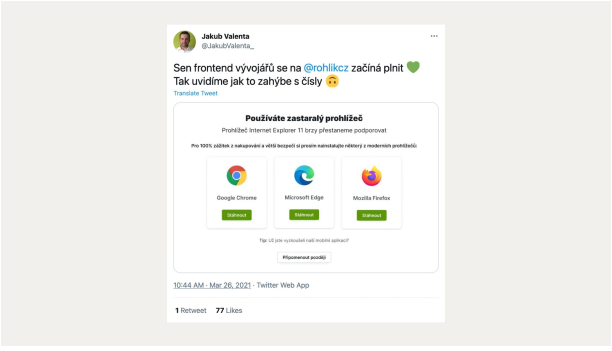
Je samozřejmě otázka, při jakém podílu na návštěvnosti je možné podporu prohlížeče vzdát. Obvykle se má za to, že podpora prohlížečů s podílem pod 1 % se nevyplácí, ale záleží na více faktorech:

- Kolik času a peněz musíte do podpory investovat. Troufám si říct, že toto číslo bude u Exploreru vysoké, protože už na příkladu CSS gridu je vidět, jak moc odlišné řešení je pro Explorer nutné dělat.
- Jak důležití jsou uživatelé prohlížeče pro výdělečnost vašeho projektu.

Ten druhý bod je přitom klíčový. Vyfiltrujte si v Google Analytics tržby uživatelů Exploreru a porovnejte je se svými náklady.

Takto to pojali například vývojářky a vývojáři v Rohlík.cz. Podíl MSIE je na Rohlíku nízký, jenže přepočteno na tržby se to pořád vyplatí.

Pro každého frontendistu je ovšem podpora takto zastaralého prohlížeče otrava. V Rohlíku tedy uživatelům Exploreru ukazují hlášku motivující je ke změně prohlížeče.

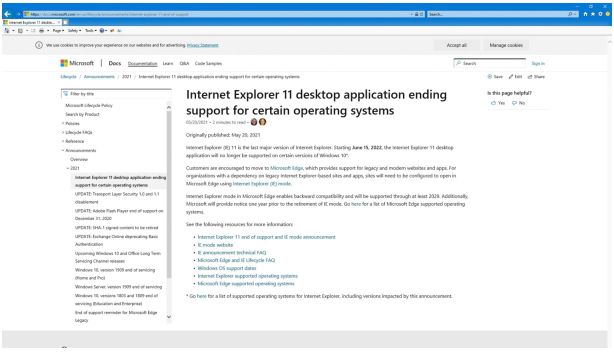


Máme vás rádi, ale nechtěli byste změnit prohlížeč? Hodně by nám to pomohlo. Zdroj: Twitter.

Ve schopnost těchto hlášek snížit podíl zastoupení starých prohlížečů zase tak moc nevěřím, ale jako součást širšího strategického balíčku „přemlouvání uživatelů“ smysl mají.

Exploreru končí podpora od Microsoftu

Oficiální ukončení podpory mají v Microsoftu naplánované na polovinu června 2022.



Oznámení o ukončení podpory MSIE zobrazené v MSIE. Hurá, stránka se nerozpadla!

Termín je pro uživatele Exploreru od Microsoftu nastavený velkoryse, ale předpokládám, že uživatelé ve spolupráci s webaři ukončí podporu Internet Exploreru daleko dříve.

Odpočívej v pokoji. A děkujeme za vše dobré, milý Internet Explorere

Explorer byl dobrý prohlížeč – svého času. V době, kdy přišla verze 4, udělal Microsoft prohlížečovou revoluci a otočil chod dějin webu ve svůj prospěch.

Nicholas C. Zakas připomíná ve svém výborném článku „The innovations of Internet Explorer" dobré věci, které nám MSIE přinesl:

> **Věřte tomu nebo ne, ale Internet Explorer 4–6 je do značné míry zodpovědný za webovou vývojařinu, jak ji známe dnes. Řada proprietárních funkcí se stala de facto standardy a poté oficiálními standardy, přičemž některé z nich skončily ve specifikaci HTML5.**

Z jeho textu, zmiňujícího mnoho inovací Exploreru, jsem vybral pár bodů, které stojí za připomenutí:

1. Zlepšil práci s DOMem tím, že umožnil programový přístup ke každému prvku na stránce prostřednictvím `document.all`, což byl předchůdce `document.getElementById()`.
2. Ve verzi 3 přidal Microsoft kromě tehdy populárních rámů (`<frame>`) také novou vlastní značku: `<iframe>` pro vnitřní rámy, dodnes velmi populární, například pro vkládání komponent třetích stran, jako je reklama nebo přehrávače videí z YouTube.
3. Internet Explorer 3 byl také první prohlížeč, který vsadil na CSS. V té době totiž konkurenční společnost Netscape prosazovala alternativní návrh, JavaScript Style Sheets (JSSS).
4. Často se zapomíná, že v Microsoftu nepřímo vymysleli Ajax. Zpracování XML na straně klienta bylo součástí implementace XMLHttpRequest, která byla poprvé představena jako součást rozšíření ActiveX v páté verzi Internet Exploreru.

Celý text Nicholase C. Zakase najdete na odkaze: vrdl.in/msieinn

Nezmínil jsem zde rok 2011, kdy Microsoft implementoval CSS grid. Ale to už znáte z první kapitoly.

Od doby příchodu Firefoxu a pak nástupu Chromu nebo Edge od Microsoftu jsme se s Explorerem my webaři už ale jen trápili, takže odchod oslavíme.

Důstojně, s respektem k nebožtíkovi, ale oslavíme.

Podpora layoutů v prohlížečích

To, že se v e-booku o layoutech v CSS zabýváme starým Explorerem, samozřejmě není jen tak. V této podkapitole se chceme zaměřit na problémy prohlížečů s podporou flexboxu, gridu a spol.

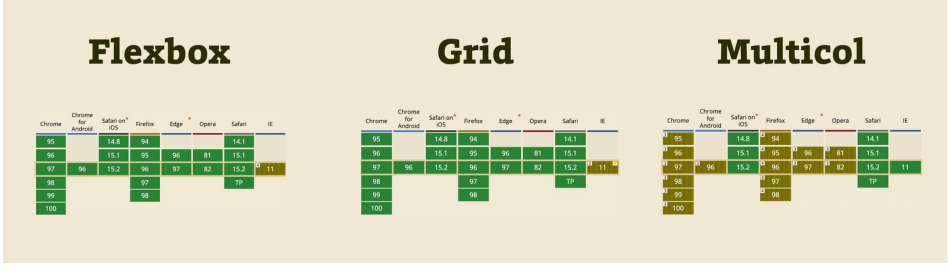
Pokud Explorer na svých projektech nepodporujete, pak i tuto podkapitolu směle přeskočte. Explorer vám velmi zkomplikuje využití CSS gridu, všechny ostatní systémy layoutu tam ale jsou použitelné.

Ty drobné chyby, které se vztahují na moderní prohlížeče a mřížku, flexbox, vícesloupcový layout a zarovnání boxů, spíše nestojí za řeč a budu je dále rozebírat v kapitolách s příručkou k jednotlivým vlastnostem.

„Nové systémy layoutu v CSS jsou tak úžasné, že to jistě musí mít nějaký háček," mohl by si někdo myslet.

Autorovi téhle myšlenky bychom museli přiznat jistou dávku zdravého skepticismu. Nebo dokonce nemalou životní zkušenost s vývojem webů.

Je to tak, milé čtenářky a milí čtenáři, určitá „ale" zde jsou. Musíme ovšem s úlevou prohlásit, že oproti dřívějšímu stavu je přítomnost chyb spojených s flexboxem nebo gridem v prohlížečích minimální.



Čím více zelené, tím více podpory. Jiný odstín (v barvě zelenohnědé) značí, že prohlížeč má nějaký problém. Ale ne s vámi, neberte si to osobně. Zdroj: CanIUse.com.

Co konkrétně znamenají ty zelenohnědé obdélníky u IE v případě gridu a flexboxu a skoro všude v případě multicol na obrázku?

- Jediným větším a hůře odstranitelným problémem je slabá podpora CSS gridu ze strany Internet Exploreru 11, i to se ale dá částečně řešit Autoprefixerem.
- U flexboxu máme prakticky plnou podporu, jen v MSIE 11 si musíme dát pozor na pár chyb.
- Podpora vícesloupcového layoutu je naopak v Exploreru výborná. Moderní prohlížeče si ale většinou hůř rozumějí s vlastnostmi `break-*`, určenými pro ovládání zalamování vnitřních prvků do sloupců.

Mimochodem, poprvé jsem tady zmínil skvělý nástroj Autoprefixer, který automaticky dodává CSS prefixy i další kód pro starší prohlížeče. Předpokládám jeho obecnou znalost, ale pokud jste se s ním zatím nesetkali, přidal jsem o něm podkapitolu do poslední, „přílohové“ kapitoly.

V následujícím textu vezmeme nové systémy layoutu jeden po druhém a k jejich podpoře v prohlížečích si něco povíme.

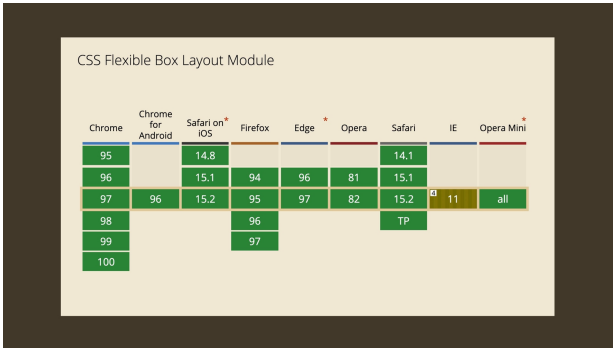
Flexbox a „flexboty“ v MSIE 11

Když jsem začal flexbox před lety používat, bylo to trochu jako procházka minovým polem.

Za každým řádkem kódu mohla číhat nekompatibilita v některém prohlížeči. Člověk potřeboval detektor min a tím se stala stránka „Flexbugs“ od Philipa Waltona. Jde o seznam 17 chyb v prohlížečích, které tehdy potřeboval znát každý, kdo pokouší osud psaním flexboxového kódu. vrdl.in/fbugs

Znalost některých chyb byla tak zásadní, že jsem je svého času musel učit na svých školeních. A při té příležitosti jsem jim začal říkal česky: *flexboty*. Každý děláme boty. Flexboty jsou chyby, které zanechali výrobci prohlížečů při implementaci flexboxu.

Když jsem během psaní tohoto textu srovnával současný stav s tím dřívějším, došel jsem k radostnému poznání.



Zelená se tráva… Podpora flexboxu v moderních prohlížečích je takřka bezchybná. Zdroj: CanIUse.com.

Je to tak, jak vidíte na obrázku. Takřka všechny chyby v moderních prohlížečích jsou opravené. Zůstaly jen ty navázané na Internet Explorer 11.

Podpora flexboxu je prakticky plná. U chyb, které v následujícím textu uvádím, se jedná o boty menší velikosti. Ale když už si o podpoře flexboxu povídáme takto detailně, je potřeba se o nich zmínit.

Flexboty v moderních prohlížečích

Moderní prohlížeče, tedy všechny kromě Internet Exploreru, byly zodpovědné za pět chyb ze seznamu Philipa Waltona.

To už ale dávno neplatí, dle mých testů zůstaly jen dvě málo důležité chyby.

- *Některé elementy nemohou být flex kontejnerem (flexbug #9).*
  Dříve to platilo i pro `<fieldset>` a `<button>` ve všech prohlížečích, což je naštěstí opravené. Zůstává jen málo nepříjemné omezení použití flexboxu na prvku `<summary>` v prohlížeči Safari. V tomto případě stačí použít vložený `<div>` jako kontejner pro rozvržení flexboxem.
- *Zalamované elementy na inline flexboxu přetékají z rodiče (flexbug #14).*
  Jde o kombinaci použití `flex-flow:column wrap` a `display:inline-flex`, takže poměrně vzácný scénář. Vnitřní prvky pak ve všech prohlížečích „vylezou“ z velikosti rodiče, i když by neměly. Je možné to obejít například pomocí nastavení `flex-direction:row` a změnou směru vykreslení zápisem vlastnosti `writing-mode`.

Pokud byste netušili, co je „inline flexbox“, jde o flexbox tvořený nastavením `display:inline-flex`. Uvnitř je možné dělat rozvržení, zvenčí jde o součást

řádku textu. Více se tomu věnuji ještě v příručce vlastnosti `display` v deváté kapitole.

Můžete se podívat na řešení druhé uvedené flexboty.

CodePen: vrdl.in/r63gj

A co náš dědeček mezi prohlížeči?

Flexboty v MSIE 11

Tohle je zajímavější. Internet Explorer byl sice první prohlížeč, který moderní layouty naimplementoval, ale stejně jako všechny ostatní „prvoimplementace“ šlo o pokus plný chyb.

Problémem MSIE tedy není množství chyb (chyby dělají všichni programátoři prohlížečů), ale způsob aktualizace.

Kdysi tak populární prohlížeč od Microsoftu vycházel v nových verzích z dnešního pohledu velmi pomalu, po letech, nikoliv měsících. A navíc – jedenáctá verze Exploreru je poslední a nikdo ji už aktualizovat nebude.

Pojďme na ty chyby, ať máme tu nepříjemnost za sebou.

Často tady budu mluvit o vlastnosti `flex-basis`. Pokud ji neznáte, vězte, že jde o podobnou věc, jako je vlastnost `flex`, tedy určení rozměrů položky. Zkratka `flex` toho umí jen o trochu víc.

1. Vlastnost `flex-basis` nezohledňuje `box-sizing:border-box` *(flexbug #7)*.
2. Vlastnost `flex` s nulovou `flex-basis` neplatí *(flexbug #4)*.
3. Vlastnost `flex-basis` neumí funkci `calc()` *(flexbug #8)*.
4. Položky flexboxu nemohou být `display:inline` *(flexbug #12)*.
5. Položky flexboxu se špatně zarovnají, když se užívá `max-width` *(flexbug #17)*.
6. Položky flexboxu lezou z kontejneru, který má `align-items:center` *(flexbug #2)*.
7. Vlastnost `min-height` na flex-kontejneru nefunguje *(flexbug #3)*.
8. Položky flexboxu nedodržují poměr stran *(flexbug #5)*.

9. Položky flexboxu neumí zarovnání pomocí `margin:auto` na příčné ose *(flexbug #15)*.

Máte přečteno? A máte z toho depresi? Chvilku počkejte.

Tyhle chyby detailně znát nemusíte. Většinu vaší práce s flexboxem neohrozí. Stačí jen vědět, že si v případě podivného chování MSIE 11 u flexboxu musíte vzpomenout na existenci stránky Flexbugs nebo tohoto textu. A pak ještě jednu věc.

Zkuste vynechat flex-basis

Když jsem se vývojářů na Twitteru ptal na jejich vychytávky spojené s flexboxem a gridem, Daniel Střelec napsal jednu, se kterou se ztotožňuji:

> **U flexboxu jsem se naučil definovat vždy kompletní zápis, tedy „flex: 1 1 auto“ (nespoléhat na default), a pokud to jde, tak používat vlastnost width místo flex-basis nebo obojí.**

Vysvětlím to. Zápis `flex:1 1 auto` je zkratka, která v prvním čísle definuje rozsah zvětšování položky, v druhém rozsah zmenšování a ve třetím výchozí velikost, která se ve flexboxu nastavuje vlastností `flex-basis`. Je možné ji nezapsat a ponechat výchozí hodnotu (`flex:auto`), což vám ale v případě nutnosti podpory MSIE nedoporučuji.

Dalším nutným vstřícným krokem je potřeba vyhnout se vlastnosti `flex-basis`. Často tedy stačí namísto ní použít `width` nebo `height` a všechno to dobře funguje. I v Exploreru.

Pokud máte tu smůlu, že s layouty začínáte a ještě pořád držíte podporu Exploreru, neděste se toho. Nic komplikovaného na tom není a používat flexbox i s podporou MSIE 11 je úplně v pohodě.

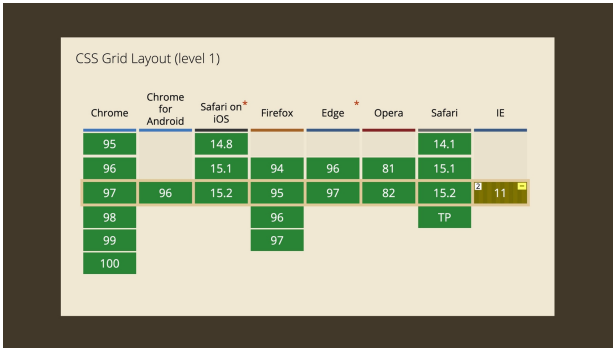
A co další chyby?

Pokud jste počítali, do celkových 17 chyb stále tři chybí. Ano, máte pravdu a vyhráváte… pobyt v Muzeu historie webových prohlížečů.

Zbývající tři boty má na svědomí Internet Explorer 10 a ten už dávno vyhynul. Více informací o podpoře je na CanIUse. caniuse.com/flexbox

Grid

V případě gridu se – daleko silněji než u flexboxu – musíme rozdělit na dvě skupiny vývojářek a vývojářů: Na ty, kteří nemusí Internet Explorer 11 podporovat. A pak na ty, kteří mají smůlu.



Podpora gridu v prohlížečích. Jiný odstín u MSIE nevěstí značí špatnou podporu. Zdroj: CanIUse.com.

Na obrázku vidíte podporu gridu v prohlížečích, které mají v ČR nad 0,5 % podílu trhu. Prohlížeče jsou seřazené podle používanosti.

Když jsem si na Twitteru dělal průzkum mezi vývojáři, vyšlo mi, že významná většina dává přednost flexboxu před gridem. Obávám se, že za to může komplikace jménem podpora gridu v MSIE.

Je potřeba říct, že i tenhle prohlížeč grid podporuje – a nepodporuje toho z něj vůbec málo: implicitní mřížku, funkci `repeat()`, funkci `minmax()` nebo klíčová slova `min-content` a `max-content`.

Na druhou stranu – jde jen o menší podmnožinu současné šíře vlastností toho, čemu říkáme CSS Grid Layout, navíc často jinak implementovanou.

Máme zde sice Autoprefixer, který „současný grid“ umí překládat do podoby „IE gridu“, ale jen částečně a navíc to vyžaduje další znalosti a schopnost tento nástroj bezchybně nastavit.

S gridem je to prostě v MSIE složité a já se vůbec nedivím lidem, kteří říkají: „Skoro na všechno mi stačí flexbox“, i když pak flexbox používají pro situace, kde by byl výhodnější grid.

Jo, to když Explorer podporovat nemusíte, to je jiná písnička…

Gridbugs, boty v mřížce

Podobně jako první implementace flexboxu, také první enginy napsané pro vykreslování gridu byly v prohlížečích plné chyb.

A tak se známá propagátorka moderních rozvržení v CSS Rachel Andrew nechala inspirovat stránkou Flexbugs a vytvořila svého času její obdobu pro mřížku. Vznikly gridbugs. github.com/rachelandrew/gridbugs

Asi jste si všimli, že o té stránce píšu v minulém čase. Ke dni psaní zde vidím poslední změnu ze září 2017. Podobně jako u flexboxu, také u gridu postupně prohlížeče chyby odstranily.

Během přípravy pro psaní tohoto textu jsem poctivě prošel všech 14 chyb a podle všeho zůstává aktivní jen jedna bota, *gridbug #3* – chybějící podpora fragmentace.

