Raphaël Goetter

CSS 3 FLEXBOX

Plongez dans les CSS modernes

EYROLLES

CSS 3
FLEXBOX

Plongez dans les CSS modernes

DANS LA MÊME COLLECTION

C. Blaess. – **Solutions temps réel sous Linux.**
N°14208, 2015, 300 pages.

W. McKinney. – **Analyse de données en Python.**
N°14109, 2015, 488 pages.

E. Biernat, M. Lutz. – **Data science : fondamentaux et études de cas.**
N°14243, 2015, 312 pages.

B. Philibert. – **Bootstrap 3 : le framework 100 % web design.**
N°14132, 2015, 318 pages.

C. Camin. – **Développer avec Symfony2.**
N°14131, 2015, 474 pages.

S. Pittion, B. Siebman. – **Applications mobiles avec Cordova et PhoneGap.**
N°14052, 2015, 184 pages.

H. Giraudel, R. Goetter. – **CSS 3 : pratique du design web.**
N°14023, 2015, 372 pages.

C. Delannoy. – **Le guide complet du langage C.**
N°14012, 2014, 844 pages.

K. Ayari. – **Scripting avancé avec Windows PowerShell.**
N°13788, 2013, 358 pages.

W. Bories, O. Mirial, S. Papp. – **Déploiement et migration Windows 8.**
N°13645, 2013, 480 pages.

W. Bories, A. Laachir, D. Thiblemont, P. Lafeil, F.-X. Vitrant. – **Virtualisation du poste de travail Windows 7 et 8 avec Windows Server 2012.**
N°13644, 2013, 218 pages.

J.-M. Defrance. – **jQuery-Ajax avec PHP.**
N°13720, 4e édition, 2013, 488 pages.

L.-G. Morand, L. Vo Van, A. Zanchetta. – **Développement Windows 8 - Créer des applications pour le Windows Store.**
N°13643, 2013, 284 pages.

Y. Gabory, N. Ferrari, T. Petillon. – **Django avancé.**
N°13415, 2013, 402 pages.

P. Roques. – **Modélisation de systèmes complexes avec SysML.**
N°13641, 2013, 188 pages.

SUR LE MÊME THÈME

H. Giraudel, R. Goetter. – **CSS 3 - Pratique du design web.**
N°14023, 2015, 354 pages.

S. Pollet-villard. – **Créer un seul site pour toutes les plates-formes.**
N°13986, 2014, 144 pages.

E. Marcotte. – **Responsive web design.**
N°13331, 2011, 160 pages.

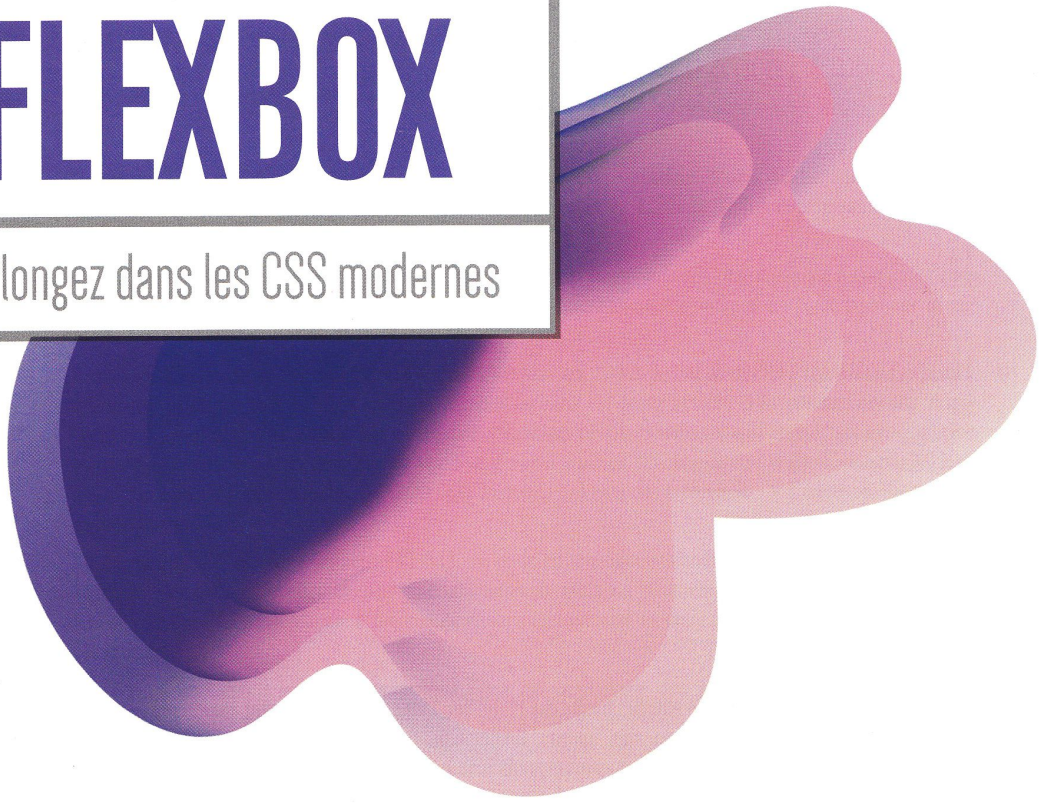
F. Draillard. – **Premiers pas en CSS 3 et HTML 5.**
N°13944, 6e édition, 2015, 472 pages.

M. Kabab, R. Goetter. – **Sass et Compass avancé.**
N°13677, 2013, 280 pages.

Raphaël Goetter

CSS 3 FLEXBOX

Plongez dans les CSS modernes



EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Avant-propos

En mémoire de mon très cher et regretté IE6…

Bonjour, je m'appelle Raphaël, je suis intégrateur et je souffre d'une addiction délicieuse : celle de toujours vouloir explorer de nouvelles contrées en CSS et de les appliquer en production lorsque cela est possible.

Par ailleurs et depuis plus de dix années maintenant, je prends un doux plaisir à partager mes connaissances sur le site web de la communauté Alsacreations.com afin d'en faire profiter librement ceux qui le souhaitent.

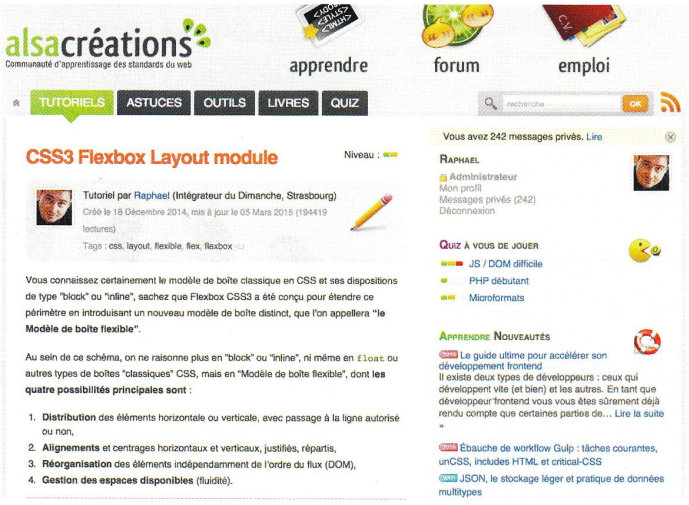
Je suis officiellement tombé amoureux de Flexbox en 2012. Je lui ai déclaré ma flamme publiquement quelque temps plus tard en lui consacrant un long article d'introduction sur Alsacréations, emballé que j'étais par cette spécification révolutionnaire qui remettait en question toutes mes années de bidouilles ayant forgé mon métier d'intégrateur web. Cet article, à ce jour, ne compte pas moins de 200 000 lectures ! Vous le trouverez à l'adresse raccourcie kiwi.gg/flexbox.

Le langage CSS et son pendant HTML se sont énormément étoffés au cours de ces dernières années. Les constructeurs de navigateurs sont de plus en plus enclins à suivre des spécifications universelles, les vitesses de connexion augmentent et les usages changent et se complexifient.

HTML 5 et CSS 3 permettent aujourd'hui de concevoir des applications web complexes, capables de gérer la géolocalisation, l'état de sa bande passante, des notifications et j'en passe. Nous sommes bien loin de l'époque des *framesets* ou des tableaux de mise en forme. Euh… quoique !

Aujourd'hui, CSS s'est constitué en un ensemble de règles et documentations tellement vaste et alambiqué qu'on peut écrire des ouvrages entiers sur une seule de ses spécifications ! La preuve ;)

Figure 1
Article « Flexbox »
sur Alsacréations



Flexbox, le modèle de boîte flexible

Ma passion pour Flexbox est devenue une évidence au moment même où je l'ai testé pour la première fois. À lui seul, ce mode de positionnement rend élémentaires tous les problèmes classiques rencontrés avec CSS depuis sa naissance :

- les alignements rendus simplissimes ;
- le centrage vertical ;
- une fluidité naturelle des éléments ;
- des hauteurs identiques entre frères ;
- la modification de l'ordre d'affichage ;
- et ce n'est qu'un début !

« **Flexible Box Layout Module** », mieux connue sous le nom de « Flexbox », est une spécification CSS 3 du W3C qui définit un nouveau modèle de boîte et de positionnement jusqu'alors inédit.

Flexbox intègre une gestion naturelle de la fluidité des éléments et du *responsive*, et rend caduc l'usage de grilles d'affichage complexes, voire de *frameworks* usines à gaz où l'on n'exploite que 10 % à peine de l'outil.

Flexbox est déjà en train de révolutionner de manière profonde et pérenne notre façon de concevoir des designs et des composants en CSS.

Bref, oubliez tout ce que vous aviez appris sur CSS et, comme moi, tombez amoureux de Flexbox, là maintenant tout de suite !

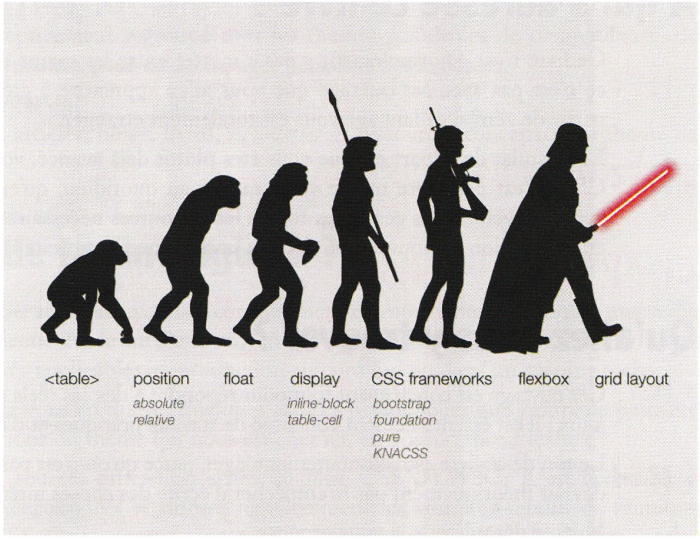


Figure 2
Le design CSS en quelques étapes clés : tables, floats, inline-blocks, frameworks, Flexbox, grid

Que va vous apporter Flexbox ?

Flexbox a été pensé et optimisé pour faire table rase de toutes les techniques bancales historiques de positionnement et des contournements de propriétés qui pullulent dans nos projets web.

Les « anciennes » méthodes encore (mal) utilisées de nos jours pour aligner ou placer des éléments ne sont souvent rien d'autre que du bricolage empirique : « tiens, à quoi peut bien servir ce *position : relative* ? », « pas grave, je mets une classe *.clearfix* partout ! », « oh ! mais pourquoi ça ne veut pas rentrer ? », etc.

Flexbox est conçu pour mettre de l'ordre dans tout ce maelström de bidouilles et revenir enfin aux bases d'un positionnement propre et adapté aux besoins actuels.

Cela ne signifie pas que nous allons bannir du jour au lendemain toutes nos déclarations de `float`, de `display : inline-block` ou de `display : table-cell`, mais simplement que nous allons retrouver leur usage initial, par exemple celui de `float` étant de placer une image à droite ou à gauche de son flot de contenu textuel alentour. Bref, ils demeureront un formidable complément à Flexbox.

Outre la simplification des schémas de positionnement, Flexbox apporte des solutions parfaites à une problématique bien ancrée dans notre époque : le *Responsive Webdesign*. Le design d'éléments flexibles, la réorganisation des blocs ainsi que la faculté à basculer très aisément d'un mode d'affichage horizontal vers un mode vertical en font un allié formidable dans nos projets d'adaptation aux tablettes et smartphones.

À qui s'adresse ce livre ?

Ce livre n'est « normalement » pas à mettre entre les mains de débutants. Plus précisément, ce n'est pas avec cet ouvrage que vous allez apprendre à créer des feuilles de styles CSS à partir de zéro si ce langage vous est totalement étranger.

Le postulat de départ est que vous êtes plutôt déjà avancé, voire expert, dans le domaine des CSS. C'est peut-être même votre métier au quotidien, qu'en sais-je. Si tel est le cas, alors vous trouverez dans ces pages toutes les ressources nécessaires pour appréhender et maîtriser en production ce nouveau modèle de positionnement excitant.

Qu'allez-vous y trouver ?

Cet ouvrage est conçu avant tout pour répondre à des cas réels de la vie d'intégrateur de tous les jours ; il est essentiellement constitué de travaux pratiques et d'exemples concrets décortiqués.

Le ton de ce livre est volontairement léger, parce qu'on n'est pas là pour s'ennuyer et que cela ne devrait théoriquement pas m'empêcher d'écrire des choses intéressantes. Je prends ce risque.

Voici le déroulé que je vous propose.

- **Chapitres 1 et 2 - Historique rapide et bases du design avec Flexbox**

  Les premiers chapitres aborderont les bases du modèle de boîte flexible, ainsi que l'ensemble des propriétés et leurs effets.
- **Chapitre 3 - Travaux pratiques : réordonner**

  Premier d'une longue série, un exercice concret et décortiqué consistant à modifier visuellement l'ordre d'affichage des éléments à l'écran.
- **Chapitre 4 - Trois astuces utiles**

  Quelques fonctionnalités très utiles au quotidien : occuper l'espace restant, centrer verticalement et pousser un élément ou un groupe d'éléments.
- **Chapitres 5, 6 et 7 - Travaux pratiques : une navigation *responsive*, un gabarit fluide et une galerie d'images**

  Un lot de travaux pratiques expliqués pour bien vous immerger dans Flexbox.
- **Chapitre 8 - Les principes fondamentaux**

  Les règles fondamentales de Flexbox, résumées en quelques points à conserver au fond de votre mémoire.
- **Chapitre 9 - Travaux pratiques : un formulaire fluide**

  Encore un exercice décortiqué, concernant cette fois les éléments de formulaires.
- **Chapitre 10 - La propriété flex en détail**

  Plongeon au cœur de la propriété la plus complexe de ce module, `flex`, et calcul des tailles finales des éléments fluides.
- **Chapitres 11 et 12 - Modèles de design et construction de grilles**

  Réflexions sur les méthodes de conception de grilles de mise en forme et applications pratiques.

- **Chapitres 13 à 15 - Encore plus loin, compatibilité, bogues des navigateurs**
  Partie où j'ai subrepticement regroupé tous les thèmes pas drôles de compatibilité, de bogues et de performances pour ne décourager personne dès le départ.
- **Chapitre 16 - Ressources**
  Sites de référence, articles, blogs, livres, *newsletter* et ressources concernant le thème de Flexbox.

Quid de votre veille technologique ?

Nous exerçons un métier merveilleux, mais constamment en mouvement. Pratiquer une veille technologique quotidienne est le meilleur moyen que nous ayons pour nous tenir à flot et informés des nouveautés applicables en production aujourd'hui ou dans un avenir proche.

Je vous livre, si le cœur vous en dit, les quelques astuces personnelles que j'emploie pour me tenir au fait des évolutions du thème qui nous est cher ici, en l'occurrence Flexbox.

Commençons par les sources officielles, celles du blog CSS du W3C. Il est accessible à l'adresse http://www.w3.org/blog/CSS/ et diffuse régulièrement les comptes-rendus de réunions du groupe de travail CSS. On y apprend toutes les décisions prises ainsi que les évolutions des spécifications sur des points très précis.

N'utilisant pas (plus) de client de flux RSS, j'ai néanmoins trouvé un excellent moyen de suivre les blogs et sites de *news* : l'outil ifttt.com. Grâce à IFTTT, je conçois mes « recettes » personnelles : par exemple, dès que le flux d'un de mes blogs préférés est rafraîchi, un *tweet* est automatiquement publié sur un compte dédié à la veille, `@goetter-veille`.