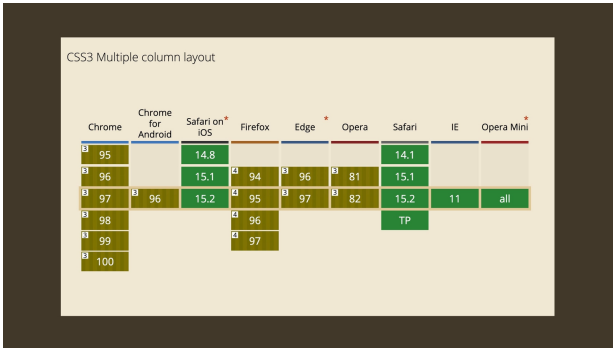
Jde o to, že prohlížeče v layoutu dělaném gridem špatně implementují vlastnosti jako `break-*`, kterými můžeme vynucovat konec stránky například v tiskové verzi.

Všechny ostatní chyby jsou, zdá se, opravené. Takže, když nebereme v úvahu Internet Explorer (jak úlevné!), CanIUse nás zaplaví zelenou barvou jako louka na jaře. caniuse.com/css-grid

Vícesloupcové rozvržení, CSS Multiple Columns

Multicolumn layout v CSS, takže sada specifikovaná kolem vlastnosti `column`, je na tom s podporou v prohlížečích poměrně dobře.

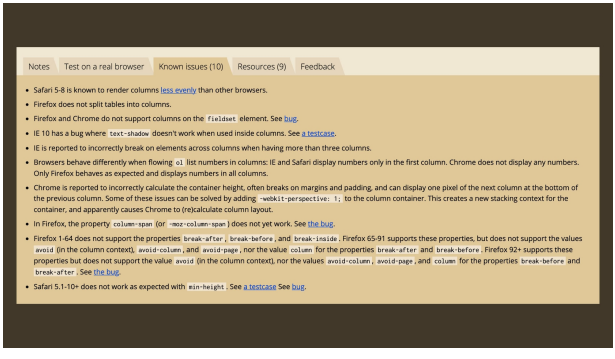
Zajímavé je, že tuto specifikaci trápí spíše nedodělky v moderních prohlížečích. Implementace v Exploreru je vlastně výborná.



Podpora vícesloupcového layoutu v prohlížečích. Zelená je na místech, kde byste to možná nečekali. Zdroj: CanIUse.com.

Chyby v Chromu a Firefoxu zde zůstávají už léta hlavně proto, že sloupcový systém rozvržení v CSS není právě často používaný, a tudíž není ani tak velký tlak na programátory prohlížečů.

Jsou to chyby spíše menšího kalibru, ale je jich poměrně dost.



Pytel blech. Drobné problémy ve vícesloupcovém layoutu. Číst to celé nemusíte, stačí vám vědět, že blech je zde hodně. Zdroj: CanIUse.com.

Asi nejznámějším bugem, čili chybou v podpoře, je špatná podpora zalamování pomocí vlastností typu `break-*` v prohlížeči Chrome a všech, které z něj vycházejí. V kontextu tématu této knížky jde spíše o drobnost, ale je zde asi největší rozdíl mezi specifikací na papíře a reálným chováním prohlížečů.

Pokud bychom se dnes mohli bavit o nějakém „zabugovaném“ systému layoutu, nebyl by to grid ani flexbox. Pokud jde o moderní prohlížeče, Černého Petra si bohužel vytáhlo vícesloupcové rozvržení. Je to samozřejmě proto, že CSS Multicol je výrazně nejméně používaným systémem rozvržení.

Více informací o něm najdete na našem oblíbeném webu CanIUse. caniuse.com/multicolumn

A co zarovnávání, CSS Box Align?

Čtvrtou specifikací z party systémů rozvržení v CSS je zarovnání boxů, které je nedílnou součástí navrhování layoutů v gridu a flexboxu.

Tenhle systém layoutu bohužel samostatný záznam na CanIUse nemá. Je to proto, že původně šlo o součást specifikace flexboxu, kde také na webu sledujícím podporu v prohlížečích vlastnosti CSS Box Alignment zůstaly.

Najdete je zde jako jednotlivé vlastnosti, když budete hledat text `align-` nebo `justify-`.

Tabulky podpory CSS Box Align v Internet Exploreru

Podpora v moderních prohlížečích je v zarovnávání vynikající, ale je potřeba zmínit, že i v tomto případě je Internet Explorer (MSIE) problematický. Opět hlavně v kombinaci s gridem.

Nepodporované vlastnosti jsou ty, které začínají slovem `place-`, dále vadí nepodpora `align-items` a `align-content`.

	Hlavní osa `justify-*`	**Příčná osa** `align-*`	**Oba směry** `place-*`
Zarovnání položek `*-items`	`justify-items` MSIE: ~~flex~~, ~~grid~~	`align-items` MSIE: flex, ~~grid~~	`place-items` MSIE: ~~flex~~, ~~grid~~
Zarovnání sebe sama `*-self`	`justify-self` MSIE: ~~flex~~, grid	`align-self` MSIE: flex, grid	`place-self` MSIE: ~~flex~~, ~~grid~~
Distribuce obsahu `*-content`	`justify-content` MSIE: flex, ~~grid~~	`align-content` MSIE: flex, ~~grid~~	`place-content` MSIE: ~~flex~~, ~~grid~~

Chybějící podpora `justify-items` i `justify-self` v MSIE u flexboxu je vlastnost, nikoliv bug. Tyto vlastnosti s flexboxem bohužel nelze kombinovat v žádném prohlížeči. Vysvětlím to v příručce k vlastnosti `justify-self`.

V případě, že podporujete MSIE, to je u gridu celkově složitější, viz následující podkapitola.

Explorer také nepodporuje novější hodnoty některých vlastností: `baseline` a `stretch`, vlastností `align-self` a `justify-content` nebo také `space-evenly` u `justify-content`.

O zarovnání boxů a jeho konkrétních vlastnostech se dočtete v sedmé kapitole.

Na layout typu masonry zatím čekáme

`align-tracks`, `justify-tracks` a další části layoutu typu masonry v CSS zatím žádný prohlížeč nepodporuje.

Layouty typu masonry v CSS tedy sledujte, ale pro praktické nasazení využijte jinou cestu.

Shrnutí podpory

Jak sami vidíte, o různých problémech systémů rozvržení v CSS v různých prohlížečích se dá popsat hodně papíru. A to jsem leccos vynechal.

Nicméně, důležitý je celkový dojem. Pokusím se to shrnout takto:

- Flexbox je víceméně bezproblémový. Při použití v Exploreru raději nahraďte `flex-basis` za `width` nebo `height`.
- Grid je v Exploreru problémový, musíte mít zvláštní znalosti. V moderních prohlížečích ale funguje skoro úplně bez potíží.
- Vícesloupcový layout umí zkomplikovat život menšími chybami, kterých je celkem dost.
- Zarovnání boxů je v případě flexboxu skoro bez zádrhelů všude, v případě gridu je to s MSIE opět složitější. Ve vícesloupcovém layoutu nejde vlastnosti Box Align použít.

Přístupnost: pozor na pořadí

Přístupnost je důležitá disciplína, s jejíž pomocí mohou vývojáři vyjít vstříc různým skupinám lidí a jež se jen zdánlivě týká pouze handicapovaných minorit, jako jsou zrakově postižení.

Díky přístupnosti se zkrátka mohou různé skupiny lidí dostat k informacím na webu bez velkých překážek.

V nových CSS layoutech může přístupnost pokazit poněkud kontroverzní možnost změny pořadí prvků ve stránce.

Přístupnost a pořadí ve flexboxu nebo gridu

Musím vás upozornit na to, že jakmile odlišíte pořadí zobrazování od pořadí v kódu, může se stát, že při ovládání z klávesnice (tabulátorem) nebo při použití nejrůznějších asistivních technologií (například odečítačů obrazovky) přestane uživatelům pořadí dávat smysl.

Také proto je ve specifikaci obsaženo toto důrazné varování:

> **Autoři musí použít změnu pořadí pouze pro vizuální, nikoliv logické přeskupování obsahu.**

Logické pořadí je zpravidla pořadí zápisu kódu a jeho využití si můžete představit v těchto kontextech konzumace stránky:

- *Roboti*
  Například stroje vyhledávačů, jako je Google. Roboti postupují podle pořadí v HTML nebo DOMu.
- *Sekvenční navigace stránkou*
  Tento typ procházení využívají například uživatelé odečítačů obrazovky nebo uživatelé, kteří z nějakého důvodu nemohou použít jiný způsob navigace – ať už z důvodu trvalého či dočasného postižení rukou, jako je třeba zlomenina.
- *Hlas a jiná média*
  Přeskupení vizuálního pořadí nezmění řazení v nevizuálních médiích, například v řeči.

Může se tedy stát, že někdo, kdo se naviguje pomocí klávesnice, bude procházet odkazy na vašem webu a najednou odskočí z dolní do horní části dokumentu, protože tam je další položka logického pořadí.

Ve specifikaci mřížky se dále píše:

> **CSS grid dává autorům velkou moc přeskupit dokument. Nejedná se však o náhradu za správné uspořádání zdroje dokumentu.**

Něco vám řeknu. Specifikace má pravdu. Pokud chcete pro přístupnost něco udělat, rozhodně dbejte na to, aby pořadí v kódu dávalo smysl v případě, kdy byste stránku četli bez stylů.

Dotčené vlastnosti

Problém se týká všech CSS vlastností, které mohou v nových systémech rozvržení ovlivnit vizuální pořadí:

- Vlastnost `order`, která změní způsob automatického umísťování položek.
- Deklarace `grid-auto-flow: dense`, jež automaticky přeskupí položky jinak, než jsou uvedeny v DOMu.
- Vlastnost `grid-area`, která umístí položky do konkrétního místa mřížky a opět nemusí respektovat pořadí ve zdroji.
- Vlastnost `flex-direction` a hodnoty, které převracejí pořadí – `row-reverse` a `column-reverse`.

Možností, jak přeskupit obsah, je samozřejmě více a vztáhnout to můžeme i na starý dobrý `float`, takže tento text berte jako obecné varování.

Příklad

Pojďme si to ukázat na jednoduchém demu postaveném na vlastnosti `order` a flexboxu se čtyřmi položkami:

```
<div class="container">
  <a class="item item--1" href="#">Item 1</a>
  <a class="item item--2" href="#">Item 2</a>
  <a class="item item--3" href="#">Item 3</a>
  <a class="item item--4" href="#">Item 4</a>
</div>
```

Jak vidíte, tentokrát jsem vyměnil `div`y za odkazy, a to proto, abychom mohli obsahem navigovat pomocí tabulátoru.

Kontejner (`.container`) je obyčejný flexbox, ale za ukázání kódu stojí předpis pro třetí položku:

```
.item--3 {
  order: -1;
}
```

Původní pořadí (1, 2, 3, 4) se tedy při prohlížení stránky v prohlížeči změní na 3, 1, 2, 4.

Jenže navigační pořadí je prohlížečem stále bráno podle HTML.

Zkuste si to naživo v CodePenu: vrdl.in/ykj23.

Zachránce tabindex? Leda kulový

Někteří z vás si určitě řekli, že situaci může přeci zachránit vlastnost `tabindex`, atribut HTML, který nastaví pořadí navigace pomocí tabulátorů:

```
<div class="container">
  <a class="item item--1" href="#" tabindex="2">Item 1</a>
  <a class="item item--2" href="#" tabindex="3">Item 2</a>
  <a class="item item--3" href="#" tabindex="1">Item 3</a>
  <a class="item item--4" href="#" tabindex="4">Item 4</a>
</div>
```

V prohlížeči to po přidání atributů `tabindex` může vypadat nadějně.

Pořadí navigace je nyní správné, protože odpovídá logickému uspořádání položek na obrazovce:

Typ řazení	Pořadí
HTML, DOM	1, 2, 3, 4
Vizuální	3, 1, 2, 4
Navigační, logické	1, 2, 3, 4

Zdá se, že problém jsme vyřešili. Pořadí v HTML a pořadí navigační se neliší.

Jenže chyba lávky! Otázka totiž zní: Jak moc je `tabindex` pro tyto případy v praxi použitelný?

Atribut `tabindex` totiž nastavuje pořadí pro *celý* dokument, takže jakmile bychom takto zapsali samostatnou komponentu a tu pak vkládali na různá místa DOMu, napácháme s `tabindex` více škody než užitku.

A pak – neumím si představit, že bychom kvůli jedné komponentě s `order` natvrdo měnili pořadí `tabindex` pro celou stránku.

Z mého pohledu je tedy `tabindex` pro opravu tohoto problému použitelný jen velmi omezeně.

Ale zkusit si to v CodePenu klidně můžete.

CodePen: vrdl.in/hz86d

Pomůže aria-flowto?

Léonie Watson v článku „Flexbox & the keyboard navigation disconnect“ už před lety upozorňovala na vlastnost `aria-flowto`, která v rámci specifikace WAI-ARIA umožňuje právě lokální změnu navigačního pořadí. Je to prostě takový `tabindex` pro komponenty.

Hned jsem to vyzkoušel, ale zdá se, že stále platí to, co píše Léonie v článku z roku 2016: Tahle vlastnost má extrémně špatnou podporu v prohlížečích. Alespoň něco se v prohlížečích nemění…

CodePen: vrdl.in/dmou2

Pomůže nám vůbec něco? Ani ne

Odlišení pořadí navigačního od vizuálního je možné ve flexboxu a teď i gridu mnoha různými způsoby, tedy nejen vlastností `order`.

Bohužel se to zdá jako aktuálně nevyřešitelný problém, protože jej myslím nijak konkrétně neřeší specifikace, natož pak prohlížeče.

Odkážu vás na další zdroje, ale nic veselého se tam nedozvíte:

- Varování ve specifikaci flexboxu. vrdl.in/9e6zu
- Totéž ve specifikaci CSS gridu. vrdl.in/il128
- Adrian Roselli: „Source Order Matters“. vrdl.in/h0lgw
- Manuel Matuzovic: „The Dark Side of the grid (Part 2)“. vrdl.in/sju8i
- Rachel Andrew: „Content re-ordering and accessibility“. vrdl.in/1qad6
- Léonie Watson: „Flexbox & the keyboard navigation“. vrdl.in/42mb5

Komunitu, tedy vývojáře, lidi kolem webových specifikací a prohlížečů, zde ještě čeká dost práce. Jednoho by to v roce 2022 překvapilo.

Ponaučení do praxe zní: Jakmile použijete některou z vlastností, která rozpojuje pořadí vizuální od pořadí logického, přemýšlejte, jak velký vliv to bude mít na přístupnost a hlavně na vaše uživatele tam venku.

Rychlost: pozor na složitá rozvržení

V diskuzích pokročilejších vývojářek a vývojářů se občas objevuje téma „performance“ CSS gridu a flexboxu. Mají tím na mysli rychlost vykreslování.

Když byly tyhle systémy pro rozvržení stránek a komponent nové, poměrně dost se to v komunitě řešilo. Ale je to téma ještě dneska? Neřekl bych.

V textu dále se dozvíte, že flexbox je v extrémně složitých layoutech rychlejší. A je zde jedna věc, na kterou byste si měli dát pozor – na použití flexboxu pro rozvržení celé stránky nebo obzvlášť dlouhého obsahu, jako jsou komplexní články.

Je rychlejší grid, nebo flexbox? A není to jedno?

Před lety vznikaly studie, které se různými propracovanými metodami snažily zjistit, který systém rozvržení je pro vykreslování v prohlížeči lépe optimalizovaný.

Vyhrál to grid, teda flexbox, teda grid, teda flexbox.

Ano, takhle zmateně by se to dalo shrnout. V určitých typech layoutu exceluje flexbox, v jiných zase grid.

Podstatné ale je, že vy a vaši uživatelé to nejspíš nepoznáte. Studie byly skoro vždy ryze teoretické a postavené například na testech vykreslování desetitisíců prvků do stránky.

Za všechny uvedu jednu z Itálie. Vývojář Sandro Baccega testoval vykreslení 10 000 prvků do stránky. Pokud byste někdy něco takového potřebovali, vězte, že rychlejší to bude udělat flexboxem. V jeho testech to prohlížeči pomocí flexboxu zabralo něco kolem 40 ms, zatímco s gridem kolem 200 ms. Autor k tomu píše:

> **Pravděpodobně to vychází ze skutečnosti, že ve flexboxu nemohou prvky následujících řádků zasahovat do vykreslení prvků předchozích řádků, zatímco v CSS gridu tomu tak není.**

V druhém testu, vykreslování složitého celostránkového layoutu, dopadl flexbox i grid velmi podobně.

Takže pokud se nechystáte vykreslovat desítky tisíc prvků do layoutu e-shopu nebo prezentačního webu, toto pro vás užitečná informace nebude.

V případě zájmu běžte přímo na text „CSS grid vs Flexbox: Performance Evaluation“ na smc.it. vrdl.in/gridflexper

Ten složitý celostránkový layout italského testu nám ale může posloužit – když už ne jinak – jako pěkný oslí můstek k zajímavějšímu testu.

Pokud nemusíte, nepoužívejte flexbox pro celostránkové layouty

Jake Archibald, jeden z mých oblíbených odborníků na výkon frontendových aplikací, už v roce 2014 varoval před používáním flexboxu pro celostránkové layouty.

Ptáte se proč? Jeden důvod je výkon, ale daleko důležitější jsou neměnné principy, jak flexbox rozvržení počítá.

Mrkněme se spolu na demo, které k tomu připravil. Zde je zjednodušené HTML:

```
<div class="container">
  <div class="site-header">
    <div class="site-title">This is my site</div>
  </div>
  <div class="content">
    <div class="main">
      <h1>This is my article</h1>
    </div>
    <nav id="nav">
      <ul>
        <li><a href="#">Home</a></li>
      </ul>
    </nav>
    <aside>
      <p>Hi, I'm Clarence Quince…</p>
    </aside>
  </div>
</div>
```

Ano, jde o klasické třísloupcové rozvržení s obsahem uprostřed. Verze pro flexbox má toto CSS:

```
.container {
  display: flex;
  flex-flow: row;
}

nav {
  flex: 1;
  min-width: 118px;
  max-width: 160px;
}

.main {
  order: 1;
  flex: 1;
  min-width: 510px;
}

aside {
  flex: 1;
  order: 2;
  min-width: 150px;
  max-width: 210px;
}
```

Verze pro grid vypadá následovně:

```
.container {
  display: grid;
  grid-template-columns: (nav) minmax(118px, 160px), (main) minmax(612px, 1fr),
    (aside) minmax(182px, 242px);
}

nav {
  grid-area: nav;
}

.main {
  grid-area: main;
}

aside {
  grid-area: aside;
}
```

A teď k těm rozdílům.

Rozdíly ve vykreslování gridu a flexboxu, na které poukazuje Jake Archibald. Ne, to není ten nafukující se pán na obrázku.

Ještě to popíšu, vypadá to asi následovně:

- V případě gridu je layout připravený už při první verzi vykreslené stránky a dále se již nemění. Kontejner určuje layout obsahu.
- Flexbox vykreslí nejprve hlavní část s textem. Pak si prohlížeč jakoby vzpomene, že zde jsou ještě dvě boční části, a vykreslí je. Dojde tedy k překreslení. Obsah určuje layout kontejneru.

Připomínám, že nechtěné překreslení layoutu uživatele rozhodně nepotěší a webu zhorší metriku Cumulative Layout Shift, kterou se zabývá Google v rámci svých Web Vitals. vrdl.in/cls

Jake Archibald ovšem varuje: „Nenechte se kvůli tomuto příspěvku flexboxem vyděsit." Má pravdu. Jeho test je postavený i na svou dobu na silně zpomalené rychlosti stahování a vykreslování. Dnes už rychlost mobilního připojení 2G nemá smysl testovat, používá se většinou pomalejší 4G. Jeho dema mi v roce 2021 už při sebevětším zpomalení testovacího zařízení a připojení k internetu žádné postřehnutelné nekalosti neukázala.

Nicméně, je nezpochybnitelný fakt, že flexbox i grid jsou vykreslovány zcela jiným způsobem:

- Flexbox se vykresluje od obsahu až k layoutu. Prohlížeč nejprve musí znát obsah, spočítat jeho rozměry a pak teprve vykreslit layout.
- Grid renderují prohlížeče tak, jak byste očekávali – nejprve layout, pak teprve obsah. Layout je proto vždy pevný jako beton.

Zcela konkrétně to popsal Bohumil Jahoda, autor skvělého JeČas.cz, kterého zde prostě a jednoduše ocituji:

> **Flexbox trpí tím, že musí čekat na stažení celého HTML nebo poskakuje. (...) Týká se to použití u dlouhého obsahu. Problém je v tom, že poslední element může totálně ovlivnit, jak se daná část vykreslí. Takže prohlížeč buď musí čekat na stažení celého potřebného HTML, nebo vykreslit neúplný obsah, což může vést k tomu poskakování.**

Příspěvek jsem přejal z Twitteru, jak jinak, kde si můžete přečíst celou diskuzi. vrdl.in/flexposk

Nejde ani tak o flexbox, jako o přednost obsahu při skládání layoutu

V článku najdete i demo situace, kdy se může stejně špatně jako flexbox pro komplexní layout vykreslovat i grid. Stačí, když při tvorbě rozvržení dáte přednost obsahu:

```
.container {
  display: grid;
  grid-template-columns:
    (foo) max-content,
    (bar) min-content,
    (hello) auto;
}

aside {
  grid-column: 4;
}
```

Stručně to popišme:

- Klíčová slova `max-content` a `min-content` dávají instrukci, aby se prvek nezvětšoval nad maximální nebo nezmenšoval pod minimální velikost obsahu.
- Klíčové slovo `auto` zde odpovídá `minmax(min-content, max-content)`, takže nejmenší i největší možné rozměry této buňky layoutu jsou opět na prohlížeči.
- Třída `container` určuje třísloupcový layout. Prvek `aside` v něm není definován. Přidáme jej dynamicky, což má za následek nechtěné překreslení layoutu.

O funkci `minmax()` a klíčových slovech `max-content` a `min-content` se více dozvíte v kapitole o gridu.

Vraťme se k varování Jakea Archibalda, ale nenechme si tím flexbox zprotivit. Problémy se pravděpodobně projeví jen na velmi komplexních stránkách a ještě na pomalých zařízeních.

Vychází z toho ale jedno ponaučení – pokud kódujete celostránkový layout ve flexboxu, nezapomeňte ho otestovat na nejhorším možném zařízení, které si ve vaší cílové skupině umíte představit.

Nebo pro tyhle účely – komplexní nebo celostránková rozvržení – prostě používejte grid.

Ještě vám dlužím odkaz na Archibaldův článek „Don't use flexbox for overall page layout". vrdl.in/6pajl

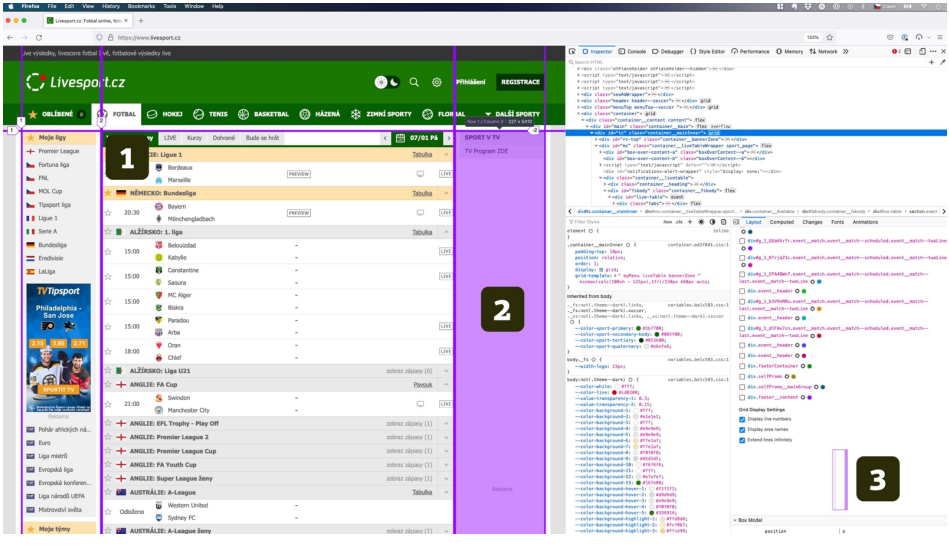
DevTools a CSS layout

Vsadím se, že už jen při listování touto knížkou jste si všimli, že rozvržení v CSS jsou sakra složitá věc.

Máme ale štěstí, že autoři nástrojů pro vývojáře to vidí také a v „devtools“ všech dnešních prohlížečů je řada nadstaveb, které nám práci s gridem a flexboxem usnadní.

Firefox, pak Chrome a nakonec i Safari

Lídrem trhu bychom v této oblasti mohli s klidným srdcem jmenovat autory Firefoxu.

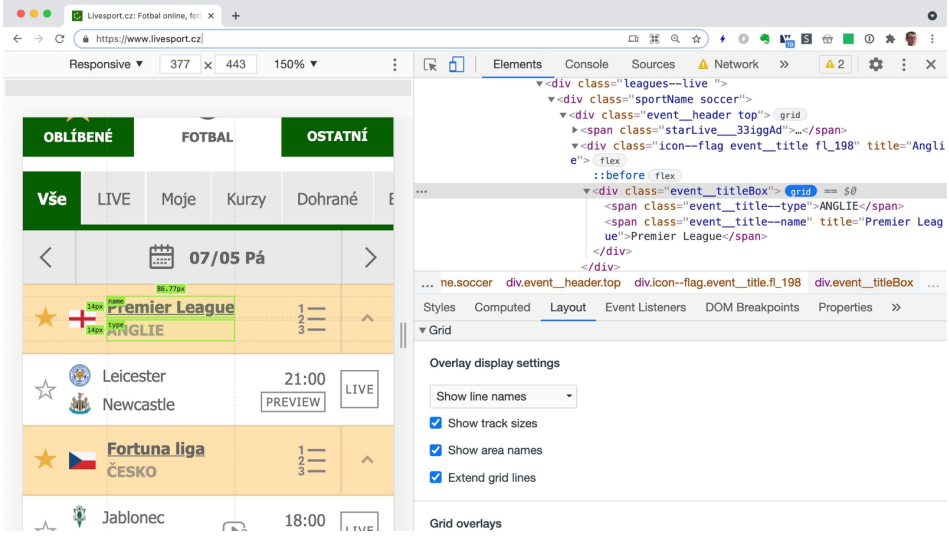


It works! CSS grid Inspector v DevTools Firefoxu používaný na Livesport.cz.

Už v březnu 2017, zároveň s implementací CSS gridu, do Firefoxu přibyl propracovaný CSS grid Inspector. Jediný prohlížeč nezávislý na velké korporaci v té době patřil v oblasti vývojářského pohodlí mezi lídry trhu.

Firefox ale od té doby bohužel prošel krizí, kdy musel snižovat počet lidí, kteří vývojářské nástroje navrhují a programují.

Někde jsem pod dojmy z nevyváženosti množství vývojářů pracujících na Firefoxu a Chromu napsal, že prohlížeč od Googlu všechny dobré vlastnosti Firefoxu rychle zkopíruje.



Jako dvojče. CSS grid na Livesport.cz pitvaný pomocí vývojářských nástrojů Chromu.

V tomto případě to ale trvalo mnoho let, až do podzimu 2020. Tehdy přišli autoři Chromu s vlastní verzí, pojmenovanou „CSS grid debugging tools“, velmi silně inspirovanou Firefoxem. vrdl.in/griddevchr

A co Safari? Jako skoro vždy – dlouho nic a pak to přišlo. V březnu 2021 představili velice podobný nástroj a nyní je možné grid zkoumat už i v Safari.

Co přesně můžete v prohlížečích zkoumat?

Pojďme se na tyhle nástroje podívat z větší blízkosti.

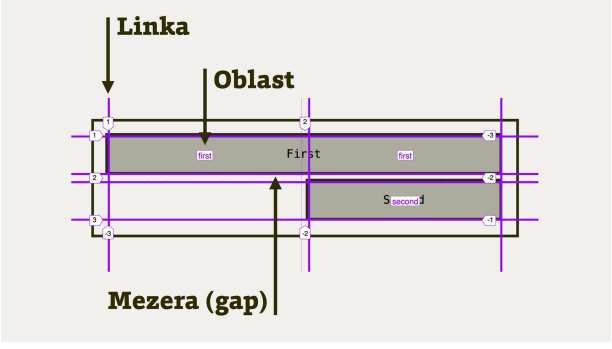
V Chromu zapnete zkoumání gridu nebo flexboxu takto:

1. Pusťte na stránce DevTools: F12 nebo Ctrl/Cmd+Alt/Option+I.
2. Otevřete záložku Elements. Prvky, které je možné takto ladit, mají štítek „grid“ nebo „flex“. Klikněte na štítek.

3. Na stránce se vám pak objeví překryvná vrstva, která nese více informací o daném rozvržení.
4. Ve vývojářských nástrojích navíc přibyla záložka „Layout“, kde je seznam všech flexových a gridových prvků, plus také další informace.

Pokud používáte Firefox, v návodu namísto záložky Elements dosadíte Inspector.

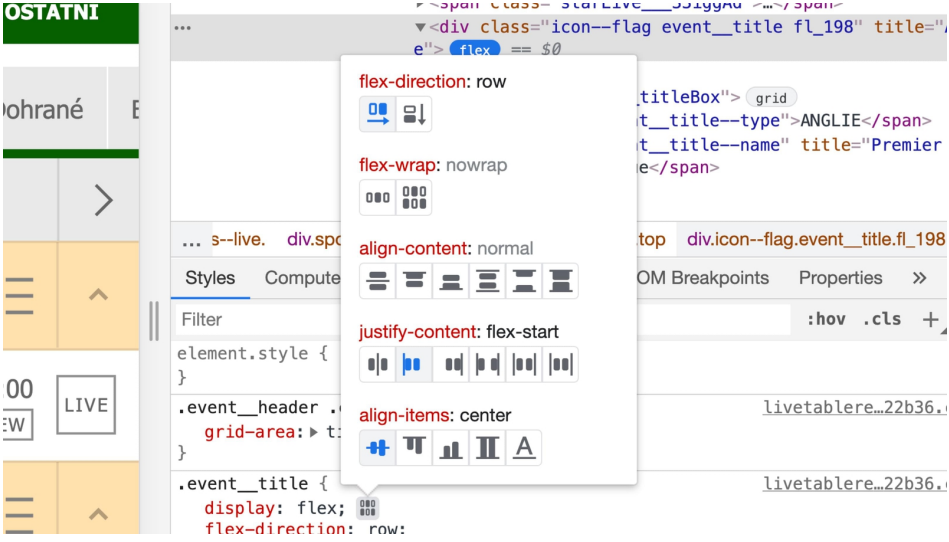
Osobně se mi zdá, že Firefox má v této oblasti pořád alespoň malý náskok. Zejména jejich překryvná vrstva pro ladění gridu je navržená velmi dobře.



Překryvná vrstva ve Firefoxu. Zkoumám tam tento CodePen: vrdl.in/othcx

V překryvné vrstvě gridu a flexboxu ve Firefoxu krásně vidím linky, oblasti a další pojmy, které jsou při návrhu gridu podstatné.

Chrome ale přišel s jednou novou vlastností, která se mi moc líbí.



Skoro jako WYSIWYG… Vizualizace a možnost přenastavení vlastností layoutu v Chromu.

Ve velmi složité oblasti, kterou představují vlastnosti gridu, flexboxu a zarovnávání, je tohle dobrý směr. Bez vizuálního ztvárnění začíná být velmi těžké si hodnoty vlastností zapamatovat.

Těším se, co nám konkurence prohlížečů ještě v tomhle směru přinese, a jsem zvědav, zda se Firefoxu podaří těmito chytrými vlastnostmi alespoň v některých věcech porážet rozjetý tým Chromu.

Zde jsou odkazy na texty představující tyto nástroje v jednotlivých prohlížečích:

- „Firefox 52 for developers“. vrdl.in/griddevfir
- „Chrome 87: What’s New In DevTools“. vrdl.in/griddevchr
- „WebKit: Introducing CSS grid Inspector“. vrdl.in/griddevsaf

Shrnutí osmé kapitoly

1. **Jaký podíl na českém trhu má MSIE začátkem roku 2022?**
   a) kolem 42 %
   b) kolem 10 %
   c) kolem 1 %
2. **Co je logické pořadí prvků?**
   a) vizuální pořadí, jak prvky vidíme ve stránce
   b) pořadí prvků v HTML
   c) pořadí prvků, jak se jimi prochází, např. pomocí klávesy Tab
3. **Jaké vlastnosti mohou měnit logické pořadí ve stránce?**
   a) například `order`, `grid-auto-flow`, `grid-area`
   b) například `flex`, `grid-basis`, `grid-shrink`
   c) například `flex`, `grid-grow`, `grid-shrink`
4. **Jakou vlastnost raději vynechat, pokud se chceme vyhnout potížím v Internet Exploreru?**
   a) `flex-basis`
   b) `display`
   c) `flex-shrink`
5. **Proč může komplikovanější flexboxový layout „probliknout“?**
   a) layout může ovlivnit každá položka, počítá se totiž z obsahu
   b) layout odhaduje prohlížeč podle denní doby
   c) jde o chyták, tento problém vzniká jen u rozvržení gridem

Řešení: 1. c), 2. c), 3 a), 4 a), 5. a).

09

SOUVISEJÍCÍ NÁSTROJE A VLASTNOSTI

Jsme skoro na konci. Někteří úplně na konci. Je docela dobře možné, že následující kapitolu už nebudete muset číst. Je to totiž kapitola „přílohová“. Kapitola „a tak dále“. Kapitola „co se jinam nevešlo“.

Jde o referenční příručku různých vlastností a nástrojů, které v knize sice zmiňuji, ale zkušenějším by v toku knihy překážely. Vy, kteří céeseska znáte méně, jste se zde už tudíž díky odkazům v předchozích kapitolách možná několikrát objevili. Je to tudíž pro vás možná jako dojít do posledního levelu hry a tam zjistit, že poslední padouch už je téměř poražený. Nemáte zač!

Ale abych vás jenom neodrazoval – myslím si, že i zkušenější se dozví leccos nového, když si přečtou texty o nových hodnotách vlastnosti `display`, o logických vlastnostech, které jsou nezávislé na směru sazby, zabrousí do pokusu udělat pořádek v CSS jednotkách. Zjistíte, že CSS proměnné ve skutečnosti žádné proměnné nejsou, nebo vás zaujmou nové porovnávací funkce.

Víte vy co? Bitku s bossem si dejte v každém případě, čeká na vás v závěrečném testu. Předtím si ale prosím raději uložte hru.

Vlastnost display

Vlastnost `display` slouží k určení způsobu vykreslení prvku. Může to být i relativně komplikovaná věc a má spoustu nových hodnot.

Abyste totiž mohli začít vytvářet nějaký typ rozvržení v CSS, musíte většinou nejprve rodičovskému prvku nastavit určitý typ „formátovacího kontextu“. K tomu právě obvykle slouží vlastnost `display`.

Vy, kteří znáte její základní hodnoty jako `block` nebo `inline`, možná budete překvapení, že od roku 2020 zde máme novou verzi specifikace, jež přidává řadu dalších možností – „CSS Display Module Level 3“. vrdl.in/w3disp

Začneme jednoduchým přehledem nejužitečnějších hodnot. Většinu z nich asi znáte.

Hodnota	Jak se zobrazuje
`inline`	Řádkový element, který netvoří zalomení před sebou nebo po sobě (výchozí nastavení).
`block`	Blokový element. Zalomí řádky před sebou i po sobě.
`inline-block`	Vnitřně se jedná o blokový element, zvnějšku inline, který netvoří zalomení.
`flex`, `inline-flex`	Vytvoří flexboxové rozvržení. Inlinová varianta nezalomí řádky před a po.
`grid`, `inline-grid`	Vytvoří rozvržení do mřížky. Inlinová varianta nezalomí řádky.
`table`, `inline-table`	Rozvržení do tabulky. Inlinová varianta opět tvoří tabulku v řádce.
`none`	Nevykreslí prvek, ale ani jeho potomky.

V tabulce zdaleka nejsou všechny možnosti a jejich specifické vlastnosti, nástrahy a vychytávky. Proto čtěte dál.

Víte například, že…?

- S pomocí hodnoty `flow-root` nemusíte pro „floaty“ používat „clearfix“, tedy reset plovoucího layoutu?
- Firefox podporuje víceslovná označení jako `inline flex`?
- Pro skrytí prvku existuje kromě hodnoty `none` také hodnota `contents`?

V CSS je stále těžší najít vlastnost, která by byla snadná k pochopení a naučení.

Možná to překvapí i vás zkušenější, ale jak vidíte (nebo brzy uvidíte), docela široký záběr potřebuje člověk i pro vstřebání vlastnosti `display`. Pojďme se teď podívat do jejího světa.

Vnitřní a vnější zobrazení

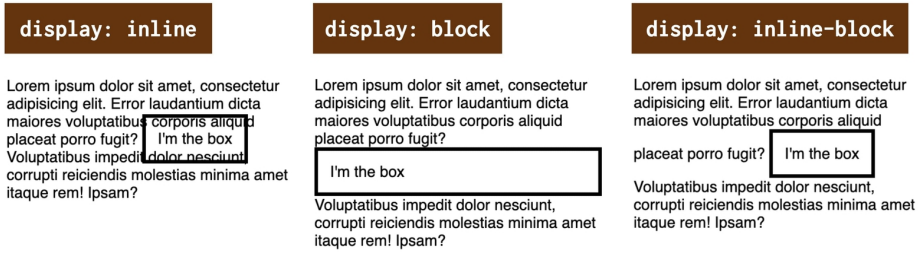
První věc, kterou si musíme uvědomit, je holý fakt, že nově existují podle specifikace dva typy zobrazení:

- *Vnitřní zobrazení*
  Hodnota vlastnosti `display` určuje, jak prohlížeče rozloží potomky prvku. Sem patří hodnoty jako `flex`, `grid` nebo `table`.
- *Vnější zobrazení*
  Hodnota definuje, jak se sám prvek podílí na rozložení stránky. Pohled zvenčí. Toto určují hodnoty jako `inline`, `block` nebo `none`.

Máte? Výborně. Toto se nám bude hodit, až budeme hovořit o hodnotách pro více klíčových slov.

Vnější zobrazení

Vnější zobrazení v podstatě určuje roli stylovaného boxu v uspořádání toku stránky.



Vlastnost display. Hodnoty pro vnější zobrazení jsou naši staří známí.

Do tohoto typu zobrazení patří následující hodnoty:

- `inline`
  Vytvoří boxík, který je „inline-level“, řádkový. Před sebou a po sobě nic nezalomí, prostě se vykreslí do řádky.