Parallèlement, j'effectue une recherche sur mon moteur préféré (Google), avec les quelques aménagements suivants :

- recherche : « flexbox » ;
- « pages en français » ;
- « moins d'une semaine ».

Cette manipulation très simple me permet d'obtenir des informations régulières et fraîches sur Flexbox. Et j'avoue que je n'aime toujours pas trop le service « Google Alertes » qui ne se débrouille pourtant pas mal du tout.

Figure 3
Google Search en action

Pour finir, j'utilise très (trop ?) fréquemment Twitter, plus particulièrement une recherche sauvegardée sur le terme « flexbox ». Je consulte cette recherche plusieurs fois dans la journée tant le fil est actif !

Si vous êtes twittophiles comme moi, je vous invite d'ailleurs à suivre cette liste non exhaustive de comptes, tous experts en CSS et en Flexbox :

- Lea Verou (en) : @LeaVerou (experte invitée W3C) ;
- Fantasai (en) : @fantasai (co-rédactrice des spécificationss Flexbox) ;
- Tab Atkins Jr (en) : @tabatkins (co-rédacteur des spécificationss Flexbox) ;
- Zoe Mickley Gillenwater (en) : @zomigi ;
- Rachel Andrew (en) : @rachelandrew ;
- Hugo Giraudel (en) : @HugoGiraudel ;
- Wes Bos (en) : @wesbos ;
- Vincent Valentin (fr) : @htmlvv ;
- Thomas Zilliox (fr) : @iamtzi ;
- Rémi Parmentier (fr) : @hteumeuleu ;
- Vincent de Oliveira (fr) : @iamvdo ;
- Nicolas Hoizey (fr) : @nhoizey.

Par ailleurs, Rachel Andrew publie régulièrement sur CSSlayoutnews les toutes dernières informations dédiées aux nouveautés sur les positionnements CSS : http://csslayout.news/issues. Même si je n'aime habituellement pas trop les newsletters, je trouve cette ressource excellente.

Sites web associés

Il vous est possible de bénéficier tout particulièrement de deux ressources en ligne associées à cet ouvrage :

- la première est le site dédié goetter.fr/livres/flexbox où vous trouverez tous les codes des travaux pratiques présents dans le livre, mais aussi les errata et diverses autres informations mises à jour ;
- la seconde ressource n'est autre que mon site personnel, goetter.fr. Il m'a servi de « Monstre de Frankenstein » sur lequel j'ai opéré toutes sortes d'expériences *flexboxesques* que vous pouvez décortiquer à loisir.

Pour ce qui est des exemples de codes présents au sein de ces pages, vous les trouverez également en ligne sur CodePen :

- les exemples basiques, consultables et modifiables, sont regroupés au sein d'une collection nommée « Livre Flexbox » à l'adresse suivante : http://codepen.io/collection/DOLBEQ/ ;
- les codes plus avancés (formulaires, gabarits, navigations) sont disponibles dans la collection « Jack in the Flexbox » : http://codepen.io/collection/njbNNQ/.

Je vous souhaite à présent une excellente lecture et une bonne mise en pratique de Flexbox.

Figure 4
Le site goetter.fr, laboratoire personnel de Flexbox

Table des matières

CHAPITRE 1
Une brève histoire de Flexbox 1
Une popularité grandissante 3
Un usage en production 3
Sites internationaux 5
Sites français 5

CHAPITRE 2
Les bases du design avec Flexbox 7
flex-container et flex-item 8
Propriétés sur le parent 8
display 8
flex-direction 9
flex-wrap 10
flex-flow 11
justify-content 12
align-items 13
align-content 14
Propriétés sur les enfants 15
order 15
align-self 16
flex-grow 17
flex-shrink 17
flex-basis 18
flex 19

CHAPITRE 3
TP : réordonner des éléments 21
Le code HTML 22
Une mission pour order ! 23
Les Media Queries en action 24

Limitations 24
N'en abusez pas ! 24

CHAPITRE 4
Trois astuces utiles 27
Occuper l'espace restant 27
Centrer verticalement 29
Pousser un élément 30

CHAPITRE 5
TP : une navigation Responsive 33
Le code HTML 34
Titre complet ou initiales ? 34
Écrans de smartphones 35
Grands écrans 36
Astuce : réordonner les éléments 37
Écrans de taille moyenne (tablettes) 37
Le truc en plus : la « variable » currentColor 38

CHAPITRE 6
TP : un gabarit simple 41
La structure HTML 42
Étape 1 : occuper toute la hauteur 43
Étape 2 : pied en bas de page 44
Étape 3 : navigation, aside et contenu principal 45
Étape 4 : le contenu en premier 46
C'est Responsive, ma bonne dame ! 46

CHAPITRE 7
TP : une galerie d'images 49
Le code HTML 50
Répartition horizontale 51
Explications 51
Limitations 52
Répartition verticale 53
Explications 53
Limitations 54
object-fit à la rescousse 55
Explications 56
Le truc en plus : object-position 56

Limitations 57
Effets au survol et focus 57

CHAPITRE 8
Les principes fondamentaux 59
Les trois principes fondamentaux de Flexbox 60
1 - De l'apparente simplicité de Flexbox 60
2 - De l'apparente omnipotence de Flexbox 60
3 - De l'apparente stabilité de Flexbox 60
Les cinq pense-bêtes de l'intégrateur 60
1 - min-width et max-width sont prioritaires sur tout 60
2 - flex-wrap est prioritaire sur flex-shrink 61
3 - flex-shrink et flex-grow sont prioritaires sur flex-basis 61
4 - flex-basis est prioritaire sur width 61
5 - Dimensions minimales intrinsèques 61
Récapitulatif des priorités 62

CHAPITRE 9
TP : un formulaire fluide 63
Objectif à atteindre 64
Le code HTML 64
Étape 1 : les deux de même hauteur 66
Étape 2 : les <input> de largeur restante 67
Étape 3 : le <textarea> de hauteur restante 67
Étape 4 : le bonus Responsive 68

CHAPITRE 10
La propriété flex en détail 71
Valeurs courantes de flex 71
Quelques mythes débusqués 72
Attention terrain glissant ! 73
L'espace restant 73
Calcul de flex-grow 74
La formule magique 74
flex-grow n'est pas flex ! 75
Quelques exemples concrets 75
Cas 1. Chaque enfant est doté de flex: 1 75
Cas 2. flex: 1 0 200px et flex: 1 0 400px 76
Cas 3. flex: 1 et flex-grow: 1 76
Cas 4. flex: 1 0 200px et width: 200px 76

Cas 5. Pas de flex et width: 200px 77
Calcul de flex-shrink 77
La formule magique 77
Des ressources bien pratiques 78

CHAPITRE 11
Modèles de design **79**
Répartition sur une ligne 80
Principe et objectif 80
Code nécessaire 80
Explications sommaires 80
Une gouttière ? 80
Question pour des champions comme vous 81
Répartition en forçant le retour à la ligne 81
La solution min-width 81
La solution width 82
Marges et gouttières 82
Le truc en plus : calc() 83
Automatiser en production 83
Répartition automatique (Autoflow) 84
Principe et objectif 84
Code nécessaire 84
Explications sommaires 85
Limitations de cette technique 85
L'astuce en plus 86

CHAPITRE 12
TP : construction de grilles **87**
Principes généraux 87
Des grilles, des grilles, des grilles 88
Frameworks CSS 88
Grilles autonomes 89
Les bases de la grille 89
Ébauche de grille en Flexbox 89
Gérer la gouttière 90
Variante avec :nth-child() 91
Automatiser avec un préprocesseur 92
Bonus : éléments double ou triple taille 93
Bonus : « à la Une » 93
Bonus : pull et push 94

Grille Responsive 95
Résultat final 96

CHAPITRE 13
Encore plus loin avec Flexbox 99
Des propriétés non appliquées 99
Un contexte de formatage particulier 100
Des pseudo-éléments embarrassants 100
Largeur minimale intrinsèque 101
Flexbox et z-index 101
Flexbox et les espaces en pourcentage 102
Flexbox et les animations 103
Flexbox et accessibilité 103
Quelques astuces pratiques 104
Décoration de titre 104
Points de conduite 105
Tunnel d'achat 105
Fenêtre modale 107

CHAPITRE 14
Performances et compatibilité 109
Performances de Flexbox 109
Compatibilité des navigateurs 110
Statistiques d'usage en France et dans le monde 111
Les outils et alternatives à Flexbox 111
Modernizr 111
Autoprefixer 112
Les préprocesseurs 114
Les polyfills 114
Bogues connus et solutions 115
La valeur par défaut de flex a changé (IE 10) 115
Bogue de min-height non appliqué (IE 10-IE 11) 116
Propriété flex non reconnue sur des éléments inline (IE 10-IE 11) 116
Pas de box-sizing sur flex-basis (IE 10-IE 11) 117
Pas de calc() sur flex (IE 10-IE 11) 118
max-width: 100% non appliqué sur les images ? 118

CHAPITRE 15
Flexbox contre Grid Layout ? 119
Les limites de Flexbox ? 119

Grid Layout 120
Le positionnement par grille 120
Les propriétés usuelles de Grid Layout 121
Mise en œuvre 121
Exemple 1 (affichage de deux blocs sur une ligne) 121
Exemple 2 (grille de 4 emplacements) 122
Variante : la syntaxe de templates 123
Exemple 3 (template) 123
Centrer les éléments 123
Exemple 4 (centrage multiple) 124
Occuper plusieurs emplacements 125
Exemple 5 (column span) 125
Et bien d'autres choses encore ! 126

CHAPITRE 16
Ressources utiles 127
Références 127
Tutoriels et articles 128
Vidéos 129
Conférences 129
Livres 129
Autres 129

ANNEXE
Mémo des propriétés 131

Index 133

1

Une brève histoire de Flexbox

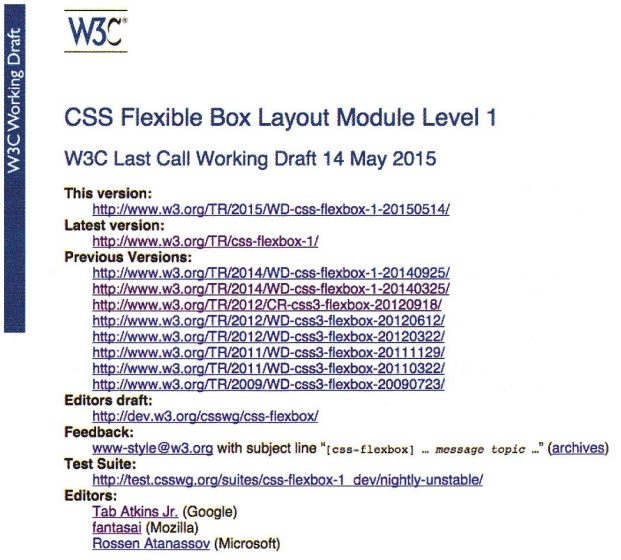
Plongez dans l'historique palpitant de Flexbox, rempli de trésors enfouis, de monstres poilus, de dragons et de trolls.

Les origines de Flexbox remontent à l'année 2003 lorsque Safari, le navigateur d'Apple sur Mac, alors en version 1.1, implémente son lointain ancêtre.

Mozilla Firefox 2.0 lui emboîte le pas dès 2006 (Flexbox est d'ailleurs basé sur XUL de Mozilla), puis Chrome et Android dès les premières ébauches de spécifications. Il a cependant fallu attendre Microsoft et sa version Internet Explorer 10, ainsi que les premiers véritables smartphones, pour voir ce module enfin démocratisé et véritablement employé en production.

Depuis son premier brouillon officiel au W3C en 2009 par David Baron de Mozilla, la spécification *Flexible Layout Module* est actuellement principalement maintenue par des éditeurs des sociétés Google, Mozilla et Microsoft.

Le document de spécification a été élevé au rang de *Candidate Recommandation* (quasi finalisé) en 2012, puis rétrogradé au stade de *Last Call Working Draft* (brouillon stable) deux années plus tard, étape indispensable dans la procédure W3C si l'on souhaite opérer des modifications d'implémentations, même minimes (ce qui fut le cas).


W3C Working Draft

W3C

CSS Flexible Box Layout Module Level 1

W3C Last Call Working Draft 14 May 2015

This version:
http://www.w3.org/TR/2015/WD-css-flexbox-1-20150514/
Latest version:
http://www.w3.org/TR/css-flexbox-1/
Previous Versions:
http://www.w3.org/TR/2014/WD-css-flexbox-1-20140925/
http://www.w3.org/TR/2014/WD-css-flexbox-1-20140325/
http://www.w3.org/TR/2012/CR-css3-flexbox-20120918/
http://www.w3.org/TR/2012/WD-css3-flexbox-20120612/
http://www.w3.org/TR/2012/WD-css3-flexbox-20120322/
http://www.w3.org/TR/2011/WD-css3-flexbox-20111129/
http://www.w3.org/TR/2011/WD-css3-flexbox-20110322/
http://www.w3.org/TR/2009/WD-css3-flexbox-20090723/
Editors draft:
http://dev.w3.org/csswg/css-flexbox/
Feedback:
www-style@w3.org with subject line "[css-flexbox] … message topic …" (archives)
Test Suite:
http://test.csswg.org/suites/css-flexbox-1_dev/nightly-unstable/
Editors:
Tab Atkins Jr. (Google)
fantasai (Mozilla)
Rossen Atanassov (Microsoft)

Figure 1–1
Le document de spécification Flexbox en 2015

L'adolescence de Flexbox a été pour le moins tumultueuse : avant de se stabiliser en 2012, il a connu trois grandes étapes importantes au cours de son existence ayant chacune conduit à des spécifications totalement différentes, du moins en termes de nommage des propriétés et des valeurs :

- le premier brouillon de juillet 2009, alors pris en charge par Firefox et Safari (également sur iPhone dès son origine) ;
- un second brouillon en mars 2012 (nommé parfois *2011 tweener syntax*), qu'Internet Explorer 10 a décidé d'implémenter ;
- la syntaxe en *Candidate Recommandation* de septembre 2012, remaniée ensuite en *Last Call Working Draft* en mars 2014.

Voilà pourquoi, aujourd'hui encore, nous risquons de rencontrer certaines valeurs historiques et désuètes dans nos feuilles de styles, ou dans celles que vous récupérerez au détour de recherches sur Internet, par exemple `display: -ms-flexbox` pour Internet Explorer 10, en lieu et place du standard `display: flex`.

J'en profite pour ajouter qu'à l'heure où cet ouvrage est rédigé, Flexbox est une spécification CSS… *Level 1*. Hein, quoi, on m'aurait menti ?! Du vulgaire CSS 1 ? C'est un peu plus compliqué que cela : après la finalisation de CSS 2.1, devenu *Recommandation Officielle* en 2011, les nouvelles fonctionnalités de CSS étaient tellement nombreuses, grouillantes et expérimentales qu'elles ont mené à un nouveau processus de normalisation du W3C. Il n'était plus question de raisonner en termes de version globale ; au contraire, chaque module spécifique progresse dorénavant à son propre rythme et se voit associé à un *niveau (level)*.

Cela signifie que *CSS 3* (et encore moins *CSS 4*) n'existe pas, ou plutôt tout est *CSS 3* depuis la finalisation de CSS 2.1. Une nouvelle fonctionnalité, comme c'est le cas pour Flexbox ou Grid Layout, sera associée au *level 1*, tandis que les anciennes spécifications peuvent se situer au *level 3* (polices, modèle de boîte). Enfin, certains modules tels que les sélecteurs ont déjà atteint le *level 4* car le niveau précédent était finalisé.

Bref, Flexbox a un bon goût de CSS 3 mais son stade officiel est aujourd'hui *CSS level 1* et un brouillon de *level 2* est déjà en cours d'écriture.

Une popularité grandissante

L'avènement d'Internet Explorer 10 ainsi que les smartphones récents, plutôt à l'aise avec Flexbox, lui ont conféré une popularité indéniable depuis quelques années. Preuve en est, les innombrables démonstrations publiques d'astuces Flexbox sur le site CodePen : chaque jour, de nouvelles prouesses techniques y sont répertoriées.

- http://codepen.io/search?q=flexbox&limit=all&order=popularity&depth=everything&show_forks=false

Par ailleurs, même si cela est plus anecdotique car purement honorifique, apprenez que Flexbox Layout Module a été promu « Technologie web de l'année 2013 » par le magazine .net.

- http://www.glazman.org/weblog/dotclear/index.php?post/2013/06/10/CSS-Flexible-Box-is-Best-New-Web-Technology-2013!