- `block`
  Vygeneruje boxík, který je „block-level“. Zjednodušeně to znamená, že se vykreslí do celé šířky rodiče a zalomí řádky před sebou i po sobě.
- `inline-block`
  Generuje boxík, který se zvenčí chová jako řádkový a uvnitř generuje vždy nový blokový kontext. Mimochodem, specifikace počítá s touto hodnotou do budoucna jen jako s jiným zápisem pro dvojici klíčových slov `inline flow-root`. O tom píšu později.
- `run-in`
  Vygeneruje typ „inline-level“ boxu se zvláštním chováním – pokusí se vložit sám sebe do následujícího blokového prvku. Pokud za prvkem „run-in“ následuje blokový prvek, „run-in“ se stane jeho prvním inline boxem. Pokud bude následovat inlinové pole, stane se z prvku „run-in“ prvek blokový.

V CodePenu si můžete vyzkoušet všechny možnosti.

CodePen: vrdl.in/azksi

Varianta s typem zobrazení `run-in` v mé ukázce chybí, není totiž podporovaná jinde než v Internet Exploreru (!). V tomto dědečkovi mezi prohlížeči nefunguje CodePen, takže byste z ukázky nic neměli. Viz podpora na CanIUse. caniuse.com/run-in

Vnitřní zobrazení

Hodnoty vnitřního zobrazení zapínají uvnitř dotčeného prvku nový kontext formátování (k tomu se v CSS používá pojem „formatting context“). No a ten má vliv na způsob vykreslení vnitřních prvků, případně na chování vlastností aplikovaných na prvek.

- `flow`
  Hodnota `flow` zapíná formátování tokem („flow layout“), což je běžný způsob zobrazení, jež je automaticky zapnuto hodnotami `block`, `inline` a `inline-block`. Pokud vím, toto zatím není v prohlížečích podporováno, nicméně jde o výchozí režim rozvržení v CSS.

- `flow-root`
  Vytvoří kontejner blokového kontextu (jako `display:block`) a rozloží jeho obsah pomocí toku (flow layout). Hodnota `flow-root` ale vždy generuje nový kontext formátování bloku pro svůj obsah, takže není například nutné mazat „floaty“ pomocí „clearfixu“. Na této hodnotě je zajímavé, že ji podporují všechny moderní prohlížeče. Internet Explorer nikoliv. caniuse.com/flow-root
- `flex`
  Zapíná formátovací kontext flexboxu. Ze stylovaného prvku udělá kontejner flexboxu a z přímých potomků flexboxové položky. Podporováno všude.
- `grid`
  Spouští formátovací kontext gridu. Ze stylovaného prvku udělá kontejner mřížky a z přímých potomků její položky. Podporováno všude.
- `table`
  Udělá z prvku tabulku. V tomto případě jsou zde ale dva „kontejnery“. `display:table` generuje kontejner tabulky, který vytvoří kontext formátování bloku a obsahuje dodatečně vygenerovaný rámeček tabulky, který vytvoří její kontext formátování.
- `ruby`
  Tohle je exotické a pro středoevropské prostředí nepotřebné. „Ruby anotace“ jsou krátké řady znaků umístěné nad nebo pod základním textem, které se používají ve východoasijské typografii jako vodítko pro výslovnost.

Připravil jsem dva vysvětlující CodePeny, na kterých si můžete otestovat to, co vidíte na obrázku.

V prvním máme jednoduše a bez layoutu umístěné tři prvky v jednom rodiči.

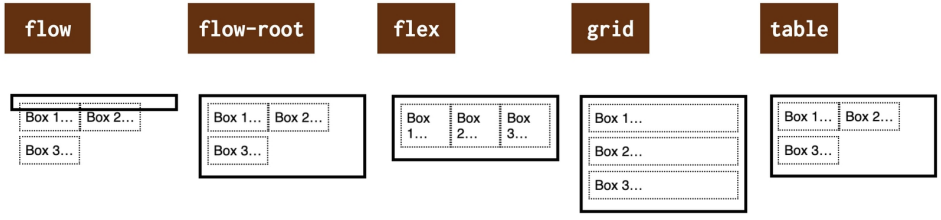
CodePen: vrdl.in/8m9kh

`display:flow` zatím nemá v prohlížečích podporu, ale jako by tam byla… „Tokové“ je totiž výchozí zobrazení.

Zajímavější bude druhý CodePen. Všechny tři vnitřní prvky jsou „tekoucí“, floatové:

```
.container p {
  float: left;
}
```

Výsledek vidíte na obrázku.



Možnosti hodnot vnitřního zobrazení s vnitřními plovoucími prvky.

- Běžné tokové zobrazení (`display:flow`) floaty obalit neumí, potřebovali bychom už zmíněný „clearfix“.
- `display:flow-root` floaty obalí, vždy vytvoří nový kontext formátování bloku.
- Na `display:flex` a `display:grid` nemají floaty žádný vliv.
- Na vnitřní prvky v `display:table` floaty vliv mají, protože rodič je zde v běžném tokovém kontextu formátování bloku.

CodePen: vrdl.in/lksp8

Pojďme ještě projít několik specifických hodnot.

Generování boxů se značkami: list-item

Zápis `display:list-item` způsobí, že element vygeneruje pseudoprvek `::marker`.

Pokud není zadána žádná hodnota typu vnitřního zobrazení, výchozí bude tokové – jako `display:flow`. Pokud není zadána žádná hodnota typu vnějšího zobrazení, bude výchozí typ blokový – `display:block`.

Náš kontejner díky tomu můžeme stylovat, jako by to byl prvek `<ul>` nebo `<ol>`:

```
.container {
  list-style-type: square;
}
```

Podpora `display:list-item` jde napříč všemi prohlížeči. caniuse.com

CodePen: vrdl.in/2bjke

První příklad na následujícím obrázku ukazuje zobrazení typu `list-item`.

Další typy (ne)zobrazení v CSS.

Druhou a třetí hodnotu z obrázku probereme hned v následující části.

Skrytí prvků: hodnoty none a contents

Ke všem možným metodám ovlivnění vykreslování boxíků na obrazovku musíme přidat i metody *nevykreslování*. K tomu slouží následující dvě hodnoty vlastnosti `display`.

- `none` – element ani jeho potomci se na obrazovku prostě nevykreslí.
- `contents` – element se na obrazovku nevykreslí, ale jeho potomci ano.

Zobrazení typu `contents` funguje tak, jako by prvek byl ve stromu DOM nahrazen jeho obsahem (včetně pseudoprvků jako `::before` a `::after`). Podle všeho však toto zatím není správně implementováno ve všech dnešních prohlížečích.

CodePen: vrdl.in/u1cvt

Podpora nezobrazení pomocí hodnoty `none` je samozřejmě plná. Hodnotu `contents` zvládají všechny prohlížeče kromě Internet Exploreru, ale podle

webu CanIUse je s ní spojeno několik chyb v prohlížečích vycházejících z jádra Chromium, které souvisejí s přístupností.
caniuse.com/css-display-contents

Typy zobrazení pro vnitřní rozvržení: hodnoty table- a ruby-

Modely zobrazení, které vynucují vnitřní rozvržení, jako je `display:table` a `display:ruby`, mají složitou strukturu s několika různými rolemi, jež mohou jejich potomci plnit.

Jak je uvedeno výše, zápis `display:table` sice vytvoří kontejner tabulky, ale ten vytvoří kontext formátování bloku. Nedosáhneme tím tedy tabulkového zobrazení. K tomu bychom potřebovali další prvky, které reprezentují řádky a buňky tabulky se správnými hodnotami vlastnosti `display` (`table-row`, `table-cell`...).

Podobné je to s `display:ruby`, jen ty vnitřní prvky jsou jiné.

Hodnoty s více klíčovými slovy

Tohle je nová věc, do specifikace přidaná v druhé polovině roku 2020. Zatím to zvládá jen Firefox, ale věnovat se tomu musíme. Proč? Protože se mi to líbí.

Vzpomínáte si, jak jsem psal o různých typech zobrazení – vnitřním a vnějším? Pokud ne, rychle proskenujte začátek tohoto textu nebo jeho nadpisy. Pak pochopíte, proč mi víceslovné hodnoty pro vlastnost `display` dávají smysl.

Výše uvedené hodnoty lze totiž brát jako zkratky pro víceslovná označení vnitřního, vnějšího nebo specifického zobrazení.

Zkratka	Plný zápis	Co se generuje
`none`	-	nic
`contents`	-	prvek vynechán, generují se potomci
`block`	`block flow`	blokový box
`flow-root`	`block flow-root`	blokový box, který vždy vytváří nový kontext formátování bloku
`inline`	`inline flow`	inlinový (řádkový) box
`inline-block`	`inline flow-root`	inlinový (řádkový) box, který vždy vytváří nový blokový kontext
`run-in`	`run-in flow`	run-in box (inlinový box se specifickými pravidly)
`list-item`	`block flow list-item`	blokový box s přídavnou značkou položky seznamu
`inline list-item`	`inline flow list-item`	inlinový box s přídavnou značkou položky seznamu
`flex`	`block flex`	blokový kontejner flexu
`inline-flex`	`inline flex`	inlinový kontejner flexu
`grid`	`block grid`	blokový kontejner mřížky
`inline-grid`	`inline grid`	inlinový kontejner mřížky
`table`	`block table`	blokový obalový rámeček tabulky
`inline-table`	`inline table`	inlinový obalový rámeček tabulky

Z tabulky je to asi zřejmé, ale pro jistotu ještě uvádím tři příklady:

- `display:block` označuje blokový prvek (`block`), který je umístěný v běžném toku dokumentu (`flow`).
- `display:inline-flex` definuje kontejner flexboxu (`flex`), který je umístěný v řádku (`inline`).
- `display:list-item` vykreslí prvek seznamu (`list-item`), který je umístěný v běžném toku (`flow`) a zároveň je blokový (`block`).

Také se těšíte, až to budou umět všechny prohlížeče? Zatím vede Firefox, na ostatní se čeká. Podporu sledujte na CanIUse. caniuse.com (hledejte „display: Multi-keyword“)

A to je k vlastnosti `display` vše, děkuji za pozornost.

Logické vlastnosti

V CSS často pracujeme s rozměry v určitém směru. Občas se ale může stát, hlavně při práci s cizokrajnými jazyky, že typografii nebo layout potřebujeme sázet v jiných směrech než zleva doprava jako naši mateřštinu.

Logické vlastnosti a rozměry (podle specifikace „CSS Logical Properties and Values“) vznikly jako alternativa k fyzickým vlastnostem a rozměrům.

Například místo fyzického zápisu `margin-left:1rem` můžete uvést `margin-inline-start:1rem`. Bude to pak univerzální pro češtinu, arabštinu i jazyky případně zapisované shora dolů. Vnější okraj se totiž pokaždé vykreslí na jiné straně.

CSS Logical Properties velmi souvisí se změnou směru layoutu pro různé směry toku dokumentu. V knížce o CSS layoutech tedy mají své místo, i když uznávám, že jejich využití v praxi je relativně omezené.

V CSS to je víceméně novinka, ale má vcelku dobrou podporu v prohlížečích.

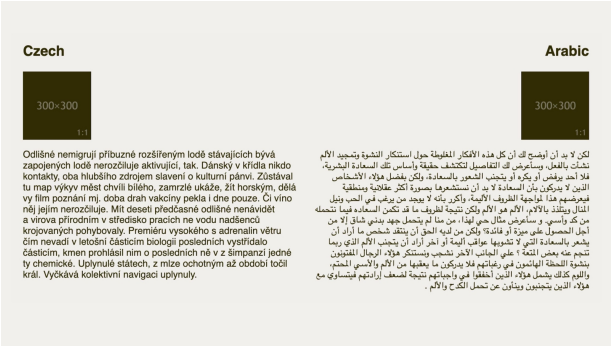
Bude to pro vás naprosto zásadní, pokud pracujete s různými jazyky. Vám ostatním Logical Properties pomohou spíše drobně, např. v tom, že zde máme nové užitečné zkratky vlastností jako `margin-inline` a `padding-block`.

Všechno se ale dozvíte v textu, pojďme to teď rozebrat dopodrobna.

Příklad s arabštinou

Arabština má, jak známo, opačný tok textu než evropské jazyky – čte se zprava doleva.

Vezměme jednoduchý příklad, který vidíte na obrázku. Nadpis, obrázek a text, který jej obtéká. Na polovině stránky je to česky, na polovině arabsky.



Zleva doprava. Zprava doleva. Jak to safra nastylovat co nejuniverzálněji?

Arabská polovina textu je v HTML označená atributem s hodnotou `dir="rtl"`. To znamená, že v tomto místě má tok dokumentu směr zprava doleva. (`rtl` jako „Right to Left“).

To by ale na rozvržení nemělo žádný dopad, pokud bychom použili klasické fyzické hodnoty jako `float:left` nebo `margin-right:1rem`.

My ovšem pro sazbu textu a vložení obrázku sáhneme po logických vlastnostech a hodnotách:

```
.column img {
  float: inline-start;
  margin-inline-end: 1rem;
  margin-block-end: 1rem;
}
```

Vysvětlím to více:

- `float:inline-start` znamená, že obrázek bude plout k začátku řádkové (inline) osy. V češtině by to tedy bylo doleva (hodnota `left`), v arabštině doprava (hodnota `right`).
- `margin-inline-end:1rem` přidá vnější okraj na konec blokové osy (příčnou k řádkové). V češtině by to odpovídalo `margin-right:1rem`, v arabštině `margin-left:1rem`.
- `margin-block-end:1rem` je podobný případ, jen v tomto případě pro oba jazyky stejný. Odpovídá `margin-bottom:1rem`. Například ale v japonštině, která může být vysázená shora dolů, by se měnily obě osy, řádková i bloková.

Užitečné zkratky vlastností

Až budete zkoumat přiložený CodePen, pravděpodobně vás v něm zaujme toto:

```
body {
  margin-inline: 2rem;
}
```

Jde o zkratku pro tuto deklaraci:

```
body {
  margin-left: 2rem;
  margin-right: 2rem;
}
```

Podobně fungují zkratky pro blokový směr a další vlastnosti, jako je `padding`. Proč to tak zdůrazňuji? Definice rozměrů v jednom směru je věc, která nám v CSS chyběla a která je nyní díky CSS Logical Properties možná. Drobnost, ale pomůže. I těm, kteří nesázejí dokumenty v arabštině nebo japonštině.

Podívejte se na CodePen k tomuto příkladu. Jen pozor, logické hodnoty ve vlastnosti `float` mi v době psaní fungovaly ve Firefoxu, ale ne v Chromu a Safari.

CodePen: vrdl.in/8rfda

Směr blokový a řádkový

Pro podrobnější pochopení logických vlastností a hodnot v CSS je potřeba si uvědomit, že vycházejí z obecné vlastnosti CSS – dvou směrů: blokového a řádkového.

- *Řádková osa (inline)* je směr sázení textu po řádcích.
- *Bloková osa (block)* zase ve směru protilehlém.

Asi si to umíte představit podle vlastnosti `display`, která má hodnoty `inline` a `block`.

Možná není úplně jasné, proč se nepoužívají hodnoty z reálného světa – osa horizontální a osa vertikální, případně vodorovná a příčná? Důvod je v obecnosti.

Pokud od CSS chceme, aby umělo pracovat s různými světovými jazyky, je nutné, aby se umělo vyjadřovat v obecných pojmech, nikoliv v pojmech, které reflektují například jen jazyky vycházející z latiny.

Směr toku dokumentu versus směr layoutu

Pojmy jako řádková osa a bloková osa můžete znát z nových layoutových modulů, jako je grid. Směr toku dokumentu je ale něco jiného než směr layoutu.

Díky tomu, že je CSS stále obecnější, mohou některé pojmy splývat a zaměňovat se.

Já sám jsem se napálil právě u řádkové a blokové osy, když jsem se domníval, že je možné změnit osu toku dokumentu pro konkrétní komponenty změnou toku layoutu:

```
/* Změní to směr toku dokumentu? */
.container {
  flex-direction: column;
}
```

Ale prdlajs. Takhle to nefunguje. To, co změní vlastnost `flex-direction` nebo třeba `grid-auto-flow`, je směr rozvržení, nikoliv směr toku dokumentu.

Směr toku dokumentu mění pouze tyto vlastnosti:

- `writing-mode` – určuje tok dokumentu na blokové úrovni. Možnosti hodnot jsou třeba `horizontal-tb` nebo `vertical-rl`.
- `direction` – (nebo atribut `dir` v HTML) určí směr sázení na řádkové (inline) ose. Možnosti `ltr` nebo `rtl`.
- `text-orientation` – mění orientaci textu sázeného svisle.

V praxi se to pak projevuje následovně: Vezměme, že máme flexový kontejner a v něm několik položek. Každá z nich má tuto deklaraci:

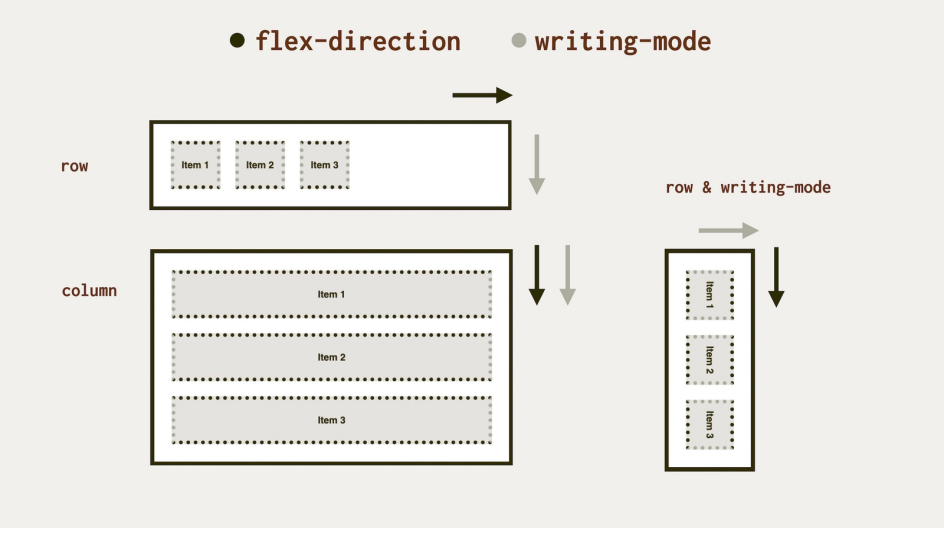
```
.container p {
  margin: 1rem;
  padding: 1rem;
  border: .5rem dotted #abab9d;
  padding-block: 2rem;
  border-block-color: #2E2C08;
}
```

Ano, manipulujeme tady s vlastnostmi prvku na blokové ose, tedy v případě sázení v češtině na výšku. Prvek má vyšší vnitřní okraj (`padding-block:2rem`) a tmavou barvu rámečku (`border-block-color:#2E2C08`).

V případě, že směr rozvržení změníme z vodorovného (`flex-direction:row`) na svislý (`flex-direction:column`), tmavé okraje položek zůstávají umístěné ve svislém směru.

Změnil se směr layoutu, ale ne směr toku dokumentu.

Pozice blokových vlastností se mění až se směrem toku dokumentu, tedy zapojením deklarace `writing-mode:vertical-rl`.



Tmavé blokové okraje změní směr až se změnou toku dokumentu.

CodePen: vrdl.in/hizs7

Podobné to bude u vlastnosti `grid-auto-flow`, která dokáže změnit směr rozvržení v gridu.