Figure 1–2
Flexbox récompensé en 2013 par .net

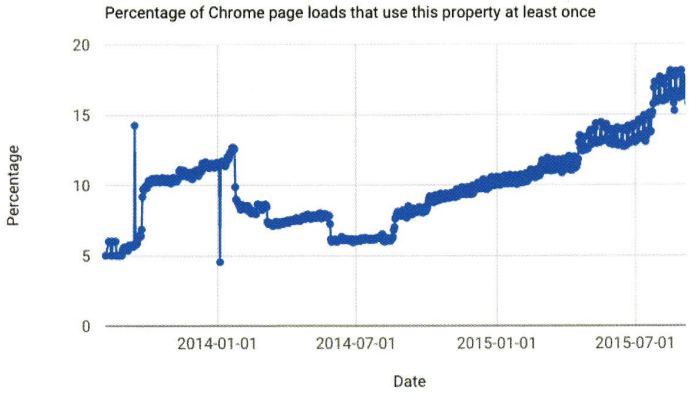
Un usage en production

Le module Flexbox n'est pas seulement une vitrine de démonstration ou un jouet à la mode. Il est également un outil formidablement pratique déjà employé en production sur des projets concrets, parfois d'envergure.

Le site web Chrome Platform Status propose des graphiques et des statistiques fort intéressants concernant l'usage des propriétés CSS au cours du temps. Ainsi, nous découvrons que la propriété `flex` est présente sur au moins 15 % des pages web en 2015.

> http://bit.ly/flexbox-usage

Figure 1–3
Évolution de l'usage de la propriété flex au sein des pages web



D'autres sources de statistiques, notamment relayées par le site de référence Caniuse.com, nous apprennent qu'à la fin de l'année 2015, plus de 95 % des navigateurs employés en France (et dans le reste du monde) appliquent les propriétés de Flexbox.

Figure 1–4
Compatibilité de Flexbox à la fin de l'année 2015

L'usage en production de Flexbox ne se limite pas aux sites personnels, aux blogs ni aux petites agences web. Figurez-vous que de plus en plus de gros sites web internationaux passent le cap de la modernité.

En voici une liste, loin d'être exhaustive…

Sites internationaux

- twitter.com ;
- nytimes.com ;
- flickr.com ;
- booking.com ;
- mappy.com ;
- ft.com ;
- walmart.com ;
- washingtonpost.com ;
- usatoday.com ;
- mailchimp.com ;
- weather.com ;
- codepen.io ;
- cma-cgm.com...

Sites français

- liberation.fr ;
- paris.fr ;
- canalplay.com ;
- 6play.fr ;
- roquefort-societe.com...

Du côté de mon agence web Alsacreations.fr, sachez que nous avons officiellement abandonné Internet Explorer 9 en début d'année 2015 et que Flexbox fait partie de notre quotidien depuis un petit moment déjà.

Cela va même un peu plus loin puisque, dans les formations que nous dispensons à notre public de professionnels, nous incitons de plus en plus fréquemment à employer Flexbox comme base de positionnement avant même de livrer bataille aux complexes `float` et `position`.

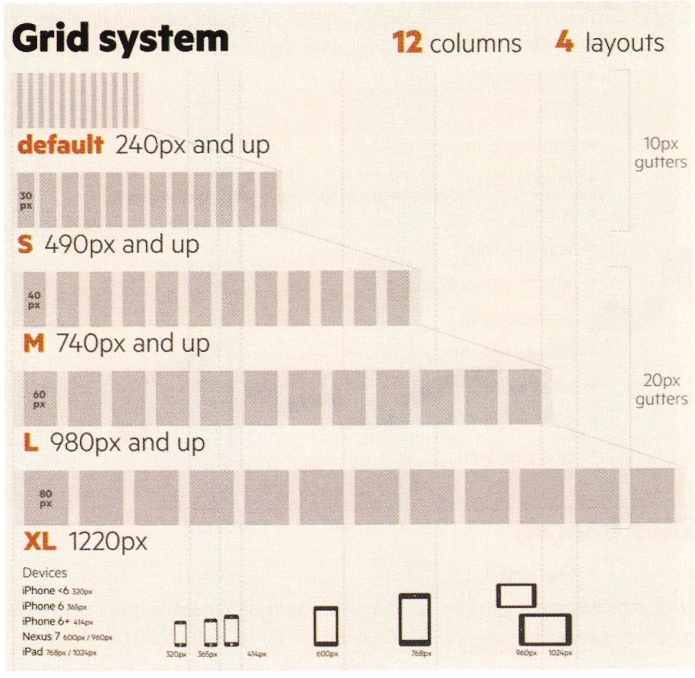


Figure 1–5
Outil de grille Flexbox du *Financial Times*. Source : https://github.com/Financial-Times/o-grid

2

Les bases du design avec Flexbox

Où l'on apprend que Flexbox est un compagnon relativement facile à adopter et que « row-reverse » n'est pas le frère de Dick...

Flexbox introduit un nouveau modèle de boîte, avec ses règles et fonctionnalités différentes du modèle de boîte historique en CSS (celui que l'on génère avec des `display: block`, `inline`, `float`, etc.).

Figure 2–1
Un exemple de menu fluide avec Flexbox

Vous connaissez certainement le modèle de boîte classique en CSS et ses dispositions de type `block` ou `inline`. Sachez que Flexbox CSS 3 a été conçu pour étendre ce périmètre en introduisant un nouveau modèle, que l'on appellera si vous le voulez bien le *modèle de boîte flexible*.

Au sein de ce schéma, on ne raisonne plus en `block` ou `inline`, ni même en `float` ou autres types de boîtes classiques CSS, mais en « contexte de Flexbox », dont les quatre possibilités principales sont :

- distribution des éléments : horizontale ou verticale, avec passage à la ligne autorisé ou non ;
- alignements et centrages horizontaux et verticaux, justifiés, répartis ;
- réorganisation des éléments indépendamment de l'ordre du flux (DOM) ;
- gestion des espaces disponibles (fluidité).

flex-container et flex-item

Flexbox (ce modèle de boîte flexible, donc) se fonde schématiquement sur une architecture de ce type :

- un flex-container créant le contexte général d'affichage pour ses enfants ;
- des flex-item qui ne sont rien d'autre que les enfants directs du conteneur, quels qu'ils soient.

Le flex-container, qui définit l'environnement global du modèle de boîte flexible, est tout simplement n'importe quel élément HTML doté de la déclaration `display: flex` ou `display: inline-flex`.

Ses enfants deviennent alors automatiquement (inutile de leur déclarer quoi que ce soit) des éléments de type flex-item qui n'occupent initialement que la taille minimale de leur contenu.

Propriétés sur le parent

Passons en revue l'ensemble des propriétés liées au flex-container ainsi que leur usage.

display

La première propriété, et non des moindres, est tout simplement `display`. Sachez qu'elle s'est vue étoffée depuis CSS 3 ; elle compte à présent une bonne vingtaine de valeurs possibles, dont celles qui nous intéressent tout particulièrement pour Flexbox.

Tableau 2–1 Valeurs Flexbox de display

Propriété et valeur	Effet sur l'élément
`display: flex`	L'élément devient un flex-container et se comporte comme un `block`.
`display: inline-flex`	L'élément devient un flex-container et se comporte comme un `inline-block`.

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

Figure 2–2
Un conteneur muni d'un display: flex

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

Figure 2–3
Deux conteneurs munis d'un display: inline-flex

> **REMARQUE Alignement vertical**
>
> Les éléments `inline-flex`, de la même manière que les éléments en `display: inline` ou `display: inline-block`, peuvent s'aligner verticalement entre eux à l'aide de la propriété `vertical-align`.

flex-direction

La distribution, c'est-à-dire le sens d'affichage horizontal ou vertical des éléments flex-item, est définie par la propriété `flex-direction` dont les valeurs possibles sont listées dans le tableau 2-2.

Tableau 2–2 Valeurs de flex-direction

Propriété et valeur	Effet sur l'élément
`flex-direction: row`	Distribution horizontale des flex-item, valeur par défaut.
`flex-direction: row-reverse`	Distribution horizontale inversée.
`flex-direction: column`	Distribution verticale.
`flex-direction: column-reverse`	Distribution verticale inversée.

Cette propriété s'applique au flex-container et détermine l'axe principal du contexte de boîte flexible.

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

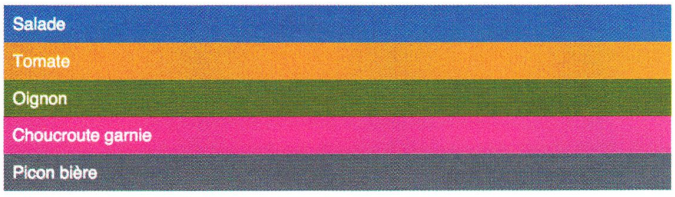
Figure 2–4
Un flex-container
avec flex-direction: row

```
.container {
  display: flex;
  flex-direction: row-reverse;
}
```

Figure 2–5
Un flex-container
avec flex-direction: row-reverse

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

Figure 2–6
Un flex-container avec flex-direction: column




```
.container {
  display: flex;
  flex-direction: column-reverse;
}
```

Figure 2–7
Un flex-container avec flex-direction: column-reverse

Pour rappel, la valeur par défaut de `flex-direction` est `row`.

flex-wrap

La propriété `flex-wrap` définit si le contenu sera distribué sur une seule ligne (ou colonne selon l'axe principal) ou sur plusieurs lignes, en clair, si les flex-item ont le droit de passer à la ligne ou non.

Tableau 2–3 Valeurs de flex-wrap

Propriété et valeur	Effet sur l'élément
`flex-wrap: nowrap`	Les éléments flex-item ne passent pas à la ligne, valeur par défaut.
`flex-wrap: wrap`	Les éléments passent à la ligne.
`flex-wrap: wrap-reverse`	Les éléments passent à la ligne dans le sens inverse.

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

```
  flex-wrap: wrap-reverse;
}
```

Figure 2–8
nowrap, wrap et wrap-reverse pour une direction row

Figure 2–9
nowrap et wrap pour une direction column

Pour rappel, la valeur par défaut de `flex-wrap` est `nowrap`, c'est-à-dire que les éléments ne passent pas naturellement à la ligne.

Notez que `flex-wrap` n'a été reconnu que tardivement par certains navigateurs, par exemple Android qui ne l'a pris en charge qu'à partir de sa version 4.4.

flex-flow

`flex-flow` est une propriété raccourcie qui regroupe les deux propriétés `flex-direction` et `flex-wrap`.

Pour des raisons de compatibilité, je vous invite à préférer pour quelque temps encore les propriétés séparées. En effet, `flex-flow` n'est reconnue qu'à partir de Firefox 28 et Safari 7.1 par exemple.

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

Figure 2–10
Un conteneur muni de flex-flow: column wrap

Sans surprise, la valeur par défaut de `flex-flow` est `row nowrap`.

justify-content

Flexbox propose de gérer très finement les alignements et centrages, en différenciant les deux axes d'affichage de la manière suivante :

- l'alignement dans l'axe principal est traité via la propriété `justify-content` ;
- l'alignement dans l'axe secondaire est géré avec `align-items`.

Les alignements dans l'axe de lecture principal, c'est-à-dire celui que vous avez défini à l'aide de la propriété `flex-direction`, sont définis à l'aide de la propriété `justify-content`, dont les valeurs possibles sont listées dans le tableau 2-4.

Tableau 2–4 Valeurs de justify-content

Propriété et valeur	Effet sur l'élément
`justify-content: flex-start`	Éléments positionnés au début de l'axe principal, valeur par défaut.
`justify-content: flex-end`	Éléments positionnés à la fin de l'axe principal.
`justify-content: center`	Éléments centrés dans l'axe principal.
`justify-content: space-between`	Éléments répartis de façon « justifiée ».
`justify-content: space-around`	Variante de répartition « justifiée ».

Figure 2–11
Un conteneur muni de justify-content: flex-start et flex-direction: row

Figure 2–12
Un conteneur muni de justify-content: flex-end et flex-direction: row

Figure 2–13
Un conteneur muni de justify-content: center et flex-direction: row

Figure 2–14
Un conteneur muni de justify-content: space-between et flex-direction: row

Figure 2–15
Un conteneur muni de justify-content: space-around et flex-direction: row

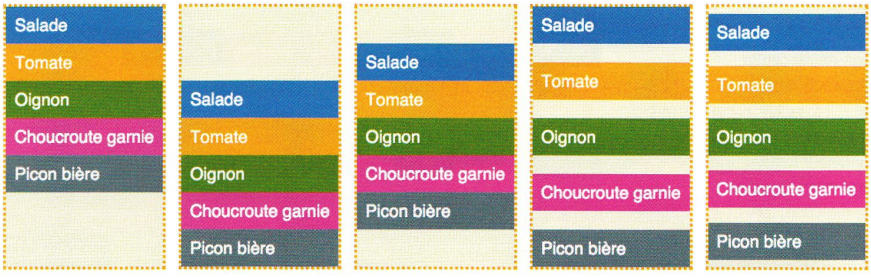


Figure 2–16 Les mêmes alignements au sein d'un contexte en flex-direction: column

Pour rappel, la valeur par défaut de `justify-content` est `flex-start`.

align-items

Dans l'axe secondaire, les alignements sont régis via la propriété `align-items`, dont les valeurs sont listées dans le tableau 2-5.

Tableau 2–5 Valeurs de align-items

Propriété et valeur	Effet sur l'élément
`align-items: stretch`	Éléments flex-item étirés dans l'espace disponible, valeur par défaut.
`align-items: flex-start`	Éléments disposés au début de l'axe secondaire.
`align-items: flex-end`	Éléments disposés à la fin de l'axe secondaire.
`align-items: center`	Éléments centrés dans l'axe secondaire.
`align-items: baseline`	Éléments alignés sur la ligne de base du texte (`line-height`), généralement identique à `flex-start` si la taille de police est uniforme.

Figure 2–17
Un conteneur muni de align-items: stretch et flex-direction: row

Figure 2–18
Un conteneur muni de align-items: flex-start et flex-direction: row

Figure 2–19
Un conteneur muni de align-items: flex-end et flex-direction: row

Figure 2–20
Un conteneur muni de align-items: center et flex-direction: row

Figure 2–21
Un conteneur muni de align-items: baseline et flex-direction: row

Figure 2–22 Les mêmes alignements au sein d'un contexte en flex-direction: column

Pour rappel, la valeur par défaut de `align-items` est `stretch` : les flex-item occupent donc tout l'espace disponible verticalement ou horizontalement sur l'axe secondaire selon la direction définie.

align-content

La propriété `align-content` s'applique uniquement sur un flex-container autorisant le passage à la ligne (`flex-wrap`).

Elle définit l'alignement des rangées, les centre ou les répartit au sein de l'espace restant dans l'axe secondaire.

Tableau 2–6 Valeurs de align-content

Propriété et valeur	Effet sur l'élément
`align-content: stretch`	Les rangées occupent tout l'espace disponible, valeur par défaut.
`align-content: flex-start`	Les rangées s'affichent au début du flex-container.
`align-content: flex-end`	Les rangées s'affichent à la fin du flex-container.
`align-content: center`	Les rangées s'affichent au centre du flex-container.
`align-content: space-between`	L'espace entre les rangées est distribué de façon « justifiée ».
`align-content: space-around`	Variante de `space-between`.

Figure 2–23 Les valeurs stretch, flex-start et flex-end au sein d'un parent en flex-direction: row

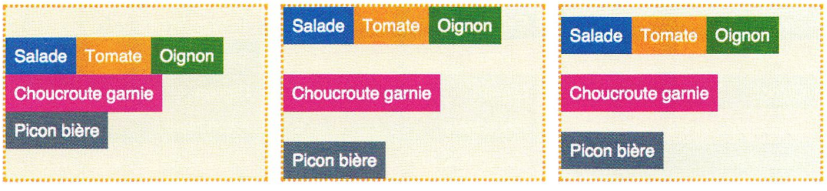


Figure 2–24 Les valeurs center, space-between et space-around au sein d'un parent en flex-direction: row

Pour rappel, la valeur par défaut de `align-content` est `stretch`, chaque rangée d'éléments faisant en sorte d'occuper équitablement l'espace restant.

Propriétés sur les enfants

Dès lors qu'un élément devient un flex-item, c'est-à-dire quand son parent définit un contexte Flexbox, il est radicalement transformé et n'est plus considéré comme un `block` ou un `inline` classique (d'ailleurs les valeurs de `display` autres que `none` et même certaines propriétés telles que `float` n'ont plus d'effet sur lui).