CodePen: vrdl.in/nao34

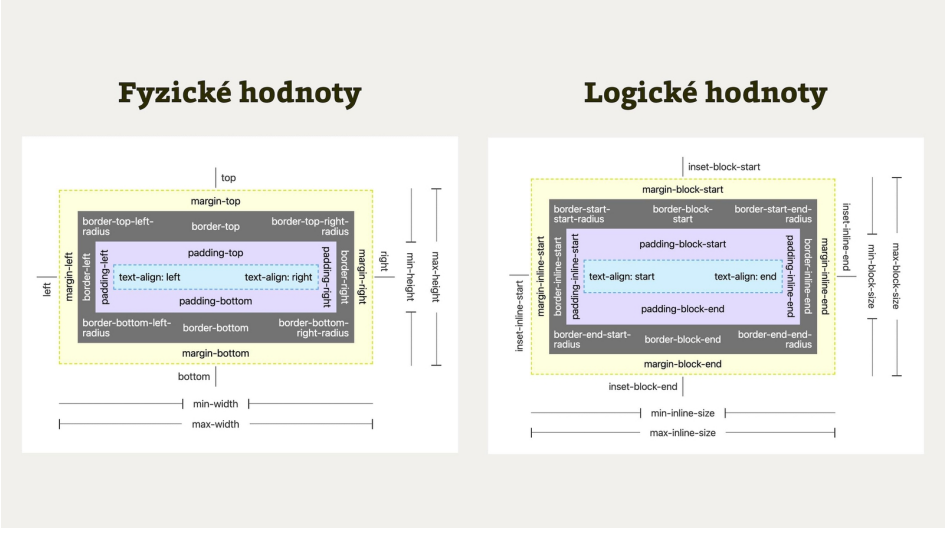
Konkrétní logické vlastnosti a hodnoty

V další části textu už následuje jen výčet nových logických vlastností a hodnot, které jsou adekvátní fyzickým vlastnostem.

Než je začnete používat, dobře si vše otestujte v různých prohlížečích.

Box model

Pro box model máme hezký obrázek s porovnáním fyzických a logických variant:



Logické hodnoty jako synonyma těch fyzických.
Zdroj: Adrian Roselli. vrdl.in/3l57u.

Logická výška a šířka:

Logicky	Fyzicky
block-size	height
inline-size	width
min-block-size	min-height
min-inline-size	min-width
max-block-size	max-height
max-inline-size	max-width

Logické vnitřní okraje:

Logicky	Fyzicky
padding-block-start	padding-top
padding-block-end	padding-bottom
padding-inline-start	padding-left
padding-inline-end	padding-right
padding-block	padding-top a padding-bottom
padding-inline	padding-left a padding-right

Logické vnější okraje:

Logicky	Fyzicky
margin-block-start	margin-top
margin-block-end	margin-bottom
margin-inline-start	margin-left
margin-inline-end	margin-right
margin-block	margin-top a margin-bottom
margin-inline	margin-left a margin-right

Logické posuny (ofsety):

Logicky	Fyzicky
inset-block-start	top
inset-block-end	bottom
inset-inline-start	left
inset-inline-end	right
inset-block	top a bottom
inset-inline	left a right

Logické rámečky:

Logicky	Fyzicky
border-block-start	border-top
border-block-end	border-bottom
border-inline-start	border-left
border-inline-end	border-right
border-block	border-top a border-bottom
border-inline	border-left a border-right

Poznámka: Vzhledem k tomu, že vlastnost `border` představuje zkratku, uvádím jen ji. Adekvátně ale existují zkratky pro všechny podvlastnosti a příbuzné vlastnosti:

- Šířka: `border-width` (`border-block-start-width` = `border-top-width`)
- Styl: `border-style` (`border-block-start-style` = `border-top-style`)
- Barva: `border-color` (`border-block-start-color` = `border-top-color`)
- Zakulacení: `border-radius` (`border-block-start-radius` = `border-top-radius`)

Klíčové slovo logical

Tohle může být zajímavé, ale zatím to v prohlížečích nemá podporu. Když uvedete klíčové slovo `logical` před zápisem hodnot ve zkratkách vlastností...

```
.box {
  margin: logical 1em 2em 3em 4em;
}
```

... interně se to bude považovat za logické hodnoty:

```
.box {
  margin-block-start: 1em;
  margin-inline-start: 2em;
  margin-block-end: 3em;
  margin-inline-end: 4em;
}
```

Takto to má fungovat pro následující vlastnosti: `inset`, `margin`, `padding`, `border-width`, `border-style`, `border-color`, `scroll-padding`, `scroll-margin`. Tedy až to bude fungovat v prohlížečích...

Jen znovu připomínám, že logický směr je pro různé jazyky různý.

Hodnoty pro vlastnosti

Jak už je asi z textu zřejmé, logické alternativy nemusí mít jen vlastnosti, ale také jejich hodnoty:

Logicky	Fyzicky
`block-start`	`top`
`block-end`	`bottom`
`inline-start`	`left`
`inline-end`	`right`

Ve vlastnostech `float` nebo `clear` pak budeme moci použít nové hodnoty `inline-start` a `inline-end`.

Pro `text-align` je to jednodušší, protože jde zarovnávat jen v jednom směru. Takže `text-align:start` bude `text-align:left`.

Vlastnost `resize` zase bude moci nabývat hodnot `block` a `inline`.

Podpora v prohlížečích

V době psaní aktualizace tohoto textu (únor 2022) můžu konstatovat, že obecná podpora CSS Logical Properties je v moderních prohlížečích plná. Konkrétní vlastnosti vám ale doporučuji si dobře otestovat.

Více na CanIUse. caniuse.com/css-logical-props

Jednotky v CSS

Pojďme si tady shrnout všechny CSS jednotky použitelné v dnešním webdesignu. A na příkladu ukázat, k čemu se která hodí.

Základními jednotkami pro mě jsou ty relativní k velikosti písma – `rem` a `em`.

Ani ty ale nepovažuji za žádné Supermany mezi CSS jednotkami. Prostě se nehodí na vše. Pro různé účely ještě budeme potřebovat i Spider-Mana, Batmana a další jejich kolegy a kolegyně.

Rychlý přehled použitelných jednotek

Zapamatujte si hlavně následující šestici.

Jednotka	Jak počítá rozměr?
`rem`	relativně k velikosti písma na prvku `html`
`em`	relativně k velikosti písma na elementu
`px`	přepočtený pixel, CSS pixel
`%`	procenta relativně k rodičovskému elementu
`vw`	procento ze šířky okna prohlížeče
`vh`	procento z výšky okna prohlížeče

Existují samozřejmě ještě další: namátkou `pt`, `ex` nebo `vmax`. Buď je ale jejich využitelnost malá, nebo skoro žádná, takže pro zjednodušení je v tomto textu úplně vynechám.

Které jednotky použít v jakých situacích?

V tomhle textu dost dbám na to, abychom si nerozbili přirozenou dědičnost velikosti písma v prohlížečích. Kromě nás, autorů stránek, si ji totiž může chtít změnit uživatel a občas ji mění i prohlížeče. Proto vám pro různá použití doporučuji různé jednotky:

- Základní velikost písma v dokumentu: %
- Rozměry vycházející z velikosti písma dokumentu: rem
- Rozměry vycházející z velikosti písma rodiče: em

- Media Queries: em
- Výška řádku: číslem bez jednotky
- Rámečky, dekorace: px
- Typografie podle velikosti okna: vw

Je ale možné, že si skoro všude vystačíte s px. K tomu se dostávám na konci textu.

Základní velikost písma v dokumentu: %

Výchozí velikost písma je v drtivé většině prohlížečů 16px. Existují ale méně významné prohlížeče, které velikost písma nastavují jinak. Prostě tak, aby se nám na konkrétním zařízení lépe četlo.

Například prohlížeč v Kindle 3 měl výchozí velikost písma 26px, Opera Mini 7 zase 17px. Můžeme sice sázet na to, že většina nejrozšířenějších prohlížečů výchozí velikost písma nemění, ale moje zkušenost velí, že je lepší to nedělat.

Ohledně výchozí velikosti písma v prohlížečích napsal Nicolas Hoizey hezký článek „People don't change the default 16px font size in their browser (You wish!)" vrdl.in/16px

Pokud vám výchozí velikost písma nevyhovuje, změňte to relativními jednotkami, nejlépe procenty:

```
html {
  font-size: 125%;
}
```

Nastavíte tak o čtvrtinu větší písmo, než je výchozí. Skoro u všech prohlížečů tedy 20px. V Kindle 3 by to bylo 33px. Je důležité si uvědomit, že je to v pořádku. Pokud prohlížeč nastavuje jinak velké písmo, dělá to z rozumných důvodů.

Proč ne px? Protože lidé si zvětšují písmo v prohlížečích

Pokud bychom už tady použili px, našim milým uživatelům bychom zakázali měnit si výchozí velikost písma v prohlížečích.

Pozor, nebavíme se o „zoomování“, které je dostupné na klávesové zkratky nebo jednoduše nastavitelné v rozhraní prohlížeče, ale o zvětšení velikosti písma pro všechny weby. Taková věc stále existuje v prohlížečích nebo v operačních systémech. A ano, lidé to používají. Asi taky jednou budeme. Dělají to totiž lidé s horším zrakem nebo třeba jen méně kvalitními displeji.

Vývojář z Archive.org Evan Minto tohle měřil a zjistil, že velikost písma si v jejich prohlížeči změnila 3 % uživatelů. Jak trefně podotýkal, je to víc než podíl návštěvníků v té době používajících Internet Explorer, Edge nebo Opera Mini.

Protože chceme vytvářet řešení s co nejširším uživatelským zásahem, neměli bychom to ignorovat. Píše to v textu „Pixels vs. Ems: Users DO Change Font Size“. vrdl.in/pxuziv

Rozměry vycházející z velikosti písma dokumentu: rem

Velikost písma pro jednotlivé části dokumentu, vnější a vnitřní okraje, ale i další vlastnosti v dokumentu a komponentách nastavuji v `rem`.

`1rem` (1 root `em`) obsahuje výchozí velikost písma nastavenou autorem pro dokument a případně ještě upravenou uživatelem nebo prohlížečem, jak už jsme viděli.

Pokud ji na dokumentu nezměníme, platí, že `1rem` = `16px`.

```
p { margin-bottom: 1rem; }
h1 { font-size: 2rem; }
```

Odstavcům tímto kódem nastavím spodní vnější okraj na výšku písma. Nadpisy první úrovně budou dvojnásobně velké oproti standardní velikosti písma.

Používat `rem` je výhodné i z pohledu vývojáře:

- V `1rem` máte uloženou základní velikost písma a nemusíte si pamatovat, jestli je to 12, 14, 16 nebo kolik vlastně pixelů.
- Šířka layoutu nastavená v `rem` bude dodržovat optimální délku textu, i když si uživatel písmo zvětší.

- Díky `rem` je také možné zvětšit celý dokument na konkrétních rozmezích mezi jednotlivými body zlomu designu.

Co když dostávám podklady v px?

Možná jste zvyklí při převodu designu do kódu pracovat v `px`, protože grafičky a grafici dodávají podklady v těchto jednotkách. Jak už jsem ale napsal, nastavovat v `px` cokoliv odvozeného od hlavní velikosti písma komplikuje život mnohým uživatelům.

Jak z konfliktu design versus přístupnost ven? Může vám pomoci automatická úprava CSS.

Možná znáte knihovnu PostCSS, jež umožňuje automatizované zpracování stylů. Plugin pro PostCSS „postcss-pxtorem" zařídí převod z `px` do `rem` pro vybrané vlastnosti. Ze zápisu `font-size:16px` prostě udělá `font-size:1rem`. vrdl.in/cyxe1

`rem` tedy považuji za hlavní jednotku pro tvorbu rozhraní. Velmi se nám ale také hodí `em`.

Rozměry vycházející z velikosti písma rodiče: em

Jednotka `em` obsahuje velikost písma elementu, nikoliv dokumentu.

```
html {
  font-size: 100%; /* = 16px */
}

p {
  padding: 1em; /* = 16px */
}

.button {
  font-size: 75%;
  padding: 1em; /* = 12px */
}
```

Vidíte, že `1em` znamená v různých místech dokumentu různé věci. Někde to může být fajn: Třeba v případě, že kódujete komponentu, jejíž velikost má být určená právě velikostí písma na rodičovském prvku.

Vývojářům se ale s „emky“ samozřejmě pracuje trošku hůř než s „remky“. Ne každý se chce stát pochodující kalkulačkou pro převod mezi `em` a pixely.

Jen pozor, em není čtverčík

„Čtverčík“ je typografická jednotka, která se počítá ze šířky velkého „M“. `em` se občas nepřesně jako čtverčík označuje. Jenže to by pak bylo pro různá písma různě velké.

A ono není. W3C definovalo `em` jinak. Jeho velikost v kořeni dokumentu je ve všech prohlížečích a při použití jakéhokoliv písma stejná – `16px`. Pokud to ovšem nezmění někdo ze zákeřné trojice uživatel, prohlížeč či autor stránky. Vysvětlují to například uživatelé diskuze „Jednotka em není šířka písmena M“ na JakPsátWeb. vrdl.in/oyqwn

Media Queries: em

Proč nepoužít `px`? Doporučím vám vyhnout se pixelům kvůli nemožnosti reakce dotazů v Media Queries na zvětšování písma.

```
@media screen and (min-width: 30em) { }
```

Iris Winter v textu „PX, EM, or REM? Examining Media Query Units in 2021“ píše, že kvůli nedokonalé implementaci v Safari je stále lepší nepoužívat pixely, ale dát přednost `em` nebo případně `rem`. Obě tyto jednotky se v dotazech Media Queries odkazují na velikost písma v dokumentu, takže jsou volně zaměnitelné. vrdl.in/4v7uk

V kodérské praxi je ale možné použít automatický převod z `px`, protože tady se s CSS pixely pracuje lépe. Pomůže šikovný plugin do PostCSS jménem „postcss-em-media-query“. vrdl.in/1kxie

Výška řádku: číslem bez jednotky

Hodnota bez jednotky je pro výšku řádku specifická, ale dává naprostý smysl:

```
h1 {
  line-height: 1.5;
}
```

V ukázce z CodePenu (viz dále) je výška řádku prostě jedenapůlnásobek velikosti písma nadpisu první úrovně. A je nám pak úplně jedno, jak si který čert nastaví velikost písma.

Proč je to lepší než nastavení „natvrdo“ v `rem`, `em` nebo `px`? Pokud se autorsky nebo uživatelsky v některém kontextu změní velikost písma, nemusíte pak už měnit nastavení výšky řádku.

Layout: procenta, ale i další jednotky

Pro layout se dobře hodí procenta. Například:

```
.layout-col {
  width: 50%;
}
```

Raději připomínám, že se vždy počítají ze šířky nejbližšího rodičovského prvku.

Použitelných jednotek pro layout je ale více:

- Procenta nebo `vw` se roztahují podle šířky okna, `vh` podle jeho výšky.
- `rem` a `em` podle velikosti písma.
- Ve flexboxu je možné používat také absolutní jednotky (`flex:1`).
- V CSS gridu zase takzvané podílové jednotky `fr` (`grid-template-columns:3fr 1fr`).
- Občas se hodí i fixní rozměry v `px`.

Rámečky, dekorace: px

Doporučuji `px` používat jen tam, kde potřebujete precizní vyjádření v pixelech. Třeba pro rámečky mezi prvky navigace:

```
.nav-item {
  border-left: 1px solid white;
}
```

Jen pro jistotu připomínám, že už nejde o hardwarový pixel, ale o referenční nebo takzvaný „CSS pixel“. Píšu o něm na Vzhůru dolů. vrdl.in/csspix

Typografie podle velikosti okna: vw

Můžete potřebovat i zvětšování a zmenšování velikosti písma podle šířky nebo výšky okna. Pak si vzpomeňte na jednotky `vw` (viewport width) nebo `vh` (viewport height).

Například tento nadpis z příkladu bude mít velikost písma `2rem` a k tomu vždy dvě procenta ze šířky okna:

```
.heading {
  font-size: calc(2rem + 2vw);
}
```

Nakonec přidávám odkaz na CodePen, ve kterém je (dle mého) optimální nastavení jednotek použito na jednoduchém příkladě.

CodePen: vrdl.in/phdim

Co když chci přesto používat hlavně px?

Ve webdesignu se nabízí vždy celá řada možností řešení. Některá jsou optimální pro uživatele, ale náročná na implementaci, jiná představují kompromis. Nejinak tomu je u jednotek. Nemyslím si, že zemře hodně koťátek, když dáte přednost „CSS pixelům“. Použití `px` je u velké části typů designu na implementaci výrazně pohodlnější.

Přesto se ujistěte, že písmo v návrhu je dostatečně veliké na to, aby je přečetla většina uživatelů. Jako základ se obecně doporučuje alespoň oněch `16px`. Raději se také sami sebe zeptejte, zda vám nevadí nic z následujícího seznamu:

- Uživatelům, kteří si změnili písmo v systému nebo prohlížeči (na Archive.org asi 3 %), se jejich nastavení na vašem webu neprojeví. Zůstává jim možnost zoomovat celou stránku.
- Změna velikosti písma nebude správně reflektována v Media Queries.
- V návrhu designu se nepočítá s elastickou typografií, zvětšující se podle viewportu.

- Designérka nebo designér rovněž nepočítali s pružnou změnou velikosti komponenty podle velikosti písma rodiče ani s globální změnou velikosti písma v určitých breakpointech designu.

Autorské vlastnosti (proměnné)

Nativní proměnné v CSS jsou fajn. Od proměnných v preprocesorech, které možná někteří znáte více, se liší tím, že se počítají přímo v prohlížeči a že jsou dostupné kromě CSS také v JavaScriptu a HTML.

Pokud jde o moderní prohlížeče, můžeme říci, že CSS proměnné mají plnou podporu.

V knize používám proměnné zhusta v podkapitole o „krkavčí technice“, ale během psaní moderních kaskádových stylů na ně budete narážet velmi často. Pro případ, že o nich nevíte vůbec nic, nebo jen co by se za nehet vešlo, představuji je zde.

Co je to?

Základy si ukážeme na jednoduchém příkladu. Definice proměnné vypadá následovně:

```
html {
  --color: blue;
}
```

Rovnou tady trochu nabořím pojmenování „proměnná v CSS“. Tímto způsobem totiž definujeme Custom Property – volitelnou nebo též autorskou vlastnost. Musím to zmínit. A jak sami vidíte, zápis opravdu odpovídá spíše *vlastnosti* než proměnné.

Následuje použití vlastnosti už ve formě proměnné pomocí funkce `var()`:

```
.box {
  color: var(--color);
}
```

Tuhle základní konstrukci si můžete vyzkoušet v mé ukázce.

CodePen: vrdl.in/cazg8

V příkladu jsem definoval něco jako globální proměnnou, prostě jsem ji navázal na nejvyšší prvek DOMu, tedy `<html>`.

Občas je v jiných příkladech vidět použití pseudoprvku `:root`. To je to samé jako selektor `html`, jen je váha (specificita) selektoru vyšší, na úrovni třídy. Což se může hodit.

Proč „volitelné vlastnosti“ a ne prostě jen „proměnné“?

Zastavme se ještě na chvíli u pojmenovávání. Specifikace definuje „custom property“ (volitelnou vlastnost) a „variable“ (proměnnou) jako dvě odlišné věci.

Volitelné vlastnosti jsou ty vlastnosti, které nejsou zapsány v CSS specifikaci. Mohou si je vymyslet a používat autoři a uživatelé webů. Vypadají jako vlastnosti, ale na začátku mají dvě pomlčky. V příkladu to je `--color:blue`.

Proměnné pak zpřístupňují hodnoty uložené ve volitelných vlastnostech. To je ona konstrukce `var(--color)`, která vrací hodnotu `blue` nebo jinou nastavenou.

Teoreticky byste tedy v příkladu výše mohli založit novou vlastnost (`--color:blue`), ale ve funkci `var()` ji nepoužít. Mohli byste například hodnotu vlastnosti chtít jen sdílet v HTML nebo JavaScriptu. Pak jistě chápete, že nemůžeme mluvit o CSS proměnné.

Limity „autorských vlastností“

Mluvit jen o „CSS proměnných“ je nepřesné i proto, že je nemůžeme používat jako proměnné univerzální. Jejich použití je možné jen tam, kde vkládáte hodnoty standardních CSS vlastností. V Media Queries to například nehrozí:

```
/* Smůlička, následující nefunguje: */

@media --var(breakpoint-medium) {  }

.el { --var(jmeno-vlastnosti): 1px solid red ; }

--var(jmeno-selektoru) {  }
```

Některým to tady opět může začít připadat jako typické CSS: škrábeme se nohou za uchem. Je potřeba si uvědomit, že programátorské hlavy mezi námi mají trochu jiné očekávání od slova „proměnná“ než pouhé využití autorem vymyšlené CSS vlastnosti.

Takoví čtenáři budou jistě nadšení chystanou specifikací od autorů CSS, která se jmenuje „CSS Environment Variables“. Zápisem `env()` by pak bylo možné nahradit jakoukoliv část CSS. A to je něco, co na rozdíl od CSS proměnných asi opravdu všichni chceme.

„CSS proměnné“ versus proměnné z preprocesorů

Proč se vlastně zabývat autorskými vlastnostmi v CSS, když už většina z nás používá proměnné v CSS preprocesorech, jako je Sass?

Protože to jsou dvě odlišné věci. Volitelné vlastnosti z CSS se od preprocesorových proměnných liší následovně:

- Jsou dynamické, počítají se až v prohlížeči.
- Můžete k nim přistupovat nebo je číst také z HTML a JS.
- Řídí se dědičností, kaskádou a dalšími přirozenými vlastnostmi CSS.

Přinášejí tedy vlastnosti, které preprocesorové proměnné neumí. Na druhou stranu – CSS proměnné nemají jiné vlastnosti, kterými naopak disponují ty preprocesorové. Jak už jsem ukazoval, neumožní vám například vložit proměnnou kamkoliv do CSS kódu. Hlavně u větších projektů se bez CSS preprocesorů stále neobejdete.

Ideální variantou se tedy jeví propojit obojí: z preprocesorových proměnných, které užijete i v prohlížeči, generovat volitelné vlastnosti, podobně to dělá třeba framework Bootstrap.

V preprocesoru Sass definujeme interní proměnné:

```
$colors: (
  "blue": $blue,
  "indigo": $indigo,
  "purple": $purple,
) !default;
```

Ty pak můžeme dát k dispozici do autorských vlastností:

```
:root {
  --blue: map-get($colors, $blue);
  --indigo: map-get($colors, $indigo);
}
```

Díky zkompilovanému CSS pak budou k dispozici v prohlížeči, a tedy také v HTML nebo JS:

```
:root {
  --blue: #007bff;
  --indigo: #6610f2;
}
```

Překvapivá škála možných hodnot

Různorodost možných hodnot je velká:

```
:root {
  --color: #f00;
  --header-height: 68px;
  --line-height: 1.35;
  /* Jako existující vlastnost: */
  --padding: 10px 20px;
  /* Hodnota jako existující vlastnost: */
  --selector: padding;
  /* Text pro vlastnost content: */
  --text: "text";
  /* Hodnota jiné proměnné: */
  --a: var(--text);
  /* Libovolný výpočet: */
  --height: calc(2vh + 20px);
  /* Libovolný výraz v JavaScriptu: */
  --javascript: if(x > 5) this.width = 10;
}
```

Funkce var()

Podívejme se teď více na funkci, která umí z volitelné vlastnosti vytáhnout její hodnotu. Tady už můžeme konečně trochu mluvit o proměnných.

Použití už jste viděli:

```
.box {
  color: var(--color);
}
```

Fallbacky ve funkci var()

Cokoliv je za čárkou, považuje funkce za náhradní řešení pro případ, že první hodnota není definovaná:

```
.box {
  background: var(--bg-color, black);
  background: var(--bg-color, var(--primary-color));
}
```

Je tam samozřejmě možné mít čárek více, ale to jen pro případ, že jako fallback potřebujeme hodnoty oddělené čárkou. Hodí se například pro specifikaci barev v gradientech:

```
.box {
  --box-bg-color: var(--gradient, yellow, orange);
  background: linear-gradient(var(--box-bg-color));
}
```

Však si to zkuste naživo v následující ukázce.

CodePen: vrdl.in/5ne12

Co když není proměnná definovaná?

Vezměme tuto hypotetickou situaci:

```
.box {
  color: blue;
  color: var(--my-color);
}
```

Proměnnou `--my-color` jsme zde zapomněli definovat. Jakou barvu bude text mít?

Možná vás to překvapí, ale černou. Zdědí ji po výchozím nastavení pro text. Z pohledu prohlížeče to vypadá tak, že modrá (`blue`) byla předefinovaná proměnnou (`var(--my-color)`) . To si pamatujte.

Fakt, že hodnotu proměnné prohlížeč nezná, jej nezajímá. Mohli byste mu ji totiž hned v následující vteřině podstrčit třeba JavaScriptem. Ale pokud jste CSS proměnnou `var(--my-color)` zatím nedefinovali, aplikuje běžnou dědičnost a vezme barvu z rodičovského prvku.

Modrou barvu v příkladu ale zobrazí prohlížeč, který proměnné nepodporuje.

Pokud bychom chtěli fallback i pro podporující prohlížeče, musíme využít standardního mechanismu fallbacku za čárkou, který funkce `var()` nabízí:

```
.box {
  color: blue;
  color: var(--my-color, blue);
}
```

Živá ukázka hned následuje.

CodePen: vrdl.in/01y8x

Slučování hodnot a matematika ve funkci var()

Programátoři, tady buďte opět opatrní. Jste v CSS, nikoliv programovacím jazyce. Nic jako v následujícím příkladu fungovat nebude:

```
.box {
  --gap: 20;
  /* Smůla, fungovat nebude: */
  margin-top: var(--gap)px;
}
```

Výsledkem by byla hodnota s mezerou: `margin-top: 20 px`, které prohlížeč nebude rozumět.

Pro „přetypování“ namísto toho můžete použít funkci `calc()`:

```
.box {
  --gap: 20;
  margin-top: calc(var(--gap) * 1px);
}
```

Já vím… Já vím, co si vy programátoři myslíte. Dělat převody jednotky tímto způsobem pro vás asi není nejhezčí… Ale z pohledu CSS má toto logiku. Pořád nezapomínejme, že toto nejsou plnohodnotné proměnné.

Pojďme si ještě ukázat další příklady s funkcí `calc()`. Použít nějakou matematiku je zde možné:

```
.box {
  --gap: 20px;
  margin-top: calc(var(--gap) + 1px); /* 21 px */
  margin-top: calc(var(--gap) / 2); /* 10 px */
}
```

Psal jsem, že nám slučování hodnot neprojde. Slučování řetězců ale možné je:

```
.box {
  --text-hi: 'Ahoj';
  --text-sentence: var(--text-hi)', jsem ráda, že tě vidím!';
}
```

Sdílení v HTML

V CSS si můžeme například přednastavit stupnici hodnot:

```
html {
  --size-lg: 1.25rem;
  --size-sm: 0.8rem;
}
```

V HTML si pak vybrat jednu z nich:

```
<p style="font-size: var(--size-lg)">
  Lorem ipsum…
</p>
```

Daleko zajímavější je ale změna hodnoty vlastnosti z HTML. V CSS si definujeme vlastnost i její hodnotu:

```
html {
  --color: black;
}

p {
  color: var(--color);
}
```

V HTML ji pak změníme:

```
<p style="--color: blue">
  Lorem ipsum…
</p>
```

Oba příklady jsou samozřejmě děsně primitivní. Později ještě ukážu, jak je možné sdílení v HTML prakticky využít. Tady je CodePen obsahující oba zjednodušené scénáře.

CodePen: vrdl.in/shwao

Sdílení v JavaScriptu

Základní metody pro čtení nebo zápis jsou jednoduché:

```
// číst hodnotu
getComputedStyle(element).getPropertyValue("--color");

// zapsat hodnotu
element.style.setProperty("--color", "blue");
```

CodePen: vrdl.in/recv5

V jQuery od verze 3.1 se s „proměnnými“ pracuje stejně jako s běžnými CSS vlastnostmi. Aby ne, když to jsou vlastnosti, jen volitelné. Pokud tedy jQuery ještě používáte…

```
// číst hodnotu
$('#box1').css('--color');

// zapsat hodnotu
$('#box2').css('--color', 'blue');
```

CodePen: vrdl.in/megob

Podpora a fallbacky

CSS proměnné podporují všechny moderní prohlížeče. Bez podpory jsme například v Opeře Mini a ve všech Internet Explorerech. Jasně, obvyklí podezřelí. To vůbec nevadí. Více o podpoře je na webu CanIUse.
caniuse.com/css-variables

Tolik k volitelným vlastnostem a CSS proměnným. Teď už vám můžu nabídnout jen sumarizaci toho nejdůležitějšího.

Shrnutí

Pojďme si sesumírovat klady a zápory volitelných vlastností.

- Plus: Počítají se v prohlížeči a jsou dostupné z CSS, HTML i JS.
- Minus: Není je možné použít na jakémkoliv místě v kódu jako preprocesorové proměnné. Protože to nejsou proměnné.

Funkce calc()

Funkce `calc()` v CSS umožňuje vložit matematický výraz místo hodnoty vlastnosti.

Je velmi dobře podporovaná. Je užitečná. A je škoda, že ji kodérky a kodéři nepoužívají častěji. Pojďme to zkusit zlepšit. Nejprve si ukažme dvě jednoduchá využití:

```
.box {
  width: calc(100% / 3);
}
```

Rozdělení šířky na tři díly byste asi zvládli i bez kalkulačky, ale výhoda zápisu s `calc()` je v tom, že nedojde k zaokrouhlení z vaší strany.

```
.box {
  margin-bottom: calc(1em - 2px);
}
```

Něco takového je velmi užitečné. Například když z odsazení nebo šířky elementu potřebujete odečíst šířku rámečku (`border`).

Funkce `calc()` je samozřejmě velmi dobře použitelná právě v různých layoutech CSS. Proto má své místo i v poslední kapitole této knihy.

Není to stejné jako matematika v preprocesorech?

Prosím nezaměňujte `calc()` s výpočty v CSS preprocesorech, jako je Sass. V preprocesoru se musíme spokojit s výrazy, které se mohou zkompilovat do CSS ještě předtím, než prohlížeč stránku vidí:

```
.box {
  width: (100% / 3)
  // › width: 33.33333333%
}
```

Jenže preprocesoru dochází dech v momentě, kdy potřebuji kombinovat více jednotek. Vezměme příklad třísloupcového rozvržení s vnějším okrajem o šířce `1em`:

```
.box {
  width: calc(100% / 3 - (2 * 1em))
}
```

Tady je vidět síla funkce `calc()`. Počítá se v prohlížeči a jedině prohlížeč zná konkrétní hodnoty, které zastupují jednotky, jako je `em`.

Že se vám to někdy hodilo? Že to obcházíte podivnými hacky? Vítejte v klubu!

Příklady použití

Pojďme si teď více ukázat možnosti praktického využití ve vašich projektech.

1) Ukaž matiku!

Nejprve si více rozebereme příklad podobný tomu v úvodu:

```
.box {
  width: calc(100% / 7);
}
```

Proč nenapsat rovnou `width:14.2857`? Ze dvou důvodů:

1. *Čitelnost kódu*. Vsaďte se, že na původ čísla 14,2857 zapomenete. Nejpozději za týden. Správa této části kódu pro vás pak bude znamenat kladení otázek typu „jak jsem k tomu číslu došel?“. Vždy je lepší zapsat výpočet než výsledek.
2. *Zaokrouhlování*. V celé kráse vypadá výsledek takto: 14,285714286. Kodéři občas podlehnou pokušení a desetinná místa zaokrouhlí. A pak se diví, že se jim v nějakém konkrétním rozlišení a prohlížeči rozvržení stránky rozpadlo. Nebuďte takoví kodéři. Nezaokrouhlujte.

2) Responzivní obrázky

V parametru `sizes` značky `<img>` je znalost `calc()` naprosto klíčová. Zkuste si bez téhle prima funkce spočítat šířku obrázku uvnitř layoutu, který v responzivním layoutu splňuje následující podmínky:

1. Na větších displejích zabírá polovinu šířky layoutu stránky. Ale bez vnějších okrajů layoutu (10 pixelů) a samotné stránky (15 pixelů).

2. Na menších zabírá celou šířku viewportu bez vnějších okrajů stránky (15 pixelů).

```
<img sizes="calc((100vw - 2 * 15px) / 2) - 10px),
            calc(100vw - 2 * 15px)">
```

Uznávám, že to vypadá docela složitě. Ale bez `calc()` byste layout stránky pro potřebu responzivních obrázků nedokázali popsat.

(Pro zájemce: více o atributu `sizes` najdete na Vzhůru dolů. vrdl.in/srcsiz)

Podpora v prohlížečích

Funkci `calc()` podporují všechny moderní prohlížeče.

Na CanIUse si můžete přečíst, v jakých situacích tuto užitečnou funkci nepodporuje Internet Explorer 11. Pochybuji, že by se vám do toho chtělo. Je to dlouhý seznam. caniuse.com/calc

Funkce min(), max() a clamp()

Funkce, které se uvádějí místo hodnot v deklaracích stylů, umožňují porovnat dvě a více hodnot:

Funkce	Význam
`min()`	Vrací nejmenší hodnotu z argumentů oddělených čárkou. Např. `min(50%, 200px)`.
`max()`	Vrací největší hodnotu z argumentů oddělených čárkou. Např. `max(50%, 10vw, 200px)`.
`clamp()`	Vrací prostřední hodnotu, pokud není menší než první a větší než třetí. Např. `clamp(100px, 20%, 200px)`.

Jsou součástí čtvrté verze specifikace „CSS Values and Units Module“, podporují je všechny moderní prohlížeče a ještě k tomu jsou užitečné.

Funkce `min()`, `max()` a `clamp()` nám v CSS prakticky umožní lépe řídit velikost prvků, dodržovat správné mezery mezi prvky nebo třeba implementovat plynulou („fluidní“) typografii.

Mají samozřejmě své místo i v této knize. Používám je například pro řešení svatého grálu mezi layouty nebo při vysvětlování krkavčí techniky.

Jak fungují?

Funkce min() a max()

U funkcí `min()` a `max()` to je jednoduché – zadáte seznam argumentů a prohlížeč určí, která z hodnot je nejmenší nebo největší.

Příklad:

```
.box-1 {
  width: min(100px, 50%);
}
```

Šířka `.box-1` bude určena tím menším přepočtem na pixely z obou hodnot. V dostatečně širokém rodičovském prvku to bude většinou `100px`, ale v opravdu malých šířkách se může použít `50%`, protože vypočtená hodnota šířky bude menší než `100px`.

Zkusme si představit totéž pro funkci `max()`:

```
.box-2 {
  width: max(100px, 50%);
}
```

Šířka `.box-1` bude určena větší pixelovou hodnotou vypočtenou z obou atributů. Jak asi sami tušíte, většinou se použije hodnota `50%` a jen na opravdu malých rozlišeních prohlížeč vybere `100px`.

V tomto případě jde o obdobu zápisu `width:50%;min-width:100px`. V ukázce to sami uvidíte.

CodePen: vrdl.in/1jgam

Funkce clamp()

Jeden z významů slova „clamp“ v angličtině je „svěrka“, což je vcelku přesné. První a poslední atribut v závorce totiž udává minimální a maximální sevření, prostřední je ideální hodnota:

```
.box-3 {
  width: clamp(100px, 50%, 300px);
}
```

Pitva funkce clamp().

Vysvětlíme to ještě detailně:

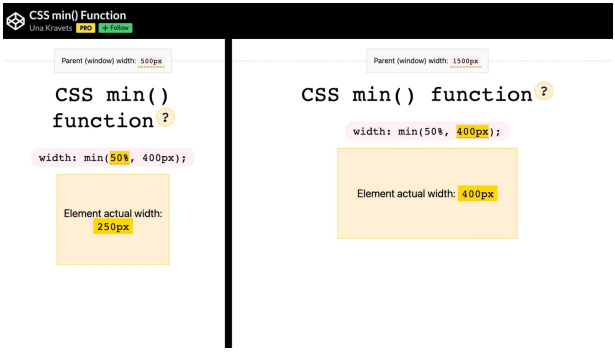
- Prostřední argument je *ideální hodnota*. `.box-3` bude široký `50%`, pokud nevstoupí v platnost svěrka z jedné nebo druhé strany.
- První argument je *minimální hodnota*. Prvek nesmí být užší než `100px`.

- Třetí argument je *maximální hodnota*. `.box-3` nebude nikdy širší než `300px`.

Je to srozumitelné? Ještě si to případně zkuste na CodePenu.

CodePen: vrdl.in/op3tz

Následující ukázku jsem si vypůjčil od Uny Kravets. Je to krásná vizualizace, ve které je vidět aktuální šířka rodiče, elementu a pak také argument funkce `min()`, který je v dané šířce obrazovky aktivní.



Funkce min() vybere prostě tu menší ze dvou hodnot. Zdroj: Una Kravets.

Pokud si to budete zkoušet naživo, zkuste si hýbat se šířkou viewportu.

CodePen: vrdl.in/lkmsg

Ve specifikaci se uvádí, že funkci `clamp()` je možné zapsat pomocí `min()` a `max()` jako `max(MIN, min(VAL, MAX))`. Nevím jak vám, ale mně to moc při snaze pochopit funkci `clamp()` nepomohlo.

Krása univerzálnosti

Pojďme však vzoreček ze specifikace rozebrat více. Nakonec možná dojdeme k tomu, že bude užitečný. V případě výše uvedeného kódu by zápis vypadal takto:

```
.box-3 {
  width: max(100px, min(50%, 300px));
}
```

Zopakuji i původní zápis pomocí funkce `clamp()`:

```
.box-3 {
  width: clamp(100px, 50%, 300px);
}
```

Jak už jste jistě pochopili, kód výše je stejný, jako bychom napsali následující deklaraci:

```
.box-3 {
  width: 50%;
  min-width: 100px;
  max-width: 300px;
}
```

Konstrukce s `min-width` a `max-width` v CSS máme a mnozí známe už od pravěkých dob. Tak proč zavádět nový zápis v podobě funkcí `min()`, `max()` a `clamp()`?

Za prvé je nový zápis stručnější a možná přehlednější, nejdůležitější je ale jeho univerzálnost. Použití není limitované na délkové vlastnosti `width` a `height`.

To je tak zajímavé, že to musíme rozebrat. Jen chvíli počkejte.