Il est alors possible de lui attribuer certaines propriétés particulières à son nouveau statut de flex-item.

order

L'une des fonctionnalités les plus avant-gardistes du modèle d'affichage Flexbox est de pouvoir réordonner à sa guise chacun des éléments indépendamment, grâce à la propriété `order`.

`order` ordonne les éléments flex-item entre eux. Sa valeur est un nombre entier qui agit telle une pondération : les éléments dont la valeur est la plus forte se trouveront en bas de la pile.

La valeur initiale de `order` est `0`. Les éléments ayant une valeur négative s'afficheront en priorité ; ceux ayant plus que zéro s'afficheront à la fin.

Plusieurs éléments peuvent bénéficier de la même valeur ; ils s'afficheront ensemble dans l'ordre de leur apparition dans le code HTML.

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

```
.choucroute {
  order: 1;
}
.tomate {
  order: -1;
}
```

Dans ce petit extrait de code, l'élément `.choucroute` s'affichera à la suite de ses frères et `.tomate` au début.

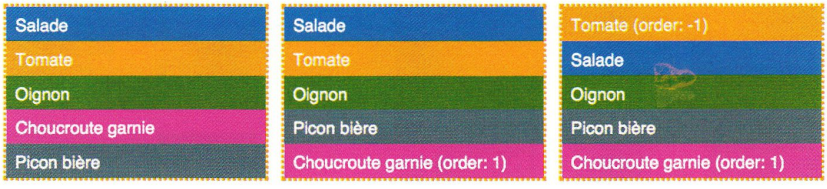


Figure 2–25 Réordonner à l'aide de la propriété order

Au sein d'un flex-container, les éléments enfants positionnés en absolu (`position: absolute`) sont également affectés par la propriété `order`, bien qu'ils ne comptent pas parmi les flex-item.

Pour rappel, la valeur par défaut de `order` est `0`.

align-self

La propriété `align-self` permet de distinguer l'alignement d'un flex-item de ses frères. Les valeurs de cette propriété sont identiques à celles de `align-items` à l'exception de `auto`.

Tableau 2–7 Valeurs de align-self

Propriété et valeur	Effet sur l'élément
`align-self: auto`	L'élément flex-item adopte l'alignement général défini via `align-items`, valeur par défaut.
`align-self: flex-start`	L'élément est aligné au début de l'axe secondaire.
`align-self: flex-end`	L'élément est aligné à la fin de l'axe secondaire.
`align-self: center`	L'élément est centré dans l'axe secondaire.
`align-self: baseline`	L'élément est aligné sur la ligne de base du texte (`line-height`).
`align-self: stretch`	L'élément est étiré pour occuper tout l'espace disponible.

Figure 2–26
L'élément « choucroute » est ici en align-self: flex-end au sein d'un parent en align-items: stretch.

Pour rappel, la valeur par défaut de `align-self` est `auto`.

flex-grow

Basiquement, la propriété `flex-grow` permet à un enfant flex-item de s'élargir pour occuper une proportion de l'espace libre restant.

La valeur que vous pouvez appliquer à cette propriété est un nombre qui exprime le « facteur d'agrandissement » de l'élément.

Par défaut, `flex-grow` vaut `0`. Pour rendre un élément flexible, il suffit de lui attribuer une valeur supérieure à zéro. Cet élément s'appropriera alors l'espace restant au sein de son conteneur.

Plusieurs éléments peuvent être rendus flexibles et se répartir l'espace restant. L'espace disponible est alors tout simplement distribué entre les éléments flexibles, réparti selon leur facteur d'agrandissement.

```
.salade {
  flex-grow: 1; /* je peux m'agrandir */
}
.tomate {
  flex-grow: 2; /* moi aussi, et plus que toi ! */
}
.oignon {
  flex-grow: 0; /* erf, pas moi :( */
}
```

Figure 2–27
Trois éléments pour lesquels flex-grow vaut respectivement 1, 2 et 0

Pour rappel, la valeur par défaut de `flex-grow` est `0`.

flex-shrink

La propriété `flex-shrink` exprime le « facteur de rétrécissement » d'un élément ; elle permet à un enfant flex-item de se compresser automatiquement s'il n'a pas d'espace suffisant au sein de son conteneur.

La valeur par défaut de `flex-shrink` est `1` qui signifie que les flex-item peuvent naturellement se rétrécir si nécessaire et tant qu'on ne leur dit pas le contraire.

```
.salade {
  flex-shrink: 1; /* je peux me rétrécir */
}
.tomate {
  /* moi aussi, puisque la valeur par défaut est 1 */
}
.oignon {
  flex-shrink: 0; /* erf, pas moi :( */
}
```

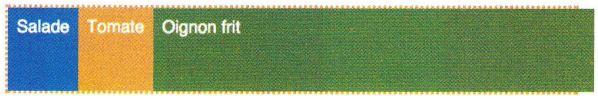


Figure 2–28
Trois éléments dont la valeur de width est 500px et dont flex-shrink vaut respectivement 1, rien et 0

flex-basis

La propriété `flex-basis` est la taille indicative (largeur ou hauteur selon l'axe) d'un flex-item avant d'appliquer les éventuels facteurs d'agrandissement ou de rétrécissement que l'on vient de parcourir.

En clair, si `flex-grow` et `flex-shrink` ont pour valeur `0`, alors `flex-basis` détermine la taille de l'élément. Enfin, euh… tant qu'il n'entre pas en conflit avec `min-width` ou `max-width`.

Tableau 2–8 Valeurs habituelles de flex-basis

Propriété et valeur	Effet sur l'élément
`flex-basis: auto`	La taille de l'élément est définie par son contenu (valeur par défaut).
`flex-basis: valeur`	La hauteur ou largeur de l'élément correspond à la valeur indiquée.
`flex-basis: 0`	La taille de l'élément devrait théoriquement être nulle, mais ne peut se réduire en deçà de son contenu minimal insécable.

Dans le cas où `flex-grow` ou `flex-shrink` a une valeur non nulle, alors `flex-basis` devient une taille indicative et représente plutôt une taille minimale ou maximale de l'élément.

`flex-shrink`, quelle que soit sa valeur, entre dans le calcul de l'« espace restant » au sein du flex-container et dans la distribution de cet espace restant. Nous traiterons cette notion en détail un peu plus tard, promis.

```
.salade {
  flex-grow: 0;
  flex-shrink: 0;
  flex-basis: 50%; /* je mesure exactement 50 % */
}
.tomate {
  flex-grow: 1;
  flex-shrink: 0;
  flex-basis: 50%; /* je mesure au moins 50 % */
}
.oignon {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 50%; /* ben moi je ne sais pas trop :( */
}
```

Pour rappel, la valeur par défaut de `flex-basis` est `auto`, ce qui correspond à la taille minimale du contenu.

Figure 2–29
Illustration du code précédent, combinant flex-grow, flex-shrink et flex-basis

flex

La propriété `flex` est un raccourci des trois propriétés `flex-grow`, `flex-shrink` et `flex-basis`, qui s'appliquent aux flex-item.

```
.choucroute {
  flex: 1; /* je peux grandir et rétrécir */
}
.kuglopf {
  flex: 0; /* erf, pas moi :( */
}
```

Tableau 2–9 Les propriétés « flexibles » de Flexbox

Propriété	Effet sur l'élément
`flex-grow`	Capacité qu'a un élément de s'étirer dans l'espace restant.
`flex-shrink`	Habilité d'un élément à se contracter si nécessaire.
`flex-basis`	Taille initiale de l'élément avant que l'espace restant ne soit distribué.

Figure 2–30
Deux éléments dont flex vaut respectivement 1 et 0 (soit 1 1 auto et 0 1 auto)

La valeur par défaut de cette propriété est `flex: 0 1 auto` ; en clair, cela signifie que par défaut :

- un flex-item n'est pas autorisé à s'agrandir ;
- un flex-item peut se réduire ;
- la taille d'un flex-item est conditionnée par son contenu.

Les spécifications du W3C encouragent vivement les développeurs à opter pour cette propriété raccourcie plutôt que ses composantes individuelles, surtout si vous souhaitez tout simplement rendre un élément flexible sans trop vous poser de questions existentielles.

Et moi aussi d'ailleurs.

3

TP : réordonner des éléments

Où l'on commence à mettre les mains dans le cambouis et où l'on découvre qu'avoir un peu d'ordre pour ranger sa chambre est toujours une bonne chose.

Commençons par un exercice dont le but est simplement d'inverser l'affichage d'une image par rapport à son contenu textuel environnant : située en bas du texte sur grand écran, elle doit s'afficher en haut dès que la surface devient limitée.

Attention, interdiction de toucher à `position: absolute` ou à JavaScript, bien entendu !

Figure 3–1
Le résultat à obtenir

Le code HTML

La structure de notre cas d'étude est très sommaire car, dans un premier temps, j'ai préféré vous ménager un peu. Promis, les choses se compliqueront par la suite !

```
<section class="description">
  <p>Bonjour !</p>
  <p>Je m'appelle <strong>Raphaël Goetter</strong>.</p>
  <p>Je travaille dans le monde merveilleux du Web, fondateur d'Alsacréations et
auteur chez Eyrolles à mes heures perdues.</p>
  <p>bla bla bla&hellip;</p>
  <p class="biopic">
    <img src="me.jpg" alt="" width="" height="">
  </p>
</section>
```

Comme vous pouvez le constater, l'image (me représentant en pleine crise de mégalomanie) que je souhaite déplacer à l'affichage se trouve dans un paragraphe de classe `.biopic` et l'ensemble des paragraphes est lui-même contenu dans une section de classe `.description`.

Figure 3–2
L'affichage initial

REMARQUE Sections et titres

Les spécifications HTML 5 recommandent d'associer un niveau de titre à chaque élément de section (`<section>` et `<article>`) et le sacro-saint Validateur vous sanctionnera d'un avertissement pour le code que je viens de vous présenter. Rassurez-vous, il ne s'agit pas d'une erreur, mais simplement d'une bonne pratique en règle générale que je me suis permis de contourner dans le but de simplifier au maximum cet exercice

Une mission pour order !

Vous le savez déjà depuis au moins un chapitre, la propriété magique permettant de réordonner les éléments entre eux se nomme `order` :

```
.biopic {
  order: -1;
}
```

La propriété `order` a `0` pour valeur par défaut et fonctionne comme une pondération ou un `z-index` : plus les éléments ont une valeur importante, plus ils se situent à la fin de la pile.

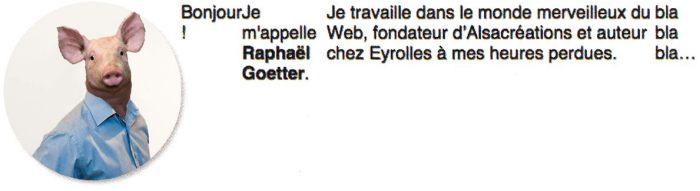
En donnant la valeur `-1` à `order`, on est assuré que le paragraphe et son image se positionnent avant tous les autres paragraphes dont la valeur est demeurée `0`.

Pour l'instant cependant, la magie n'opère pas encore : pour qu'`order` s'applique sur des éléments, il est nécessaire que ceux-ci se situent dans un contexte « flex ».

Le contexte d'affichage est appliqué sur la section `.description` à l'aide de `display: flex`. Cette section devient alors le flex-container et ses enfants directs, les paragraphes, deviennent des flex-item :

```
.description {
  display: flex;
}
.biopic {
  order: -1;
}
```

Figure 3–3
Résultat intermédiaire après application de display: flex



L'application de `display: flex` sur la section a des effets immédiats sur l'affichage : la propriété `order` fonctionne à merveille, mais tous les éléments s'affichent les uns à côté des autres tels des `inline-block` car la propriété `flex-direction` vaut initialement `row`. Il est donc nécessaire pour notre exercice de rétablir un affichage vertical en précisant que l'on souhaite une valeur de `column` :

```
.description {
  display: flex;
  flex-direction: column;
}
.biopic {
  order: -1;
}
```

Les Media Queries en action

Si vous avez déjà eu l'occasion de vous confronter au doux monde du Responsive Webdesign, la notion de Media Query (notée `@media (...)`) ne devrait pas avoir de secrets pour vous.

Dans le cas contraire, sachez qu'il s'agit d'une fonctionnalité CSS 3 reconnue par tous les navigateurs mobiles, tablettes et de bureau depuis Internet Explorer 9, capable de détecter certaines caractéristiques du support sur lequel le media s'affiche (notamment sa largeur, sa densité de pixels ou encore son orientation).

> Vous trouverez chez Alsacréations un article détaillé sur les Media Queries.
> ▸ http://kiwi.gg/mediaqueries

Grâce aux Media Queries, nous pouvons cibler les fenêtres de petite taille, par exemple inférieures ou égales à 480 pixels, et ne modifier l'affichage qu'au sein de cette condition pour obtenir le code CSS final de ces premiers travaux pratiques :

```
@media (max-width: 480px) {
  .description {
    display: flex;
    flex-direction: column;
  }
  .biopic {
    order: -1;
  }
}
```

Il n'y a rien de bien compliqué, n'est-ce pas ?

Limitations

La propriété `order` est un superbe jouet pour remodeler visuellement les éléments d'une page sans se soucier de l'ordre du DOM et sans nécessiter de `position: absolute` ou de JavaScript.

Il s'agit toutefois d'une propriété qui s'applique uniquement entre des frères flex-item. Vous ne pourrez pas réordonner un élément par rapport à l'un de ses cousins ou un oncle par exemple.

N'en abusez pas !

Au-delà de l'attrait incommensurable de cet exercice accompli, il me faut calmer d'emblée vos ardeurs en vous déconseillant de réaliser massivement ce genre de pirouettes en production.

Il s'avère que réordonner visuellement du contenu situé au-dessus de la « ligne de flottaison » (la partie de l'écran visible sans avoir à faire défiler) est une relative mauvaise idée en termes de performances d'affichage.

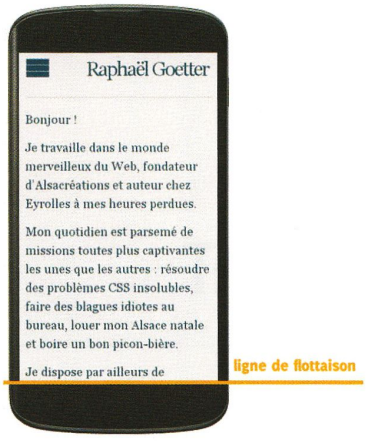


Figure 3–4
Illustration de la ligne (imaginaire) de flottaison

En effet, le navigateur peint la page et ses éléments une première fois dans l'ordre, puis va devoir repeindre une seconde fois certaines zones entières en raison de quelques éléments déplacés et qui poussent les contenus suivants. Les effets de bord visuels peuvent être assez perturbants si les contenus incriminés sont présents dès le début de la page.

La capture d'écran qui suit (générée avec le superbe outil – français – Dareboost) vous convaincra de ne pas abuser de ce genre de techniques.

▸ https://www.dareboost.com/

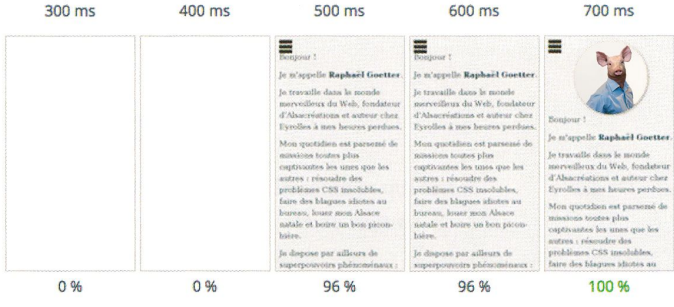


Figure 3–5
Le temps passé par le navigateur lorsqu'il réordonne des éléments

Ceci dit, comme je suis très joueur, j'ai conservé cette fonctionnalité sur mon site personnel, où vous pouvez l'observer dans son milieu naturel.

▸ http://www.goetter.fr/

4

Trois astuces utiles

Où – j'en suis persuadé – vous ne retiendrez au final que les deux mots « centrer » et « verticalement ».

Les présentations avec Flexbox et ses nombreuses propriétés (que j'espère vous connaissez déjà par cœur) étant faites, je vous invite sans tarder à découvrir trois astuces devenues totalement basiques grâce à ce modèle de positionnement et que vous allez adorer :

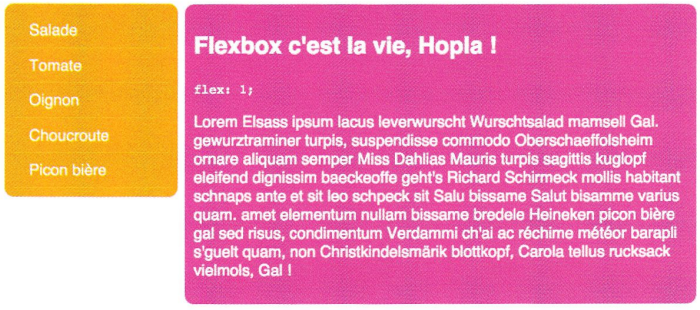
- occuper l'espace restant ;
- centrer verticalement ;
- pousser un élément ou un groupe d'éléments.