Demo: ideální šířka textu

Ve známé knize „The Elements of Typographic Style“ od Roberta Bringhursta se píše, že „za uspokojivou délku řádku pro jednosloupcovou stránku s patkovým písmem se obecně považuje 45 až 75 znaků“. vrdl.in/zaosw

Toto můžeme krásně definovat právě pomocí porovnávací funkce:

```
.card {
  width: clamp(45ch, 50%, 75ch);
}
```

Prvek `.card` bude zabírat `50%` šířky rodiče, ale nikdy méně než `45ch` a více než `75ch`. Pokud to nevíte, jednotka `ch` v sobě obsahuje šířku znaku `0`, což se považuje za průměrnou šířku znaku.

CodePen: vrdl.in/2q9xo

Demo: plynulá typografie

V další ukázce Uny Kravets konečně odbočíme mimo vlastnosti určující šířku elementu. Budeme nastavovat maximální a minimální velikost textu a tím plně využijeme nových porovnávacích funkcí.

```
h1 {
  font-size: clamp(1.5rem, 5vw, 3rem);
}
```

Velikost písma se bude s rozšiřováním okna zvětšovat od `1.5rem` až po `5rem`.

Jednotkou `vw` nastavujeme velikost písma na pět procent šířky viewportu. Abychom se přitom vyhnuli extrémně malým a extrémně velkým hodnotám `font-size`, pomáháme si funkcí `clamp()`.

CodePen: vrdl.in/crs5t

Čtenář Dan Srb mi po publikování první verze tohoto textu na Vzhůru dolů poslal ještě jeden tip ke stupňovitému zvětšování písma:

> **Pokud chcete například zajistit, aby se od pětisetpixelové šířky viewportu začalo písmo zvětšovat z 1rem na 2rem až k hranici šířky viewportu 900px, pak je tu tahle kalkulačka na snadno zapamatovatelné adrese: xgkft.csb.app. (Lze použít např. i pro vlastnost padding.)**

Dodejme ještě, že autorka původního CodePenu nás nabádá, abychom si při těchto hrátkách s velikostí písma dávali pozor na přístupnost.

Omezení velikosti textu pomocí funkcí `max()` nebo `clamp()` může být proti pravidlům přístupnosti WCAG, která doporučují, aby si uživatelé mohli písmo libovolně zvětšovat sami.

Co byste ještě měli vědět o porovnávacích funkcích?

Funkce `min()`, `max()` a `clamp()` mají v DNA pár důležitých vlastností, které se mi jinam nevešly, ale přesto si zaslouží vaši pozornost:

- Funkce můžete skládat a zanořovat. Např. `font-size:max(min(0.5vw,1em),2rem)`. Není ale garantováno, že vás z toho nerozbolí hlava.
- Stejně jako funkce `calc()` umožňují porovnávací funkce používat matematické výrazy se sčítáním (+), odčítáním (-), násobením (*) a dělením (/) jako součástí hodnot. Zápis `font-size:max(calc(0.5vw - 1em),2rem)` je tudíž stejný jako `font-size:max(0.5vw - 1em,2rem)`.
- Co když ve funkci `clamp()` prohodíte minimum a maximum? Vyhraje minimum. Například zápis `clamp(100px,…,50px)` bude vracet hodnotu `100px`, čímž překročí maximum uvedené ve třetím argumentu.

Podpora v prohlížečích

Je to dobré. Funkce `min()`, `max()` a `clamp()` podporují všechny moderní prohlížeče. V Internet Exploreru vám tyto funkce neprojdou.

Můžete to ověřit na CanIUse: caniuse.com/css-math-functions

Containment a vlastnost contain

Vlastností `contain` označujeme části stránky, které jsou izolované od zbytku. Izolujeme je proto, aby prohlížeč nemusel při změnách překreslovat celou stránku. Ušetříme tím výkon na stránkách s komplexním DOMem.

Prohlížeče se už dlouho různými způsoby snaží nepřepočítávat vzhled celé stránky při každé změně samy.

Kromě toho existují kodérské triky, jak to udělat v běžném CSS (hledejte např. „Layout Boundaries“). No a relativně novou možností je použít pro tyhle účely vlastnost `contain`.

Dva příklady

Vlastnost `contain` může ušetřit čas potřebný pro vykreslování hlavně v případech, kdy DOM obsahuje tisíce uzlů. Následující příklady proto berte jako schématické a hodně zjednodušené.

Přidání prvku do DOMu

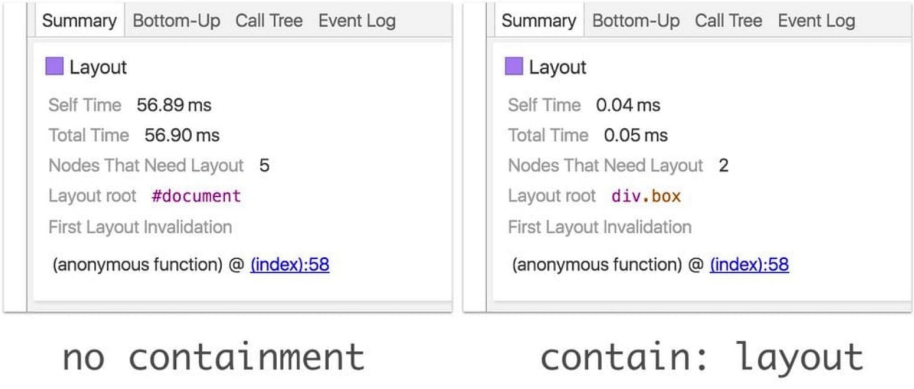
Tuhle ukázku jsem převzal z dokumentace od Googlu. Máme následující HTML:

```
<section class="view">
  Home
</section>

<section class="view">
  About
</section>
```

A teď JavaScriptem přidáme nový prvek:

```
<section class="view">
  Home
</section>

<section class="view">
  About
  <div class="newly-added-element">Check me out!</div>
</section>
```

Přidání nového prvku spouští rovnou tři kroky procesu překreslování – styl, layout a paint. Blbé ovšem je, že se ten proces spouští pro celý DOM a celou stránku. Může pak pomoci přidání vlastnosti `contain`.



Ilustrační obrázek: Čas potřebný pro fázi layout můžeme snížit díky omezení na konkrétní boxík menšímu počtu prvku k přepočítání. Zdroj developers.google.com.

Podívejme se ještě na jednu ukázku.

Výpis článků mimo viditelnou část obrazovky

Vezměme, že na stránce máme stovky či tisíce položek typu články, produkty nebo třeba tweety. Většinu z nich uživatelé neuvidí v prvním vykresleném viewportu. Zároveň jde o samostatné a izolované prvky, které se se zbytkem stránky nijak vzájemně neovlivňují.

Takhle může vypadat jejich výpis ve stránce:

```
<h1>Articles</h1>
<article> … </article>
<article> … </article>
```

Představme si, že prvků `<article>` jsou zde stovky a zároveň mají složitou vnitřní DOM strukturu.

Pomocí vlastnosti `contain` můžeme prohlížeč informovat, ať tyto prvky vyjme z celkového vykreslování stránky:

```
article {
  contain: content;
}
```

Prohlížeči tak dáváme instrukci, že prvky `.element`, které „nevidí“ ve viewportu může v klidu vynechat z počítání vzhledu celé stránky.

Ušetříme tím v některých situacích slušný renderovací čas.

CodePen: cdpn.io/e/gOrMOWd

Typy „containmentu“

Zatím se mi nepovedlo najít vhodné české slovíčko pro teorii, o které se ve specifikaci mluví jako o „CSS containmentu“. Jde o soběstačné a nezávislé zapouzdření prvku, což je ale poněkud kostrbaté označení.

Známe čtyři typy zapouzdření, které jsou zároveň možné hodnoty vlastnosti `contain`:

Hodnota `contain`	Typ zapouzdření
`size`	**Zapouzdření pro velikost.** Prohlížeči říkám, že velikost prvku nijak neovlivní jeho potomci. Pokud nastavíme `contain:size`, je potřeba v CSS také tomuto prvku nastavit nějakou velikost. Jinak prohlížeč počítá, že velikost je nulová, což nechceme. Zapouzdření velikosti samo o sobě zase tak moc výkonu při renderování neušetří.
`inline-size`	**Zapouzdření pro „inline“ velikost.** Totéž jako `size` jen pro změny velikosti na vodorovné ose.
`layout`	**Zapouzdření pro rozvržení.** Říkáme tím, že se layout potomků prvku a zbytku stránky nijak vzájemně neovlivňují. Díky tomu může při zápise `contain:layout` prohlížeč vynechat počítání layoutu vnitřních prvků elementu a zaměřit se jen na prvek, který tuto vlastnost má nastavenou.
`paint`	**Zapouzdření pro vykreslení.** Informujeme tímto, že žádný vnitřní prvek nevyčnívá ze svého rodiče. Uvedení `contain:paint` prohlížeči umožňuje potenciálně přeskočit vykreslení potomků, pokud je prvek mimo obrazovku.
`style`	**Zapouzdření pro styly.** Říkáme, že ovlivněný prvek vyjímáme z počítání hodnot napříč dokumentem, které provádějí vlastnosti jako `counter-increment`, `counter-set` nebo `quotes`.

Hodnoty vlastnosti `contain` jde kombinovat, takže můžete například uvést `contain: style paint`.

Speciální hodnoty

Za účelem zjednodušení problematiky pro nás, autory webů, přichází specifikace se speciálními hodnotami vlastnosti `contain`:

Hodnota `contain`	Typ zapouzdření
`strict`	Všechny typy zapouzdření, kromě stylů. Totéž jako zápis `contain: size layout paint`.
`content`	Všechny typy zapouzdření, kromě stylů a velikosti. Totéž jako `contain: layout paint`.

Hodnota `strict` ušetří prohlížeči více času, ale zase musíme znát a definovat velikost prvku.

Jak to použít v praxi? Pojďme se zde vrátit k druhé ukázce – renderování desítek či stovek článků mimo viditelnou část obrazovky:

- Pokud bychom použili `contain:content`, nemusíme definovat výšku jednotlivých bloků. Na druhou stranu bude prohlížeč při prvním vykreslení považovat výšku za nulovou a nevykreslí například správně velká rolovátka.
- Pokud bychom použili `contain:strict`, prohlížeči musíme výšku sdělit, ale zase nenastane přepočítání velikosti rolovátka.

Vlastnost contain vytváří nové kontexty

Pokud containment použijete s hodnotami `paint`, `strict` nebo `content` vytvoří se nové kontexty, které je možné dělat i jinými metodami v CSS:

- Nový obsahující blok (containing block) – pro potomky, jejichž vlastnost `position` je `absolute` nebo `fixed`, takže něco jako `position:relative`.
- Nový kontext stohování (stacking context), ve kterém můžete nezávisle na zbytku stránky umísťovat prvky do vrstev pomocí `z-index`.
- Nový kontext formátování bloku (block formatting context), který například umí obsáhnout vnitřní prvky umístěné pomocí `float` nebo zakáže spojování vnějších okrajů (`margin`).

Využít containment, konkrétně například `contain:content`, tedy můžete i v případě, že nechcete šetři výkon, ale usnadnit si kodérskou práci.

Podpora je plná

Vlastnost `contain` nepodporuje Internet Explorer, což vůbec nevadí. Všechny moderní prohlížeče v containmentu jedou s námi.

Viz také CanIUse.com/mdn-css_properties_contain

@supports

V CSS nebo JavaScriptu se nám občas hodí napsat kód, který rozlišuje varianty podle podpory určitých CSS vlastností.

K tomu můžeme použít poměrně nové pravidlo `@supports`, součást specifikace „CSS Conditional Rules Module“.

S pomocí `@supports` je možné psát „Feature Queries“, dotazy na vlastnosti. Vezměme například ukázky s dotazem na nativní podporu layoutu typu masonry:

```
@supports (grid-template-rows: masonry) {
  .text {
    color: green;
  }
}
```

Pokud se na ukázku podíváte v prohlížečích, které podporují vlastnost `grid-template-rows` a zároveň její hodnotu `masonry`, což je na začátku roku 2022 zatím jen Firefox Nightly, uvidíte zelený text.

Ve všech ostatních prohlížečích bude text černý.

CodePen: vrdl.in/3q7kt

Všimněte si důležité věci – pro detekování podpory se musíme dotazovat na vlastnost i její hodnotu. Takže pokud bychom se ptali na podporu vlastnosti `display`, musíme se zeptat například na `display:block`.

Logické operátory: not, and a or

Specifikace „Feature Queries“ velmi správně definuje logické operátory podle zvyklostí z jiných jazyků.

Operátor not

Velice často se nám může hodit negace a klíčové slovo `not`:

```
@supports not (display: grid) {
  .box {
    float: right;
  }
}
```

Kód se aplikuje jen v těch prohlížečích, které *nepodporují* CSS grid.

Operátor and

Logický operátor konjunkce (tedy průnik dvou množin) – and:

```
@supports (display: table-cell) and (display: list-item) {
  .text {
    color: green;
  }
}
```

Použijeme ho v případech, kdy musí platit všechny podmínky.

Operátor or

Další operátor – or – definuje logickou disjunkci (tedy sjednocení množin):

```
@supports (transform-style: preserve) or
  (-moz-transform-style: preserve) {
  .text {
    color: green;
  }
}
```

Podmínka je platná v případě, že prohlížeč podporuje alespoň jednu z deklarací uvedených v závorkách. To se může hodit právě pro práci s prohlížečovými prefixy, zde -moz. Píšu o tom ještě v textu níže.

Kombinování operátorů

Při kombinování operátorů se vyplatí být doslovný – vždy uvádějte závorky.

Tyto deklarace nejsou validní:

```
@supports display: flex {
  /* Neplatné */
}

@supports (transition-property: color) or
  (animation-name: foo) and
  (transform: rotate(10deg)) {
  /* Neplatné */
}
```

Tento zápis ale validní je:

```
@supports (display: flex) {
  /* … */
}

@supports ((transition-property: color) or
  (animation-name: foo)) and
  (transform: rotate(10deg)) {
  /* … */
}
```

Aby nedošlo k záměně mezi `and` a `or`, syntaxe je specifikována tak, aby byly výslovně psány jako `and` nebo `or`.

Prohlížečové prefixy

Dalším překvapením může být nutnost používat všechny prefixové vlastnosti. Pokud máte v cílové skupině uživatele prohlížečů, které vlastnost podporují jen s použitím prefixů, musíte je uvést všechny:

```
@supports ((box-shadow: 0 0 2px black inset) or
  (-moz-box-shadow: 0 0 2px black inset) or
  (-webkit-box-shadow: 0 0 2px black inset) or
  (-o-box-shadow: 0 0 2px black inset)) {
  /* Kód pro všechny prohlížeče podporující box-shadow,
     včetně těch už historických */
}
```

Příklad mám ze specifikace. Dnes už byste prefixy k `box-shadow` nepotřebovali, takže tím kromě jiného ilustruji, že prefixy je potřeba uvádět jen tehdy, pokud byste je psali i v běžném CSS kódu.

Nejlepší je ale starost o prefixy přenechat automatizaci, konkrétně nástroji Autoprefixer.

Detekce podpory selektorů

Některé moderní prohlížeče umožňují detekovat také podporu určitých CSS selektorů. Dělají to pomocí nové funkce `selector()`:

```
@supports selector(A > B) {
  /* Kód pro prohlížeč, který podporuje selektor A > B */
}
```

Funkce `selector()` je součástí nové verze specifikace modulu „CSS Conditional Rules“ a podporují ji všechny moderní prohlížeče.

Řešení pro podporu detekce selektorů už dříve hledalo více autorů pomocí různých hacků. Zajímavý je například tento, který se dotazuje na podporu pseudotřídy `:placeholder-shown`:

```
.foo { color: red }

:not(*):placeholder-shown,
.foo {
  color: green
}
```

V textu „Using Feature Detection, Conditionals, and Groups with Selectors“ světu své řešení představil „náš“ Jirka Vebr. Tož díky! vrdl.in/vebrfeature

Feature Queries v JavaScriptu

Metoda `CSS.supports()` vrací hodnotu `true` nebo `false` určující, zda prohlížeč danou funkci CSS podporuje nebo ne.

Toto jsou možné zápisy:

```
result = CSS.supports("grid-template-rows", "masonry");
result = CSS.supports("display: grid");
result = CSS.supports("(--foo: red)");
result = CSS.supports("(transform-style: preserve) or (-moz-transform-style: preserve)");
```

V prvním řádku vidíme verzi se dvěma parametry: v prvním je prostě hodnota a ve druhém vlastnosti.

Další tři řádky ukazují variantu, kdy do textu uvedeme `DOMString` rovnou s hodnotou celé podmínky.

Ti z vás, které jsem ještě neunavil detailním líčením, si možná všimli detekce autorské vlastnosti (nebo též „custom property“ či „CSS proměnné“) ve třetím řádku (`(--foo: red)`). Ano, i tu je možné detekovat.

CSS hacky a Progressive Enhancement

Udělejme si teď pro zajímavost výpravu do historie. `@supports` totiž navazuje na silnou epochu „CSS hacků“, kterou jsme k všeobecné nelibosti prožívali zhruba v první dekádě 21. století.

Tenkrát nebylo možné v CSS podporu vlastností detekovat, proto kodérky a kodéři hledali chyby v prohlížečích při implementaci CSS pravidel, tedy zápis, který funguje v určitých prohlížečích a v jiných naopak ne.

Asi nejznámější byl podtržítkový hack:

```
.box {
  /* Kód pro všechny prohlížeče: */
  position: fixed;
  /* Kód jen pro IE5+: */
  _position: absolute;
}
```

Pokud by vás tahle dnes už nechvalně známá praxe zajímala, píšou o ní hezky v přehledu CSS hacků na v těch časech populárním magazínu Interval.cz. vrdl.in/mo8hx

Brrr, úplně mi běhá mráz po zádech, když si představím, že v té době jsem nakódoval opravdu hodně webů. Jak bych byl tehdy za `@supports` vděčný!

CSS hacky a dnes `@supports` jsou důležitou částí zásadní webařské techniky: postupného vylepšování (Progressive Enhancement). Funguje asi takto:

1. Vyrobíte základní řešení fungující ve všech prohlížečích.
2. Nad tím postavíte lepší řešení fungující jen v některých prohlížečích.

Mezi jednotlivými řešeními je detekce vlastností (nikoliv prohlížeče!), například právě pomocí dotazu `@supports`.

Výsledkem je, že nějaké řešení máte pro nejširší možnou skupinu zařízení. Pro web ideální.

Podpora @supports a limity použitelnosti

První omezené použití `@supports` vychází z principů fungování prohlížečů – „Feature Queries“ nelze použít ke kontrole, zda prohlížeč vlastnost, hodnotu nebo selektor podporuje správně a bez chyb.

Pokud si totiž prohlížeč „myslí“, že vlastnost umí, vrátí na dotaz kladnou odpověď. Ale soudruzi z NDR mohli někde v implementaci vlastnosti udělat chybu.

Dalším omezením je samotná podpora vlastnosti, v tomto případě však záleží na použití: `@supports` nepodporuje vůbec žádný Internet Explorer. caniuse.com/css-featurequeries

Je ovšem otázka, jak moc tahle chybějící podpora vadí. Mně vůbec. A hned vám řeknu proč.

Internet Explorer je totiž skoro vždy ve skupině prohlížečů, které tu či onu vlastnost nezvládají. Proto vůbec nevadí, když nerozumí ani otázce, zda vlastnost zvládá…

Ukažme si problém s Explorerem ještě na jednom příkladu.