Occuper l'espace restant

Imaginez devoir réaliser le gabarit fluide le plus élémentaire qui soit :

- deux blocs (`<nav>` et `<main>`) s'installent l'un à côté de l'autre ;
- le premier bloc, `<nav>`, a une largeur définie à `10em` ;
- le second (`<main>`) doit occuper toute la largeur libre au sein de son parent, ici `<body>` ;
- un petit espace de 8 pixels doit séparer les deux blocs.

Figure 4–1
Le résultat final à obtenir



La partie HTML de notre gabarit est assez sommaire :

```
<body>
  <nav id="navigation">
    <a href="#">Salade</a>
    <a href="#">Tomate</a>
    <a href="#">Oignon</a>
    <a href="#">Choucroute</a>
    <a href="#">Picon bière</a>
  </nav>

  <main class="content">
    <p>Ici du contenu vraiment pertinent</p>
  </main>
</body>
```

Les différentes techniques historiquement envisageables pour réaliser notre mission sont nombreuses : `position: absolute`, `float`, `display: inline-block` et `display: table-cell`. Dans le détail, chacune aura des inconvénients : sortie du flux, superpositions, espaces indésirables à gérer, « clearfix », nécessité de bidouiller à l'aide de `calc()` ou `overflow: hidden`, etc.

Vous l'aviez déjà deviné : c'est Flexbox qui s'en sortira le mieux. Et de loin. La base tient en quelques lignes :

```
body {
  display: flex; /* crée un contexte flex pour ses enfants */
}
nav {
  width: 10em;
}
.content {
  flex: 1; /* occupe la largeur restante */
  margin-left: 8px;
}
```

C'est enfantin, non ?

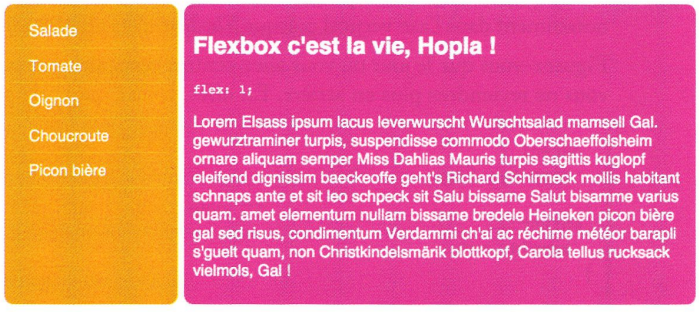


Figure 4–2
Affichage intermédiaire après application d'un contexte Flexbox

Un petit détail ne vous a sans doute pas échappé : les deux flex-item, à savoir la navigation et le bloc de contenu, ont tous deux par défaut la même hauteur. C'est plutôt agréable à l'œil, mais ce n'est pas le résultat que nous souhaitions obtenir.

L'explication tient au fait que la propriété `align-items` qui régit l'alignement dans l'axe secondaire (ici vertical) a une valeur par défaut de `stretch`. Les éléments sont donc étirés.

L'ajustement tient en une ligne :

```
body {
  …
  align-items: flex-start; /* alignement vertical */
}
```

Centrer verticalement

S'il y a bien une fonctionnalité que l'on attend depuis une bonne vingtaine d'années en CSS, c'est bien de pouvoir centrer verticalement n'importe quel élément au sein de son parent, sans s'arracher les cheveux.

À une époque pas si lointaine, nous mettions en œuvre de jolis tableaux HTML ornés d'un simple `vertical-align: middle` et c'était réglé, mais on nous a vite tapé sur les doigts, alors… eh bien on bidouille !

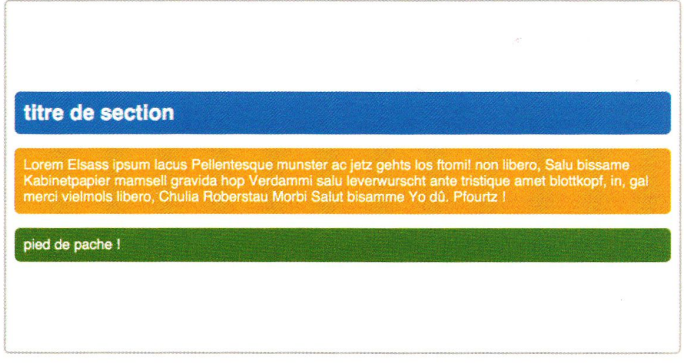


Figure 4–3
Le résultat final à obtenir

La propriété `margin`, lorsqu'elle est affectée à un flex-item, ouvre de nouvelles perspectives, notamment dans l'axe vertical puisque Flexbox n'est plus lié à un sens de lecture en particulier.

Figurez-vous que le jour où vous allez centrer verticalement pour la première fois via Flexbox, vous ne reviendrez plus en arrière. En effet, voilà à quoi cela peut ressembler :

```
.parent {
  display: flex;
}
.enfant {
  margin: auto;
}
```

L'explication de ce petit prodige est simple : Flexbox étant multidirectionnel, `margin: auto` se doit d'opérer aussi bien horizontalement que verticalement dans un flex-container.

Une autre solution eût été de signifier la propriété de centrage vertical directement sur le conteneur :

```
.parent {
  display: flex;
  flex-direction: column;
  justify-content: center;
}
```

Votre choix entre ces deux méthodes dépendra essentiellement du nombre d'enfants à centrer verticalement : s'ils sont plusieurs, la seconde technique fonctionnera parfaitement, tandis que s'il n'y a qu'un élément à center, `margin: auto` est suffisant.

Pousser un élément

« Les marges automatiques, ce n'est pas automatique… mais c'est rudement pratique ! » Voilà une maxime qui ne veut pas dire grand-chose a priori mais qui prendra beaucoup de sens au sein de vos contextes Flexbox.

En dehors du célébrissime `margin: auto` pour centrer horizontalement (et verticalement dans Flexbox comme nous venons de le voir), connaissiez-vous les bienfaits de `margin-left: auto` et `margin-top: auto` appliquées sur un flex-item ?

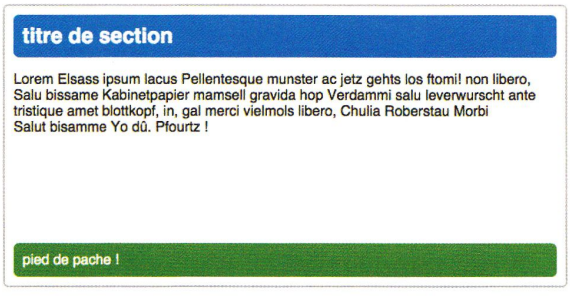
- `margin-left: auto` pousse l'élément vers la droite.
- `margin-top: auto` le pousse vers le bas.
- `margin-bottom: auto` le pousse vers le haut.

Connaître ces astuces vous aidera à réaliser très facilement quelques prouesses de positionnements en vous passant de flottants ou de positions absolues.

```
<section>
  <h1>titre de section</h1>
  <p>Lorem Elsass ipsum bla bla</p>
  <footer>pied de pache</footer>
</section>
/* contexte général flex sur le conteneur */
section {
  display: flex;
  flex-direction: column;
  min-height: 20em;
}

/* hop, on pousse footer en bas */
footer {
  margin-top: auto;
}
```

Figure 4–4
Le résultat final à obtenir



Une seule instruction pour pousser un pied de page en bas de son conteneur sans sortir du flux, c'est plutôt excitant, n'est-ce pas ? Et vous allez voir que ce n'est qu'un début !

5

TP : une navigation Responsive

Où l'on affronte enfin avec hardiesse le Boss du premier niveau à grands coups de Media Queries level 42.

Entamons à présent un exercice pour le moins délicat et primordial pour un site web : celui de concevoir un menu de navigation agréable et Responsive.

Le module de positionnement Flexbox nous sera d'une aide précieuse dans cette quête, notamment pour sa gestion des alignements verticaux et horizontaux.

Pour information, la fonte utilisée pour ce travail pratique est Calendas Plus, libre de droits et offrant moult jolies ligatures de glyphes.

▸ http://calendasplus.com/

À quelques détails près, il s'agit de la navigation actuellement employée sur mon site personnel ; vous pouvez donc vous y référer et en décortiquer le code source.

Figure 5–1
Le résultat final sur grand écran

Le code HTML

Sans plus attendre, je vous présente la structure qui servira de base à nos travaux pratiques :

```
<nav id="navigation">
  <h1>
    <a href="" data-abbr="RG">
      <span>Raphaël Goetter</span>
    </a>
  </h1>
  <a href="">à propos</a>
  <a href="">blog</a>
  <a href="">labo</a>
  <a href="">contact</a>
</nav>
```

Voici quelques précisions quant à ce bout de code :

- vous n'y trouverez pas de liste `<ul>`, `<li>` : je ne la considère ni nécessaire, ni systématique, la « sémantique » étant apportée par l'élément `<nav>` ;
- il n'y a pas d'attribut d'accessibilité ARIA `role=navigation` non plus. Étant intrinsèque à l'élément `<nav>`, il serait ici redondant et déconseillé par les spécifications WAI-ARIA ;
- l'`id` du parent a un double emploi : celui d'ancre pour assistances techniques et celui d'identifiant pour un script JavaScript, mais il ne servira pas de sélecteur CSS ;
- le premier élément de navigation, menant vers la page d'accueil, a également le rôle de titre. Il est donc représenté par un `<h1>`.

Titre complet ou initiales ?

Selon la surface d'affichage, j'ai souhaité adopter deux comportements différents :

- sur petit écran, par exemple celui d'un smartphone, l'intitulé entier « Raphaël Goetter » apparaît et sert de lien vers la page d'accueil ;
- sur grand écran, ce contenu est remplacé par les initiales « RG » décorées grâce aux ligatures contenues dans la fonte Calendas.

Voici comment j'ai procédé en trois temps :

1. par défaut, le titre est affiché. Tout commence bien, pas besoin de Media Queries pour cela ;
2. sur grands écrans, on masque le `<span>` tout en conservant l'information pour les assistances techniques et lecteurs vocaux :

```
@media (min-width: 481px) {
  nav h1 span {
    position: absolute;
    clip: rect(0,0,0,0);
  }
}
```

3 toujours sur grands écrans, on affiche les initiales « RG » à l'aide d'un pseudo-élément :

```
@media (min-width: 481px) {
  nav h1 a::before {
    content: attr(data-abbr);
    display: inline-block;
    width: 17rem;
    height: 17rem;
    border-radius: 50%;
    background-color: #335271;
    ...
  }
}
```

Écrans de smartphones

Figure 5–2
Le résultat final sur écrans de smartphones

En observant le visuel à obtenir sur smartphones, une évidence nous saute aux yeux : à l'exception de quelques décorations purement esthétiques, nos liens de navigation sont disposés tels de simples blocs les uns sur les autres et occupant toute la largeur.

Plutôt que de débuter, comme habituellement, par des styles pour grands écrans et de les écraser par la suite pour les écrans réduits via Media Queries, le plus profitable est ici de signifier aux liens qu'ils se comportent par défaut en `display: block` et de ne modifier cet état que lorsque les conditions de largeur changent.

Bref, commençons notre réflexion à partir de la taille d'écran minimale et pensons *mobile first* ; nous y économisons à la fois en Media Queries et en règles CSS surchargées :

```
nav a {
  display: block;
  padding: .4em 1em;
  text-decoration: none;
  color: #fff;
}
```

Cela ne devrait pas vous surprendre : sur un écran tactile, l'état naturel de survol n'existe pas vraiment. Pour simuler cet état lors du « tap », n'hésitez pas à cumuler les événements CSS pour être sûr que l'un d'entre eux fonctionne :

```
nav a:hover,
nav a:focus,
nav a:active {
  background: #49535d;
}
```

Grands écrans

Figure 5–3
Le résultat final sur grands écrans

Grosso modo, la mission à accomplir sur les écrans larges est de disposer les liens les uns à côté des autres, au centre de l'écran et de les répartir de part et d'autre du titre `<h1>`.

Une Media Query telle que `@media (min-width: 1025px) {...}` ne nous sera pas de trop pour entamer notre tâche. Au sein de cette Media Query, indiquons les quelques instructions suivantes :

```
@media (min-width: 1025px) {
  nav {
    display: flex;
    justify-content: center;
    align-items: center;
  }
}
```

`<nav>` est notre flex-container et ses enfants s'affichent horizontalement (`flex-direction: row;` par défaut).

Les alignements horizontaux et verticaux sont tous deux définis au centre, via les propriétés `justify-content` et `align-items`. C'est tout, le reste n'est que de la pure décoration.

Avouez que c'est un vrai jeu d'enfant !

Figure 5–4
L'affichage intermédiaire sur grands écrans après application d'un contexte Flexbox

Astuce : réordonner les éléments

Pour parfaire l'affichage, il ne reste plus qu'à positionner l'élément de titre `<h1>` au centre de ses frères, sans modifier le code HTML bien entendu.

Il suffit pour cela de tirer profit de la propriété Flexbox `order`, une véritable bénédiction pour ce genre de mission ordinairement périlleuse :

```
nav a:nth-of-type(-n+2) {
  order: -1;
}
```

C'est à la fois rapide et suffisant pour cibler les deux premiers liens enfants de `<nav>` et les afficher avant l'élément de titre `<h1>` ainsi que les autres liens dont la valeur de `order` demeure `0`.

Figure 5–5
L'affichage sur grands écrans après avoir réordonné les blocs

Pour en savoir plus sur les sélecteurs de type `:nth-child()` ou `:nth-of-type()`, compatibles depuis Internet Explorer 9, je vous invite à les tester sur le bac à sable en ligne :

- http://nth-test.com/

Écrans de taille moyenne (tablettes)

Figure 5–6
Le résultat final sur tablettes

Pour s'adapter à un écran de taille plus réduite, celui d'une tablette, il est tout à fait envisageable de modifier l'affichage et de passer à une distribution verticale en disposant les liens deux par deux, par exemple :

```
@media (min-width: 481px) and (max-width: 1024px) {
  nav {
    flex-wrap: wrap;
  }
  nav h1 {
    width: 100%;
    text-align: center;
  }
  nav > a {
    width: 50%;
    text-align: center;
  }
}
```

Le titre occupe à présent toute la largeur de la navigation tandis que les autres liens se répartissent deux par deux sur leur rangée. Le passage à la ligne est rendu possible grâce à la déclaration `flex-wrap: wrap`.

Le truc en plus : la « variable » currentColor

Pour finir en beauté, ajoutons une touche d'interactivité au survol et au focus (navigation avec les touches de clavier) à ce magnifique menu de navigation.

Le but du jeu est de souligner par défaut les liens d'un trait de 1 pixel, puis d'élargir le trait à 5 pixels lors du survol. Ce trait doit automatiquement adopter la couleur du texte du lien.

Afin de bénéficier d'un soulignement aussi malléable et stylable que possible, nous n'allons pas utiliser `text-decoration` ni `border`, mais nous orienter vers les dégradés linéaires CSS 3. Nous allons générer un dégradé d'une couleur identique du début à la fin de son parcours. Le gros avantage de cette technique est que le soulignement peut être positionné au pixel près (avec `background-position`), dimensionné (avec `background-size`) et même subir une transition de couleur, de position ou de taille ; tout cela sans modifier la taille de la boîte de l'élément.

Dans l'exemple de code qui suit, `background-size` définit la hauteur de notre ligne : elle est de `1px` par défaut et passe à `5px` au survol ou au focus, le tout en transition de 0,25 secondes.

Le truc en plus est l'emploi du mot-clé `currentColor` (reconnu depuis IE9 et les autres navigateurs modernes) qui va nous servir de variable parfaite : **la couleur du soulignement sera toujours et dynamiquement identique à celle du texte du lien.**

```
nav > a {
  background-image: linear-gradient(to right, currentColor, currentColor);
  background-repeat: no-repeat;
  background-size: 100% 1px;
  background-position: center bottom;
  transition: background-size .25s;
}
```

```
nav > a:hover,
nav > a:focus {
  background-size: 100% 5px;
  outline: 0;
  color: #ba3176;
}
```

Grâce à l'astuce de `currentColor`, combinée au choix d'un dégradé linéaire et saupoudrée d'une transition CSS, nous obtenons un effet de survol des liens pour le moins réussi !

6

TP : un gabarit simple

Où l'on récupère le fameux Saint-Graal perdu depuis 20 ans sous un paquet IP, tout en profitant de l'occasion pour sauver la princesse qui traînait aussi dans le coin.

Dans les milieux autorisés, le *Holy Grail* (le Saint-Graal) désigne un gabarit d'affichage extrêmement fluide, introduit en 2006 par un article de référence de *A List Apart* (http://alistapart.com/article/holygrail) et dont voici les caractéristiques fondamentales :

- la partie centrale doit être fluide et les parties latérales de largeur fixe ;
- la colonne centrale doit apparaître en premier dans le code HTML ;
- toutes les colonnes doivent avoir la même hauteur, quel que soit leur contenu ;
- le pied doit être collé au bas de page même s'il y a peu de contenu dans la page ;
- les codes HTML et CSS doivent être minimaux.

Beaucoup de webdesigners s'y sont cassé les dents par le passé et aucun jusqu'alors ne l'avait réussi sans moult hacks et bidouilles.

En entrant dans l'ère Flexbox, ce genre de gabarit est passé directement du stade « Holy Grail Layout » à… « Gabarit Simple », tellement sa réalisation est rapide et simple. En effet, seules quelques propriétés basiques de Flexbox suffisent à le concevoir.

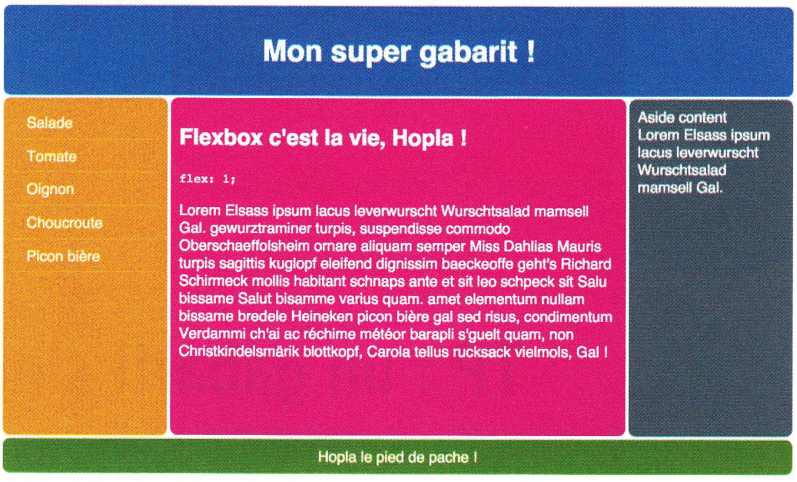


Figure 6–1 Le résultat final à obtenir

La structure HTML

Voici à quoi s'apparente le code HTML prévu pour cet exercice pratique (notez que j'ai volontairement omis de préciser les éléments `<doctype>` et `<head>` pour plus de lisibilité) :

```
<body>
  <header>
    <h1>Mon super gabarit !</h1>
  </header>

  <main class="wrapper">

   <section class="content">
      <h2>Flexbox c'est la vie, Hopla !</h2>
      <p>Lorem Elsass ipsum lacus leverwurscht Wurschtsalad mamsell Gal.
gewurztraminer Carola tellus rucksack vielmols, Gal !</p>
    </section>

    <nav id="navigation">
      <a href="#">Salade</a>
      <a href="#">Tomate</a>
      <a href="#">Oignon</a>
      <a href="#">Choucroute</a>
      <a href="#">Picon bière</a>
    </nav>
```

```
    <aside>
      <h2>Aside content</h2>
      <p>Lorem Elsass ipsum lacus leverwurscht Wurschtsalad mamsell Gal.</p>
    </aside>

  </main>

  <footer>Hopla le pied de pache !</footer>
</body>
```

Donnons quelques explications sommaires :

- le corps de mon gabarit est `<body>`, il est parent des trois frères `<header>`, `<main>` et `<footer>` ;
- `<main>` est lui-même parent de la section `.content`, de `<nav>` et de `<aside>`.

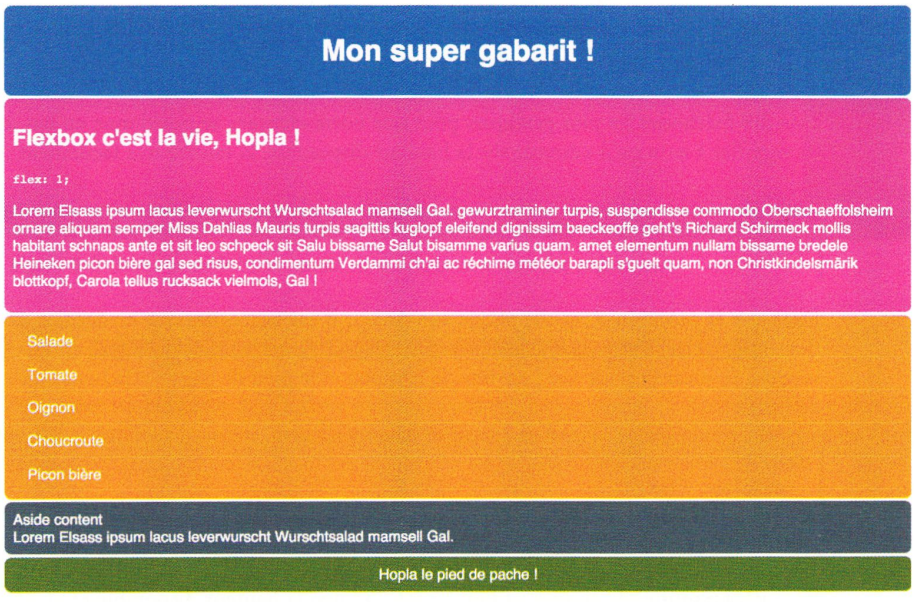


Figure 6–2 L'affichage initial avec la disposition naturelle

Étape 1 : occuper toute la hauteur

Première mission : nous aimerions que notre gabarit occupe toujours au moins toute la hauteur de la page, même lorsqu'il y a peu de contenu.

Grâce aux unités de *viewport*, compatibles depuis Internet Explorer 9 et les autres, cette étape est réalisable en une seule instruction :

```
body {
  min-height: 100vh;
}
```

L'unité `vh` correspond à la hauteur de la fenêtre. Elle signifie *viewport height* et `100vh` désigne 100 %, soit l'intégralité, de cette hauteur.

REMARQUE height c'est le mal !

Optez pour `min-height` plutôt que `height` car cette dernière propriété est figée une fois pour toutes : un vaste contenu, plutôt que de l'allonger, va alors déborder de son conteneur.

Étape 2 : pied en bas de page

Une des méthodes les plus pratiques pour que notre `<footer>` se positionne en bas de page est de signifier à `main` qu'il s'étende sur la hauteur disponible, là encore très facilement :

```
main {
  flex: 1 1 auto; /* flex-grow devient 1 */
}
```

Pour que la propriété `flex` s'applique sur `<main>`, il est nécessaire que ce dernier devienne un flex-item, ce qui implique que son parent, en l'occurrence `<body>`, devienne un flex-container, vertical de surcroît pour que l'en-tête, le contenu et le pied de page s'affichent l'un en dessous de l'autre :

```
body {
  display: flex;
  flex-direction: column;
}
```

Et voilà, l'essentiel du travail est fait : `<main>` occupe la hauteur disponible et `<footer>` est par conséquent toujours collé en bas de page même s'il y a peu de contenu sur la page.

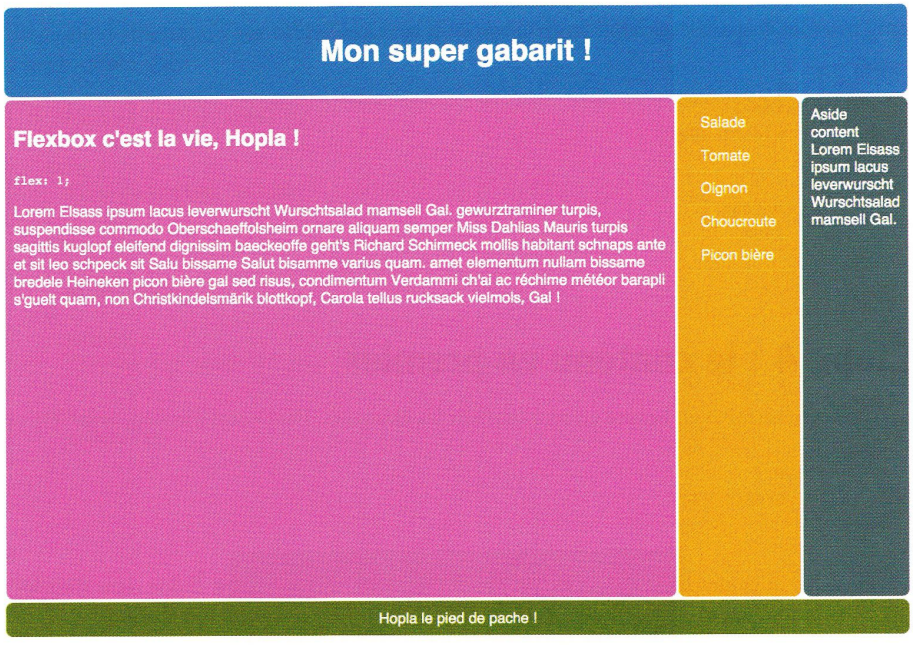


Figure 6–3 Étape intermédiaire, occupation de toute la hauteur et pied de page en bas

Étape 3 : navigation, aside et contenu principal

Pour créer un contexte de Flexbox pour ses enfants, `<main>` doit à présent lui-même devenir un flex-container (oui oui, tout en étant flex-item à la fois, la schizophrénie est tolérée chez Flexbox).

```
main {
  display: flex;
}
```

Il se trouve que, pour notre plus grand bonheur, les propriétés de direction et d'alignement vertical sont définies par défaut avec les valeurs suivantes : `flex-direction: row` , `align-items: stretch`.

C'est parfait pour nous, car cela permet à `<section class="content">`, `<nav>` et `<aside>` de se disposer les uns à côté des autres et de s'étirer automatiquement sur toute la hauteur de leur parent. La vie est quand même bien faite, non ?

Il suffit à présent d'indiquer une largeur à `<nav>` et `<aside>` puis d'autoriser `.content` à occuper tout l'espace disponible horizontal, de la manière la plus simple qui soit :

```
nav,
aside {
  width: 10em;
}

.content {
  flex: 1;
}
```

Étape 4 : le contenu en premier

On termine en beauté en réordonnant les éléments. Pour cette mission, `order` est notre ami :

```
nav {
  order: -1;
}
```

En quelques lignes de CSS, on obtient aujourd'hui un gabarit fluide dans tous les sens, sans bidouille des Temps Anciens et en 13 minutes et 37 secondes top chrono.

C'est Responsive, ma bonne dame !

Adapter ce gabarit à différentes tailles d'écrans se fait sans heurts et en quelques lignes également.

La première solution consiste tout simplement à initialiser les valeurs de `display` et de `width` afin de sortir du modèle Flexbox et de se retrouver dans un flux de blocs les uns sur les autres.

```
/* Responsive */
@media (max-width: 640px) {
  main,
  nav,
  aside,
  .content {
    display: block;
    width: auto;
  }
}
```

Une solution plus évoluée continue à profiter du modèle Flexbox en modifiant simplement la direction d'affichage. Cela offre l'avantage de pouvoir réordonner les éléments à l'écran grâce à la propriété `order`.

```
/* Responsive (variante) */
@media (max-width: 640px) {
  main {
    flex-direction: column;
  }
  nav,
  aside {
    width: auto;
  }
  .content {
    flex-basis: auto; /* pour écraser la valeur 0 */
  }
  nav {
    order: 1;
  }
}
```

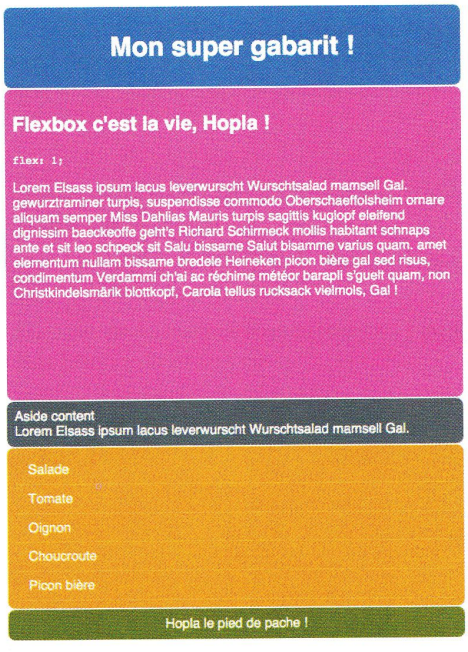


Figure 6–4
La version Responsive « petits écrans »

7

TP : une galerie d'images

Où l'on s'affaire à décorer les tapisseries du donjon avec les albums photo de tous les selfies *de la princesse. Et il y a du travail !*

L'objectif de ce nouvel exercice pratique est de tirer bénéfice de Flexbox pour l'affichage d'un lot de photographies au sein d'une page web.

Pour ce faire, nous allons éprouver les deux types de distributions possibles (horizontale et verticale), puis en tirer les conclusions en production.

Notre lot est composé d'images de formats différents (carrées, paysage, portrait) pour corser la difficulté, parce qu'il faut bien s'amuser un peu.

Vous l'avez compris, le défi est d'obtenir une composition harmonieuse de tous ces éléments aux proportions hétéroclites, en les répartissant au sein de l'espace disponible tout en conservant les ratios d'affichage intrinsèques.

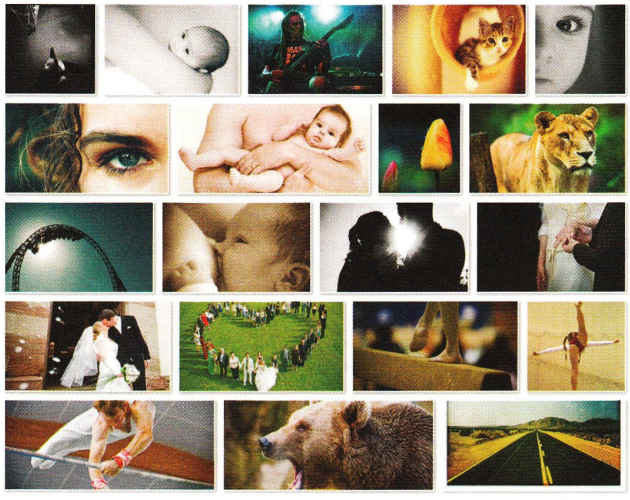


Figure 7–1
L'une des versions d'affichage souhaitées

Le code HTML

La partie HTML est des plus spartiates :

```
<div class="gallery">
  <img src="" alt="" height="" width="">
  <img src="" alt="" height="" width="">
  <img src="" alt="" height="" width="">
  ...
</div>
```

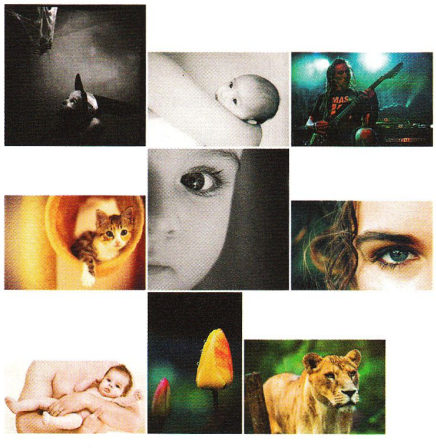


Figure 7–2
Affichage initial, sans aucun style

Répartition horizontale

Commençons nos expériences en disposant les images horizontalement à l'aide du code CSS suivant :

```
.gallery {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: space-between;
  align-items: flex-start;
}

.gallery   img {
  flex: 0 1 auto;
  width: 24%; /* ou bien flex: 0 1 24% */
  height: auto;
}
```

Figure 7–3
Affichage horizontal, première tentative

Explications

La répartition horizontale est ici définie via `flex-direction: row;` (valeur par défaut). Chaque image occupe une largeur de `24%` ; il demeure donc 4 % d'espace restant. Celui-ci n'est pas comblé puisque `flex-grow` vaut `0` ; au contraire, il est réparti entre les images grâce à `justify-content: space-between;`.

Enfin, la déclaration `align-items: flex-start;` empêche les images d'être étirées (contre toute attente, la valeur par défaut de cette propriété est `stretch`).