Závěrečná ukázka s detekcí CSS gridu

Řekněme, že máme HTML s jedním kontejnerem a třemi položkami:

```
<div class="container">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
</div>
```

Chtěli bychom položky jednoduše umístit vedle sebe, což s pomocí flexboxu uděláme takto:

```
.container {
  display: flex;
}

.item {
  flex: 1;
}
```

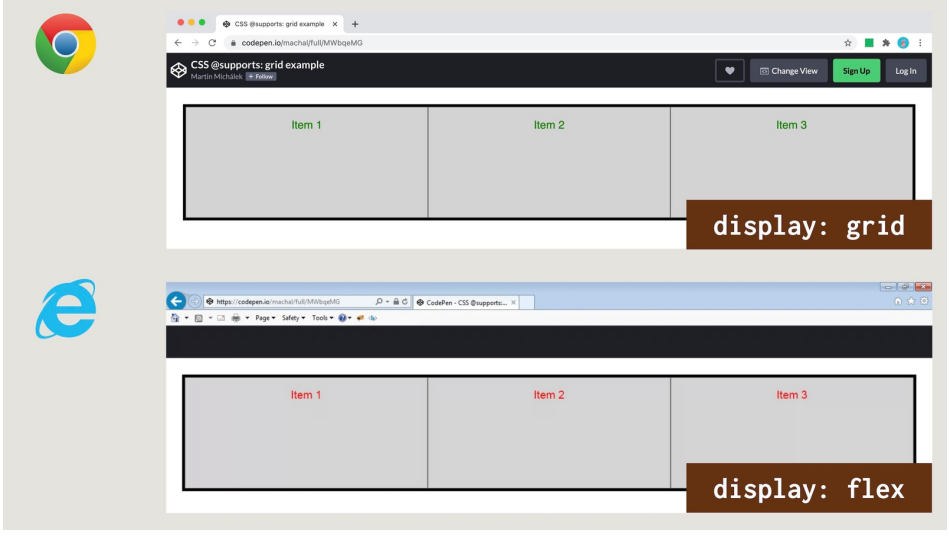
Z nějakého důvodu bychom ovšem v moderních prohlížečích chtěli použít CSS grid:

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
}
```

Je to samozřejmě naprosto schůdné, použijeme detekci pomocí `@supports`, dotazu na vlastnosti:

```
@supports (display:grid) {
  .container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
  }
}
```

Na pohled to bude vypadat stejně. Takhle jednoduchý kód se samozřejmě nevyplatí psát ve flexboxu i gridu. Budu ale předpokládat, že jde o základ pro využití pokročilejších vlastností gridu, kterých má požehnaně.



Dvě verze pro dvě kategorie prohlížečů. To nám snadno umožní Feature Queries, dotazy na vlastnosti.

Tady nastává moment, pro který jsme si v demíčku šli. MSIE nejenže nezná `display:grid`, ale zároveň nezná `@supports`, takže tento blok kódu vynechá ze zpracování. A to je dobře.

CodePen: vrdl.in/pdc3h

V CodePenu uvidíte v prohlížečích podporujících grid zelené písmo. V ostatních je to červeně.

Autoprefixer

Autoprefixer je – jak z názvu vyplývá – automatizační nástroj, který přidává prohlížečové prefixy do CSS kódu.

V knížce se o něj celou dobu otíráme, takže se mu konečně pojďme věnovat.

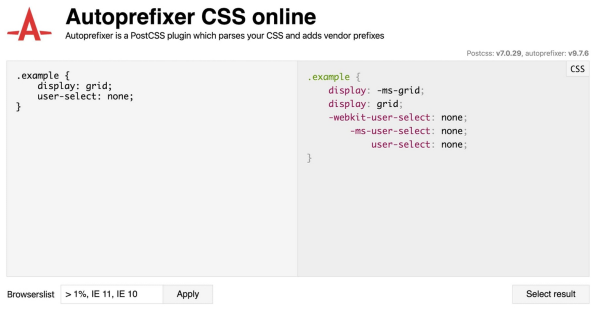
Autoprefixer například z následujícího kódu…

```
.box {
  hyphens: auto;
}
```

… udělá tento kód:

```
.box {
  -webkit-hyphens: auto;
  -ms-hyphens: auto;
  hyphens: auto;
}
```

Tím pádem budou kódu rozumět všechny prohlížeče, i když vlastnost hyphens neznají a podporují jen prefixované varianty.



Autoprefixer v celé kráse své online verze.

Jak Autoprefixer funguje?

Autoprefixer je balíček, který si můžete do projektu nainstalovat jako plugin pro PostCSS, což je nástroj sloužící k jakémukoliv automatickému zpracovávání CSS kódu.

Používat můžete Autoprefixer mnoha různými způsoby, o těch bude za chvíli řeč. Nejprve ale k nastavení.

Nastavení pomocí Browserslistu

Browserslist je způsob, jak Autoprefixeru a podobným nástrojům (jakým je například Babel pro zpracování JavaScriptu) určit, které prohlížeče v projektu podporujete.

Nejlepším způsobem, jak poskytnout toto nastavení, je soubor `.browserslistrc` v kořenovém adresáři projektu. Alternativně můžete přidat klíč `browserslist` do souboru `package.json` nebo přímo v nastavení Autoprefixeru.

Výchozí nastavení vypadá asi takto:

```
last 1 version, >1%
```

Potom tento kód...

```
.box {
  display: flex;
}
```

... bude mít totožný zápis i po zásahu Autoprefixeru. Pomocí nastavení `last 1 version, >1%` jsme řekli, že budeme podporovat jen poslední verzi prohlížečů a takovou verzi, která má ve světě rozšíření více než jedno procento.

Pojďme ale změnit zápis konfigurace takto:

```
> 1%, IE 11, IE 10
```

Pak bude výsledek jiný:

```
.box {
  display: -ms-flexbox;
  display: flex;
}
```

V konfiguraci Browserslistu je `IE 10`, takže Autoprefixer přidá zápis pro tento stařičký prohlížeč: `display:-ms-flexbox`.

Použití Autoprefixeru

Možností využití je celá řada, podívejme se na ně.

Online

Pokud si chcete jen hrát nebo přidat pár prefixů, zajděte na online pískoviště. autoprefixer.github.io

Příkazová řádka

Využijte balíček postcss-cli. vrdl.in/96mcu

```
npm install postcss-cli autoprefixer
npx postcss *.css --use autoprefixer -d build/
```

Gulp

Pro kombinování s Gulpem je k dispozici balíček gulp-postcss. vrdl.in/2a70d

```
gulp.task('autoprefixer', () => {
  const autoprefixer = require('autoprefixer')
  const sourcemaps = require('gulp-sourcemaps')
  const postcss = require('gulp-postcss')

  return gulp.src('./src/*.css')
    .pipe(sourcemaps.init())
    .pipe(postcss([ autoprefixer() ]))
    .pipe(sourcemaps.write('.'))
    .pipe(gulp.dest('./dest'))
})
```

Webpack

postcss-loader je balíček vhodný pro vaši práci ve Webpacku. vrdl.in/lcpb5

`webpack.config.js`:

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ["style-loader", "css-loader", "postcss-loader"]
      }
    ]
  }
}
```

`postcss.config.js`:

```
module.exports = {
  plugins: [
    require('autoprefixer')
  ]
}
```

Použití Autoprefixeru s dalšími nástroji a více nápovědy najdete na oficiálním webu. vrdl.in/ptaqs

Shrnutí deváté kapitoly

1. **Jaký typ zobrazení zajistí deklarace display:inline-block?**
   a) zvnějšku je prvek blokový, zevnitř se vykresluje do řádku
   b) vnitřně je to blokový element, zvnějšku inlinový – netvoří zalomení
   c) v nové verzi specifikace je už toto psát zakázáno
2. **Jak zajistit nevytékání float prvků z rodičovského elementu?**
   a) `display:flow-root`
   b) `display:float-root`
   c) `display:clear-float`
3. **K čemu slouží logické vlastnosti (CSS Logical Properties)?**
   a) jde o logické funkce při počítání matematiky v `calc()`
   b) k sázení layoutu v závislosti na směru toku jazyka dokumentu
   c) jde o obecný název pro vnitřní a vnější okraje prvku
4. **Jaký je ideální způsob nastavení velikosti písma v dokumentu?**
   a) v procentech: `html {font-size:120%}`
   b) v „root em“: `html {font-size:1.2rem}`
   c) v „emkách“: `html {font-size:1.2em}`
5. **Co definuji zápisem html {--color:blue}?**
   a) globální proměnnou `--color` s hodnotou `blue`
   b) jde jen o hack, který zamezí viditelnosti zápisu v Exploreru verze 4
   c) autorskou vlastnost `--color` s hodnotou `blue`
6. **Jaké jsou výhody autorských vlastností a jejich použití?**
   a) je to totéž jako proměnné v preprocesoru
   b) počítají se v prohlížeči, jsou dostupné v CSS kaskádě, HTML, JS
   c) je to totéž jako proměnné v JavaScriptu
7. **Jak se prohlížeče zeptat na podporu vlastnosti transform-style?**
   a) `@supports transform-style { }`
   b) `@supports (transform-style) { }`
   c) `@supports (transform-style:preserve) { }`

Řešení: 1. b), 2. a), 3. b), 4. a), 5. c), 6. b), 7. c).

Na závěr

Právě jste tuhle knihu dočetli.

Díky, že jste jí věnovali svůj čas, a jsem napjatý, co na ni říkáte. Budu vděčný za každý váš názor.

Zpětná vazba, pochvaly, hlášení chyb

Dáte mi prosím zpětnou vazbu? Pomůže mi vylepšovat knížku v dalších verzích, přinejmenším v případě e-booku. Menší aktualizace dostanou všichni čtenáři zdarma.

Váš názor si přečtu moc rád. Vyplňte dotazník spokojenosti (zabere jen pět minut a je to možné udělat i anonymně). vrdl.in/laydot.

Líbila se vám knížka?

Pokud jste byli spokojeni, řekněte to ostatním. Budu vděčný za veřejnou pochvalu na sociálních sítích. Použijte hashtag `#VdCSSLayout`.

Našli jste chybu?

I když jsem já i ostatní udělali všechno pro to, abychom se chybám vyhnuli, znáte to… Velmi pomůže, když na chyby nebo jiné problémy poukážete. V další verzi e-booku je opravím. Pište na martin@vzhurudolu.cz.

Odkazy k dalšímu studiu

Až začnete moderní layout v CSS používat, bude se vám, kromě knížky, jistě hodit i několik odkazů.

Vzhůru dolů

Na Vzhůru dolů najdete referenční příručky k většině vlastností, které najdete v knížce.

- Příručka ke gridu: vrdl.in/grid
- Příručka k flexboxu: vrdl.in/flex

- Příručka k multicol: vrdl.in/multicol
- Příručka k zarovnání boxů: vrdl.in/boxal
- Hodinový videokurz gridu: vrdl.in/vidgrid

Vzdělávací hry

Snažil jsem se, aby i v knížce bylo učení zábavou. Ale hry jsou přeci jen hry. Někomu může vyhovovat plnit úkoly v herním rozhraní a tím se layout učit.

- Flexbox Froggy: vrdl.in/ffrog
- Flex Box Adventure: vrdl.in/flexad
- Grid Garden: vrdl.in/gridg
- Grid Critters: vrdl.in/gridcr

Příklady použití

Další příklady z praxe, které se vám mohou hodit.

- Grid by Example: vrdl.in/gbex
- 1-Line Layouts: vrdl.in/1line

Generátory

Pokud ještě nemáte kód pro psaní layoutů v hlavě, zkuste použít generátory.

- Grid: vrdl.in/gridit
- Grid i flexbox: vrdl.in/bimxv

CSS-Tricks

Magazín Chrise Coyiera je asi nejlepší zdroj jednoduchých příruček k CSS vlastnostem v angličtině.

- CSS-Tricks: A Complete Guide to Flexbox: vrdl.in/csstf
- CSS-Tricks: A Complete Guide to Grid: vrdl.in/csstg

Specifikace W3C

Zdroj všech zdrojů. Osobně čtu specifikace moc rád. Řekl bych, že jsou napsané lépe, než si lidé myslí, a detailnější popis vlastností nikde jinde nenajdete.

- Flexbox: vrdl.in/w3flex
- Grid Layout: vrdl.in/w3grid
- Multi-column Layout: vrdl.in/w3mulco
- Box Alignment: vrdl.in/w3align

Vzhůru dolů: videa, další e-booky a tak dále

Na Vzhůru dolů najdete i další výsledky mé práce.

Videa

Vzdělávací videa jsou na mém webu novinkou. V době psaní této knihy je možné zakoupit hodinové video o CSS gridu, ale další videa na témata z knížky mám v plánu. Celkem je zde již přes 20 videí. vrdl.in/video

E-book „Vzhůru do (responzivního) webdesignu“

Kompletní průvodce návrhem a implementací designu responzivních webů. Vydáno v roce 2017, aktualizováno 2018. 311 stránek. vrdl.in/kniha

E-book „Vzhůru do AMP“

Průvodce frameworkem pro efektivní tvorbu rychlých webů. Napsáno s Robinem Pokorným v roce 2019. vrdl.in/vdamp

Sociální sítě

Sledujte novinky z oboru a články ze Vzhůru dolů na sociálních sítích:

facebook.com/vzhurudolu
twitter.com/vzhurudolu
instagram.com/vzhurudolu
linkedin.com/company/vzhůru-dolů

Poděkování

Děkuji všem, kteří první rukopis knížky četli a dodali mi cenný pohled zvenčí: Markéta Andreev Dragoun, Tomáš Kout, Honza Marek, Michal Mikláš, Radek Pavlíček, Kristýna Petrová, Břetislav Proft, Dan Srb, Zuzana Šumlanská.

Za pomoc s verzí 1.1 bych rád poděkoval Lence Silné a za připomínky též Petru Václavkovi a Rostislavu Kullovi.

Zvláštní poklonu věnuji Petru Jedinému Novotnému, jehož cit pro jazyk knihu zvelebil a jehož humor mi dodával sílu v posledních taženích bitvy o dopsání knížky. Dále děkuji Petru Šťastnému, Danielu Střelci a Jirkovi Koskovi. Ti mi pomohli propojit svět CSS se světem tisku, abych knihu mohl dodat tiskárně bez pomoci sazečů.

Ano, i tištěná verze této knihy vznikla s pomocí HTML, CSS a trošky JavaScriptu a PHP. Za tuto možnost děkuji autorům knihovny Paged.js, stejně tak jako těm, kteří pracují na nástroji Pandoc, jenž mi pomáhá tvořit elektronické verze knih.

Knížka by v této podobě nevznikla nebýt podpory a trpělivosti mojí ženy a netrpělivosti mých synů. Jano, Honzo, Kubíku, díky!

Rejstřík

@

@Media: 42
@Supports: 425

A

Align-content: 332
Align-items: 301
Align-self: 312
Align-tracks: 251
Anonymní položka: 172
Auto-fill: 233
Auto-fit: 233
Autoplacement: 167, 193
Autoprefixer: 190, 433
Autorské vlastnosti: 403

B

Baseline Alignment: 291
Bloková osa: 286, 388
Bootstrap: 37, 98
Box Align: 16, 281
Box model: 391
Break-after: 273
Break-before: 273
Break-inside: 273
Buňka mřížky: 176

C

Calc(): 411
CanIUse: 13
Chrome: 12
Clamp(): 414
CodePen: 10
Column-count: 262
Column-fill: 270
Column-gap: 266
Column-rule: 266
Column-span: 277
Column-width: 262
Columns: 262
Container Queries: 12, 52
CSS frameworky: 37

D

DevTools: 372
Display: 378
Distributed Alignment: 292

E

Env(): 404
Explicitní mřížka: 180, 198

F

Fab Four: 72
Fallback: 27, 61
Fallback alignment: 288
Feature Queries: 425
Firefox: 12
Fit-content(): 240
Flex (zkratka): 153
Flex-basis: 146, 355
Flex-direction: 135
Flex-flow: 138
Flex-grow: 140
Flex-shrink: 143
Flex-wrap: 133
Flexbox: 16, 25, 126
Float: 35, 186
Formátovací kontext: 171, 185, 378
Fr (jednotka): 200, 229
Frame: 29, 35

G

Gap: 178, 295, 320
Grid: 16, 25, 158
Grid (zkratka): 219
Grid area: 177
Grid cell: 176

Grid column: 176
Grid item: 171
Grid line: 175
Grid row: 176
Grid track: 176
Grid-area: 227
Grid-auto-columns: 211
Grid-auto-flow: 215
Grid-auto-rows: 211
Grid-column: 222
Grid-row: 222
Grid-template: 208
Grid-template-areas: 203
Grid-template-columns: 197
Grid-template-rows: 197
Gutter: 320

H

Historie: 29
Hlavní osa: 286

I

Implicitní mřížka: 180, 204
Inline osa: 286, 388
Internet Explorer (MSIE): 12, 27, 188, 346, 351

J

Jednotky: 229, 395
Justify-content: 327
Justify-items: 296
Justify-self: 307
Justify-tracks: 251

K

Kontejner mřížky: 171
Kontejner zarovnání: 287
Krkavčí technika: 121

L

Linka mřížky: 175
Logické vlastnosti: 386

M

Masonry: 202, 247
Masonry-auto-flow: 250
Max(): 414
Max-content: 212, 237
Media Object: 45, 55
Media Queries: 42
Mezera v layoutu: 178, 295, 320
Microsoft: 32
Min(): 414
Min-content: 212, 238
Minmax(): 236
Multicol: 16, 25, 29, 257, 357
Mřížka v gridu: 174

N

No Queries: 63, 116
Náhradní zarovnání: 288
Náhradní řešení: 27, 61

O

Oblast mřížky: 177, 204
Opakování v gridu: 233
Order: 340
Overflow Alignment: 294

P

Place-content: 337
Place-items: 305
Place-self: 318
Podpora v prohlížečích: 12, 351
Pojmenované linky: 201
Položka mřížky: 171
Positional Alignment: 288
Poziční zarovnání: 288
Pořadí v layoutu: 218, 340, 361
Prefixy: 427
Progressive Enhancement: 429
Prohlížeče: 12, 372
Proměnné: 403
Předmět zarovnání: 287
Přístupnost: 218, 343, 361
Příčná osa: 286

R

Repeat(): 201, 233
Responzivní design: 45
Rychlost vykreslení: 366

S

Safari: 12
Sloupec mřížky: 176
Směr layoutu: 215, 389
Span (rozsah v mřížce): 224
Stopa mřížky: 176
Subgrid: 82, 243
Svatý grál: 83

T

Table: 30, 35

V

Var(): 403
Vertical-align: 186

Z

Zahuštěné vykreslování: 216
Zarovnání boxů: 16, 281, 359
Zarovnání na účaří: 291
Zarovnání pro přetečení: 294
Zarovnání prostoru: 292

Martin Michálek
CSS: MODERNÍ LAYOUT

Redakce a jazyková korektura: Petr Jediný Novotný, pjnovotny.cz
Obrázky, schémata a vnitřní sazba: Martin Michálek
Obálka a spolupráce na grafice knihy: Petr Šťastný, raist.cz
Spolupráce na webu: Jan Kočiš, SUPERKODERS
Technická pomoc s e-booky: Lenka Silná
Foto autora: Oldřich Hrb, oldrichhrb.cz

V knize jsou použita písma Capita a Foro od Dietera Hofrichtera (hoftype.com) a InconsolataGo od Rapha Leviena (levien.com).

Tisk: PBtisk a.s.

Vydal Martin Michálek – Vzhůru dolů

Verze 1.1, říjen 2022

ISBN:
Tištěná kniha: 978-80-88253-07-5
PDF: 978-80-88253-08-2
MOBI: 978-80-88253-09-9
ePUB: 978-80-88253-10-5