Voici un petit conseil en passant : préférez encore quelque temps `width: 24%;` à `flex: 0 1 24%`. En effet, sur certains navigateurs, l'irremplaçable déclaration `box-sizing: border-box` qui

s'applique parfaitement à `width` n'a pas d'effet sur `flex-basis`, ce qui aura pour conséquence que vos éléments pourraient avoir une taille dépendant des valeurs de `padding` et `border` qui leur auraient été associées.

Limitations

Cette première approche de distribution en ligne laisse un goût d'inachevé. En effet, si le ratio des images est conservé (grâce à `height: auto;`) et si les éléments sont harmonieusement répartis horizontalement (grâce à `justify-content`), l'inconvénient majeur est que le rythme vertical laisse franchement à désirer : de nombreux espaces blancs sont à déplorer et le résultat pourrait être plus joli.

À partir de ce brouillon, plusieurs expériences complémentaires peuvent être menées :

1. autoriser `flex-grow`. Nous perdons alors l'espace interélément offert par `space-between` mais nous pouvons le récupérer artificiellement à l'aide de marges externes aux images. Peu d'intérêt au final ;
2. ne pas indiquer de largeur ou préciser une largeur fixe (par ex. `width: 200px`) ; nous obtenons alors le même type de résultat qu'avec une combinaison de `display: inline-block` et de `text-align: justify`. Là encore, rien de transcendant ;
3. imposer une hauteur (par ex. `height: 100px`) et une largeur automatique (`width: auto`) ; le rythme vertical est grandement amélioré, mais nous perdons la régularité de l'espacement entre les images. Cette solution est cependant bien plus agréable à l'œil que les précédentes.

Figure 7–4
Affichage horizontal avec hauteur imposée

En bref, quelle que soit la variante choisie, l'affichage obtenu n'est pas digne des promesses d'un positionnement tel que Flexbox.

Pas de panique, voyons si l'on s'en sort mieux en tentant une répartition verticale.

Répartition verticale

Deuxième chance, organisons cette galerie via le code suivant :

```
.gallery {
  display: flex;
  flex-direction: column;
  flex-wrap: wrap;
  align-content: space-between;
  height: 100vh;
}

.gallery > img {
  width: 24%;
  height: auto;
}
```

Figure 7–5
Affichage vertical, première tentative

Explications

Cette fois, `flex-direction` indique une orientation de type vertical (`column`). Les images s'empilent donc les unes sous les autres en respectant un rythme vertical parfait. La répartition horizontale entre les rangées est assurée par la déclaration `align-content: space-between`.

Seule légère ombre à ce tableau, il est absolument nécessaire de définir une hauteur au conteneur (`height` ou `max-height`) afin que la propriété `flex-wrap` fasse effet. Et là, c'est moins drôle !

Limitations

Imposer une hauteur fixe au conteneur des photos est un réel frein à l'adoption de cette technique verticale.

Pour qu'elle fonctionne correctement, il est nécessaire de connaître le nombre d'images à afficher, mais aussi de tester différentes tailles d'écrans (et des `media queries`) et de corriger au cas par cas les débordements et défauts d'affichage.

Ici, j'ai choisi de définir une hauteur qui n'est autre que celle de la fenêtre, via `height: 100vh;`, mais cela ne suffit pas à éviter les débordements et effets de bords.

Il existe une astuce pour réduire un bon nombre de désagréments. Elle consiste à indiquer les valeurs de largeurs minimale et maximale, par exemple ainsi :

```
.gallery > img {
  width: 24%;
  max-width: 200px;
  min-width: 130px;
  height: auto;
}
```

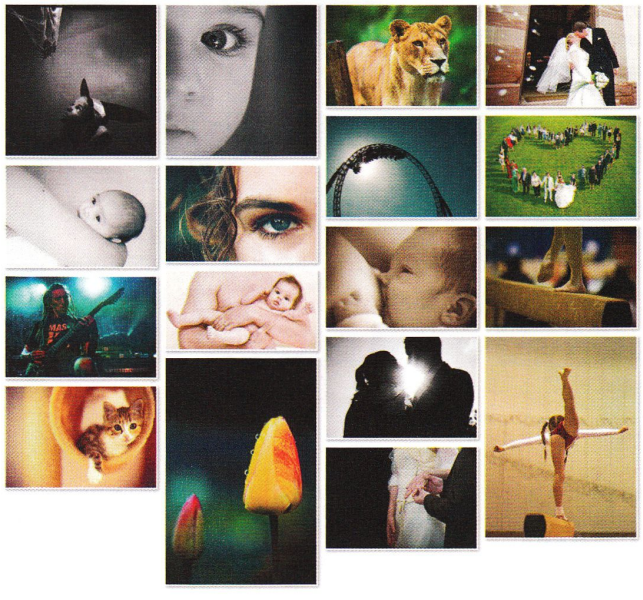


Figure 7–6
Affichage vertical avec hauteur et largeur encadrées

Même à l'aide de cette astuce, le résultat n'est pas parfait ni dénué de mauvaises surprises selon l'espace disponible.

Bref, quel que soit le cas de figure et le type de répartition, il semble impossible de répartir des images en comblant tout l'espace libre et en conservant leurs ratios d'affichage respectifs.

Avant de se résigner et de retailler toutes nos photos une à une dans un même format d'affichage, envisageons une dernière solution apportée par la récente et peu connue propriété `object-fit`.

object-fit à la rescousse

La propriété `object-fit` restreint la taille d'un contenu à celle de son parent tout en conservant son ratio, un peu comme le fait déjà la propriété `background-size` appliquée à des images d'arrière-plan avec ses valeurs `cover` et `contain` (`object-fit` reconnaît d'ailleurs les mêmes valeurs que `background-size`).

La voici appliquée à nos tentatives d'affichage horizontal :

```
.gallery {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: flex-start;
}
.gallery > img {
  flex: 1 1 auto;
  width: auto;
  height: auto;
  max-width: 300px;
  max-height: 130px;
  margin: 5px;
  object-fit: cover;
}
```

Figure 7–7
Affichage horizontal avant application de object-fit

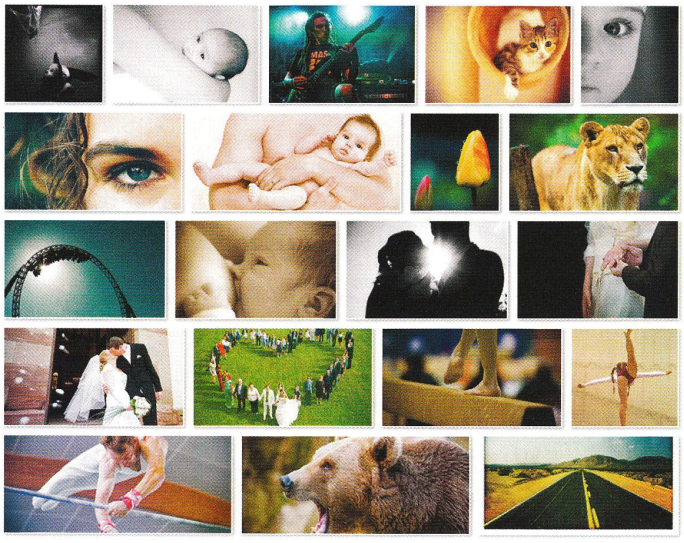


Figure 7–8
Affichage horizontal après application de object-fit

Explications

À présent, nos images disposent de trois caractéristiques qu'elles ne pouvaient pas cumuler dans nos essais précédents :

- une largeur maximale définie ;
- une hauteur maximale définie ;
- un ratio conservé via `object-fit: cover`.

Pour que `object-fit` s'applique, il est nécessaire d'indiquer une largeur et une hauteur à l'élément, ce qui ne nous frustre pas outre mesure à condition d'accepter que certaines photos soient rognées pour « entrer dans la boîte ».

Le truc en plus : object-position

La propriété `object-position`, dont la valeur par défaut est `50% 50%`, est la parfaite complémentaire de `object-fit`, dans la mesure où elle va permettre de positionner l'élément tel une image de fond.

Vous pouvez donc, au cas par cas selon l'image, l'affubler d'une classe qui va la replacer comme vous le souhaitez plutôt à gauche, ou à droite ou en bas, etc.

```
.to-top {
  object-position: top;
}
```

```
.to-bottom {
  object-position: bottom;
}
.to-center {
  object-position: center;
}
.to-left {
  object-position: left;
}
.to-right {
  object-position: right;
}
```

Limitations

La compatibilité de `object-fit` et surtout de `object-position` n'est pas vraiment exceptionnelle, comme le montre notre fidèle Caniuse.com : pas d'`object-fit` pour Internet Explorer et pas d'`object-position` du côté de Safari à la fin 2015.

Voyons la vie du bon côté et disons-nous qu'en attendant une meilleure prise en charge, ce n'est pas si dramatique… à condition d'accepter que nos images soient légèrement déformées sur IE et systématiquement centrées sur Safari.

Effets au survol et focus

Juste pour le plaisir, nous pouvons pousser l'expérience utilisateur encore plus loin en lui proposant de ne garder bien présente que l'image survolée en floutant, grisant et « dézoomant » les autres :

```
.gallery img {
  height: 63px;
  object-fit: cover;
  transition: transform .2s;
}
.gallery:hover img {
  filter: blur(1px) brightness(.8) grayscale(.7);
  transform: scale(.8);
}
.gallery img:hover {
  filter: blur(0);
  transform: scale(1);
}
```

Figure 7–9
Affichage des éléments avant et après leur survol

8

Les principes fondamentaux

Où l'on découvre un à un tous les cheat mode *piochés sur Google pour anéantir le vilain dragon Boss final du jeu.*

À présent que notre aventure dans le monde de Flexbox est bien entamée et que nous avons pu mettre les mains dans le cambouis de certains cas concrets, je pense qu'il est grand temps de vous enseigner certaines règles fondamentales de ce module somme toute très particulier.

Ce chapitre et ses conseils sont à conserver au chaud dans un coin de votre ordinateur cervical pour ne pas avoir de surprise dans vos intégrations.

Flexbox est une spécification qui se veut simple, puissante et très stable… en tout cas sur le papier ! Dans la pratique, les navigateurs ne sont pas toujours de cet avis. Voilà pourquoi je vous propose ici un condensé de la façon d'appréhender Flexbox en tant que webdesigner ou intégrateur, sous forme de :

- trois principes fondamentaux généraux ;
- cinq règles techniques à ne jamais oublier pour éviter les mauvaises surprises.

Les trois principes fondamentaux de Flexbox

1 - De l'apparente simplicité de Flexbox

Flexbox est un module très simple.

… Une fois que l'on en retient toutes les propriétés et valeurs non intuitives.

… Et tant que l'on n'essaye pas de comprendre en détail les propriétés `flex-grow`, `flex-shrink` et `flex-basis`.

En effet, ces dernières – bien qu'intuitives – sont fondées sur des spécifications et des algorithmes assez complexes.

2 - De l'apparente omnipotence de Flexbox

Flexbox est potentiellement capable de remplacer tout autre module de positionnement.

… Mais n'essayez pas de tout faire avec Flexbox : chaque schéma de positionnement a ses avantages et inconvénients et ne doit pas être utilisé n'importe comment.

Cela signifie que `float`, `inline-block` et `position: absolute` ont toujours leur mot à dire ; simplement, la distinction entre les rôles de chaque positionnement devient plus tranchée.

3 - De l'apparente stabilité de Flexbox

Comme tous les autres positionnements, Flexbox n'est pas exempt de bogues de navigateurs. Avant de crier victoire et de s'extasier devant la magie opérée, prenez le temps de tester sur Internet Explorer (10 et 11 notamment), ou sur d'anciennes versions d'Android (2 à 4 par exemple) ; vous risquerez de vite déchanter.

Les cinq pense-bêtes de l'intégrateur

Voici à présent quelques règles plus techniques destinées à vous faire économiser quelques heures de frustration et quelques séances de psychanalyse.

Notez bien que toutes les règles énoncées ci-après sont valables dans les deux directions d'affichage (ligne ou colonne), c'est-à-dire que tout ce qui vaut pour `width`, `max-width` et `min-width` vaut pour `height`, `max-height` et `min-height`.

1 - min-width et max-width sont prioritaires sur tout

Première loi de Flexbox Layout : `min-width` et `max-width` sont toujours prioritaires au final sur les calculs de dimensions obtenus via les propriétés `width`, `flex-grow`, `flex-shrink` et `flex-basis`, de même que pour l'application de `flex-wrap`.

En outre, s'il y a bataille entre `min-width` et `max-width`, c'est toujours le premier qui l'emporte.

Concrètement : un élément disposant d'un `flex-basis: 50%` et d'un `min-width: 80%` appliquera la valeur de `min-width`.

2 - flex-wrap est prioritaire sur flex-shrink

La propriété `flex-wrap` sur un flex-container est prioritaire par défaut sur le `flex-shrink` de ses flex-item.

Concrètement : même s'il dispose d'un `flex-shrink` positif, un élément dimensionné ne se réduit pas et passe à la ligne si le `flex-wrap` de son parent vaut `wrap` ou `wrap-reverse` ; *ensuite seulement* l'élément se réduit s'il bénéficie d'un `flex-shrink` positif.

3 - flex-shrink et flex-grow sont prioritaires sur flex-basis

Un élément, même dimensionné avec `flex-basis`, demeure contractile s'il dispose d'un `flex-shrink` positif et extensible s'il dispose d'un `flex-grow` positif.

Concrètement : un flex-item muni d'un `flex-shrink` ou d'un `flex-grow` non nul se débrouille toujours pour s'agrandir ou se réduire dans l'espace restant, même s'il est dimensionné au départ.

4 - flex-basis est prioritaire sur width

Si un flex-item dispose d'un `flex-basis` de valeur autre que `auto` (qui est la valeur par défaut), alors `flex-basis` écrase systématiquement la propriété `width` de ce flex-item.

Concrètement : en indiquant un `flex: 1` sur un élément, la propriété `width` n'a plus d'effet sur lui car `flex: 1` confère la valeur de `0%` à `flex-basis` qui devient alors prioritaire sur `width`.

5 - Dimensions minimales intrinsèques

Par défaut, un flex-item ne peut pas se rétrécir (`flex-shrink`) en deçà de sa largeur de contenu qui est représentée par la valeur `min-content` (grossièrement, son mot le plus long ou autre contenu insécable ou dimensionné) ; ceci hors contre-indication de `width`, `min-width` ou `max-width`.

Concrètement : des éléments enfants d'un flex-container et contenant eux-mêmes des éléments de taille fixe, des images par exemple, ne pourront par défaut pas se réduire en deçà de cette largeur.

Récapitulatif des priorités

Pour reprendre synthétiquement l'ensemble des règles sus-décrites, voici une liste des propriétés classées dans leur ordre de priorité d'application. Apprenez-la par cœur pour éviter toute surprise !

1. `min-width/min-height`
2. `max-width/max-height`
3. `flex-wrap`
4. `flex-shrink/flex-grow`
5. `flex-basis`
6. `width/height`

9

TP : un formulaire fluide

Où l'on découvre avec effroi que le contrat nuptial entre la princesse et le dragon contient des centaines de champs de formulaires mal dimensionnés.

Ce n'est certainement pas à vous que je vais apprendre que les éléments de formulaires comptent parmi les plus alambiqués à styler correctement, au point qu'une ancienne légende tchétchène les comptabilisait parmi les douze travaux d'Hercule, à mi-chemin entre « la newsletter au pixel près » et la « parallaxe Responsive sur IE 6 ». Vos parcelles de cheveux arrachés et votre psychanalyste se souviennent parfaitement de votre dernière mission d'intégration de formulaires et de l'angoisse qu'elle a engendrée.

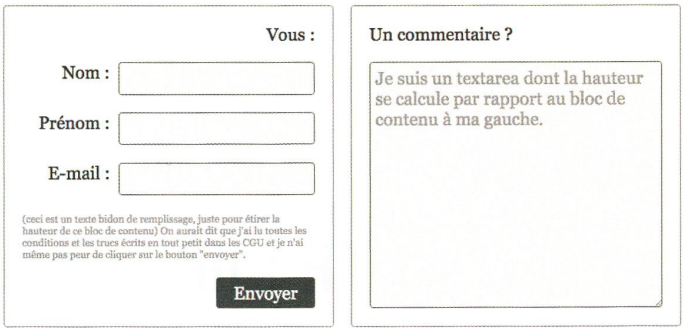
L'élément `<input>` à lui seul et selon certaines sources mal informées, proviendrait directement du cœur des Enfers tant il est impossible à dompter. C'est très simple : il n'occupe jamais la taille souhaitée. Jetez-moi la première pierre si vous n'avez jamais commis un `width: 98.752%` sur un `<input>` récalcitrant en priant pour que « ça rentre ».

Grâce à Flexbox, toutes ces péripéties seront bien vite un lointain mauvais souvenir. Le potentiel de flexibilité de ce module risque bien de vous redonner le moral et la foi dans tous les prochains formulaires à venir.

Objectif à atteindre

Voici le visuel de ce que nous allons obtenir :

Figure 9–1
L'affichage final du formulaire



Globalement, le principe et le comportement souhaités sont ceux-ci :

- les deux parties (gauche et droite) du formulaire ont toujours la même hauteur ;
- les champs `<input>` doivent occuper le reste de l'espace disponible ;
- l'élément `<textarea>` doit occuper toute la hauteur disponible au sein de son parent ;
- sur écran de petite taille, les blocs doivent s'afficher verticalement et les commentaires doivent apparaître en priorité.

Le code HTML

Voici la structure HTML constituant notre formulaire de travaux pratiques :

```
<form action="">
  <div>
      <p>Vous :</p>
      <p>
          <label for="nom">Nom : </label>
          <input type="text" id="nom" name="nom">
      </p>
      <p>
          <label for="prenom">Prénom : </label>
          <input type="prenom" id="prenom" name="prenom">
      </p>
      <p>
          <label for="email">E-mail : </label>
          <input type="email" id="email" name="email">
      </p>
```

```
    <p><small>(ceci est un texte bidon de remplissage, juste pour étirer la hauteur
de ce bloc de contenu). On aurait dit que j'ai lu toutes les conditions et les trucs
écrits en tout petit dans les CGU et je n'ai même pas peur de cliquer sur le bouton
« Envoyer ».</small></p>
    <p>
        <input type="submit" value="Envoyer">
    </p>
  </div>
  <div>
    <p><label for="comm">Un commentaire ?</label></p>
    <textarea id="comm" name="comm" cols="10" rows="5">Je suis un textarea dont la
hauteur se calcule par rapport au bloc de contenu à ma gauche.</textarea>
  </div>
</form>
```

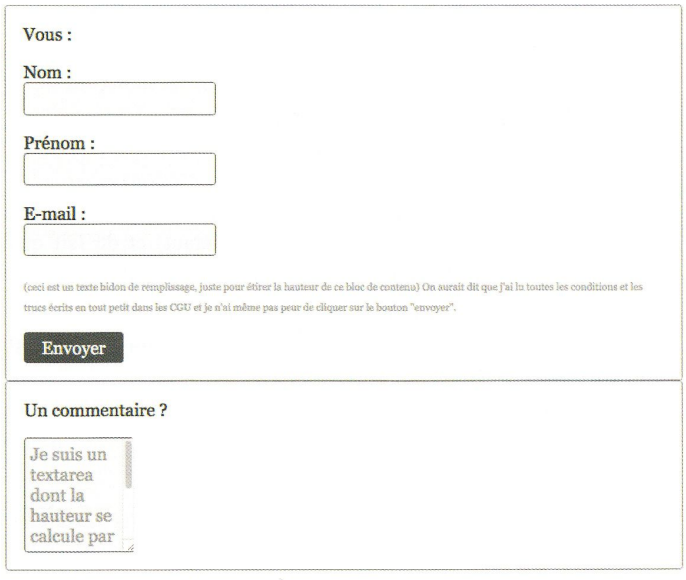


Figure 9–2
Le formulaire de départ, stylé de manière minimaliste

REMARQUE Pas de <fieldset> ?

Vous avez judicieusement constaté que j'ai opté pour un conteneur neutre, `<div>`, plutôt qu'un `<fieldset>` au sein de mon exercice. La raison de ce choix vient du fait que les styles applicables sur `<fieldset>` sont laissés à la discrétion du navigateur. À ce jour, certains éléments tels que `<fieldset>` ou `<button>` ne sont pas affectés par `display: flex` ni `inline-flex` car cela entre en conflit avec les styles système « UI » du navigateur. Il s'agit d'une déficience connue et qui devrait être résolue prochainement, peut-être est-ce d'ailleurs déjà le cas au moment où vous lisez ces lignes.

Étape 1 : les deux de même hauteur

Notre première étape consiste à harmoniser la hauteur des deux blocs de contenu principaux. Une seule propriété suffit à obtenir ce comportement :

```
form {
  display: flex;
}
```

L'explication est simple : avec un parent doté de `display: flex;` les deux enfants `<div>` se placent automatiquement l'un à côté de l'autre (`flex-direction: row` par défaut) et occupent instantanément la même hauteur (`align-items: stretch` par défaut).

Je peux poursuivre en indiquant les largeurs et les espacements entre chacun des deux blocs :

```
form > div {
  width: 50%;
}
form > div + div {
  margin-left: 1em;
}
```

Notez que, quelle que soit la valeur de la marge entre les deux blocs, rien ne cassera : les blocs profitent de leur `flex-shrink: 1` (par défaut) et du fait qu'il n'y a pas de `flex-wrap` pour se réduire équitablement, même si on leur indique une largeur `width` de `50%`.

Nous aurions également très bien pu remplacer `width: 50%` par `flex: 1` mais cela aurait été trop facile et je n'aurais alors plus pu vous expliquer le paragraphe précédent !

Figure 9–3
Les deux principaux sont positionnés.

Étape 2 : les <input> de largeur restante

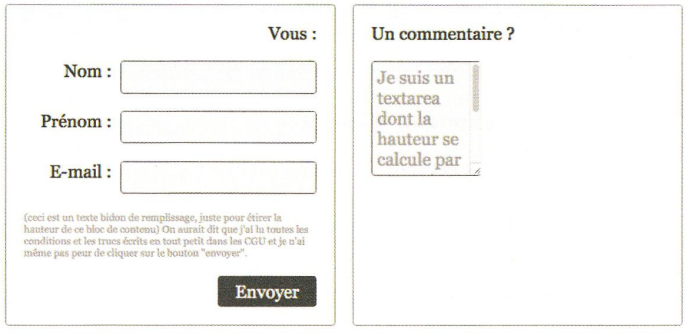
Passons à présent aux `<input>` dont la largeur `auto` est loin de nous convenir.

Leur sort est vite réglé. Un contexte `flex` sur leur parent pour les rendre flex-item, puis un simple mais toujours efficace `flex: 1` fait l'affaire : ils absorbent l'espace restant dans leur ligne. Et hop, emballé c'est pesé !

```
div > p {
  display: flex;
}
form input:not([type=submit]) {
  flex: 1;
}
```

Pour éviter des lourdeurs et des surcharges de CSS, j'ai opté pour le sélecteur `:not()` pour exclure les `<input>` de type `submit`. Ma règle fonctionnera donc même en ajoutant de nouveaux `<input>` exotiques non prévus initialement.

Figure 9–4
Les <input> sont devenus fluides.



Étape 3 : le <textarea> de hauteur restante

Le `<textarea>` ne va guère offrir de résistance non plus : un contexte `flex` sur le second `<div>` accompagné d'une distribution verticale, puis un classique `flex: 1` sur le `<textarea>` et le voilà occupant tout l'espace disponible en hauteur. Facile.

```
div + div {
  display: flex;
  flex-direction: column;
}
textarea {
  flex: 1;
}
```

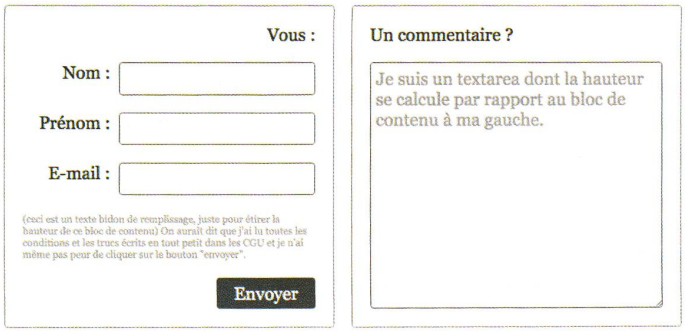


Figure 9–5
Le formulaire finalisé, avec son <textarea> fluide

Étape 4 : le bonus Responsive

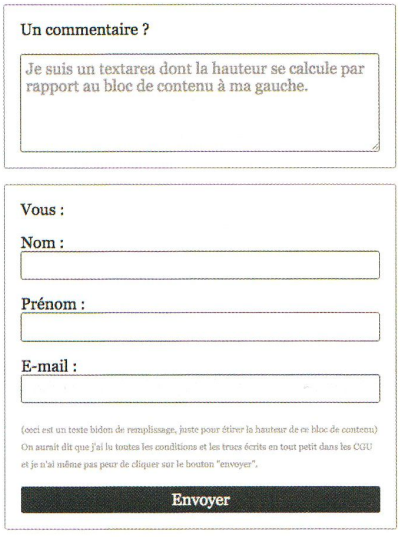
Pour finir en beauté, occupons-nous de la couche de Responsive Webdesign.

Sur petit écran, on modifie simplement la direction d'affichage au sein du formulaire, on rétablit le comportement bloc par défaut des paragraphes et, si on est joueur, on inverse l'ordre d'affichage des deux `<div>` à l'aide d'un `order: -1` sur le second :

```
@media (max-width: 600px) {
  form {
    flex-direction: column;
  }
  form p {
    display: block;
  }
  label {
    display: block;
    text-align: left;
  }
  form > div {
    width: auto; /* si vous avez mis width: 50% avant */
  }
  form > div + div {
    margin-left: 0;
    margin-bottom: 1em;
    order: -1;
  }
  input {
    width: 100%
  }
}
```

Vous voyez que ce n'était vraiment pas la peine d'en faire toute une histoire et que l'on va tranquillement pouvoir passer à des choses un peu plus croustillantes.

Figure 9–6
La version « petits écrans »



Si cela vous intéresse, sachez que le résultat est visible en ligne à l'adresse : cdpn.io/NqPQBa

10

La propriété flex en détail

Où la princesse a déjà mis le dragon au régime « pour mieux entrer dans la niche », dit-elle. Et en lui empruntant sa carte bleue au passage.

À présent que vous êtes aguerri avec la syntaxe usuelle de Flexbox, son fonctionnement et la conception de composants basiques tels que des navigations, des galeries d'images ou des formulaires, passons à un niveau encore supérieur, celui de comprendre (vraiment) la propriété `flex`.

Souvenez-vous de l'une des lois de Flexbox énoncées précédemment :

- Flexbox est un module très simple ;
- … une fois que l'on en retient toutes les propriétés et valeurs non intuitives ;
- … et tant que l'on n'essaye pas de comprendre en détail les propriétés `flex-grow`, `flex-shrink` et `flex-basis`.

Eh bien parlons-en justement, de `flex-grow`, `flex-shrink` et `flex-basis`.

Valeurs courantes de flex

Pour débuter en douceur, découvrons quelques valeurs de `flex` répertoriées au sein des spécifications :

- `flex: initial` est identique à `flex: 0 1 auto` ;
- `flex: auto` est identique à `flex: 1 1 auto` ;

- `flex: none` est identique à `flex: 0 0 auto` ;
- `flex: nombre` est identique à `flex: nombre 1 0%`.

▸ http://www.w3.org/TR/css3-flexbox/#flex-common

REMARQUE Pourquoi une unité « % » ?

Dans la syntaxe raccourcie de flex, la valeur de flex-basis est fixée à 0 %, et non 0. Ceci est dû à une version précédente des spécifications demandant explicitement une unité lorsque la valeur est zéro pour « éviter toute ambiguïté ». Cette règle n'existe plus, mais certains navigateurs tels qu'Internet Explorer nécessitent toujours cette unité pour appliquer la propriété.

Il me paraît important de rappeler qu'un élément flex-item par défaut, donc avec la valeur `flex: initial` :

- ne peut pas s'étirer (`flex-grow` vaut `0`) ;
- peut se réduire au besoin (`flex-shrink` vaut `1`) ;
- est initialement doté d'une dimension égale à son contenu minimal (`flex-basis` vaut `auto`).

REMARQUE flex: auto moins risqué que flex: 1 ?

Entre les syntaxes `flex: 1` et `flex: auto`, une différence de taille réside dans la valeur calculée de `flex-basis`. Dans le premier cas, elle passe à `0` tandis que dans le second elle est conservée à `auto`.
Lorsque `flex-basis` vaut `0` et que `flex-direction` bascule en colonnes, notamment en Responsive, j'ai pu constater des chevauchements d'éléments sur certains navigateurs (notamment Internet Explorer) car ceux-ci considèrent que la hauteur des éléments doit être nulle.
Dans certains cas, lorsque la valeur du facteur d'agrandissement vous importe peu, je vous conseille – maintenant que nous la connaissons – de privilégier la forme `flex: auto`, afin de conserver la valeur `auto` de `flex-basis`. C'est à vous de voir !

Quelques mythes débusqués

Histoire de vous aguicher quelque peu avant les grandes révélations qui vont suivre, voici quelques affirmations considérées comme évidentes (n'est-ce pas ? :)), mais qui se révèlent bien loin de l'être en pratique.

- Si tous les enfants ont un `flex: 1`, alors ils auront la même taille finale, quelle que soit leur taille initiale.

  → Oui, c'est vrai.
- Si tous les enfants ont un `flex-grow: 1`, alors ils auront la même taille finale, quelle que soit leur taille initiale.

  → Eh bien non, pas forcément.
- Un élément `flex-grow: 2` est plus grand qu'un élément `flex-grow: 1`.

  → Pas forcément non plus.

- Il est impossible pour un être humain normalement constitué de se souvenir de la formule exacte du calcul de la propriété `flex-shrink`.
  → C'est tout à fait vrai.

Attention terrain glissant !

Les spécifications Flexbox, en guise d'introduction à la section « Flex layout algorithm » (algorithme d'affichage de Flexbox), débutent par ce petit avertissement que je trouve fort charmant et bien à propos : « *Note : This section is mainly intended for implementors. Authors writing web pages should generally be served well by the individual property descriptions and do not need to read this section unless they have a deep-seated urge to understand arcane details of CSS layout.* »

▸ http://dev.w3.org/csswg/css-flexbox/#layout-algorithm

En français (et en extrapolant légèrement), cela signifie : « *Cette partie des spécifications est destinée aux constructeurs de navigateurs et non aux webdesigners ou aux intégrateurs. Si vous n'êtes pas un chevalier Jedi bien rôdé, ne tentez pas de comprendre dans le détail chaque propriété mais contentez-vous d'utiliser la propriété raccourcie* `flex` *sans trop vous poser de questions.* »

Et je dois bien avouer que – une fois n'est pas coutume – les spécifications n'ont pas forcément tort.

Pour comprendre dans quoi nous nous embarquons, commençons par débroussailler la notion d'espace restant au sein d'un parent.

L'espace restant

En théorie, la notion d'espace restant au sein d'un conteneur ne semble pas d'une complexité inabordable : si ses enfants n'occupent pas toute la surface, alors il reste de la place. Rien n'est plus élémentaire.

En pratique et dans le monde d'un flex-container, c'est sensiblement plus compliqué et cela nécessite quelques mises au point préalables :

- l'espace restant au sein d'un flex-container est (uniquement) calculé à partir des valeurs de `flex-basis` de ses enfants ;
- les propriétés `width`, `min-width` ou `max-width` des flex-item ne participent pas au calcul de l'espace restant ;
- l'espace restant peut être positif ou négatif selon la valeur totale des `flex-basis`. Il y aura alors soit de l'espace en trop, soit de l'espace manquant pour les flex-item.

Prenons par exemple un flex-container de largeur `1000px` contenant un seul flex-item :

- si, pour ce flex-item, `flex-basis` vaut `0`, alors l'espace restant est de 1 000 px ;
- si `flex-basis` vaut `200px`, alors l'espace restant est de 800 px ;

- si `flex-basis` vaut `1400px`, alors l'espace restant est de -400 px ;
- si `flex-basis` vaut `auto`, alors l'espace restant dépend de la taille du contenu de l'enfant ; il est de `1000px-(largeur contenu du flex-item)`.

Si plusieurs flex-item sont présents, il faut bien entendu additionner les valeurs de tous les `flex-basis` afin de calculer l'espace restant positif ou négatif.

Figure 10–1
Un élément de flex-basis: 400 px dans un conteneur de 1 000 px

Une fois l'espace restant déterminé, nous pouvons procéder au calcul de `flex-grow` et (si on est très motivé) de `flex-shrink`, pour connaître les dimensions finales des éléments flex-item.

Vous vous familiariserez avec la notion d'espace restant à travers cette démonstration éditable sur CodePen :
▸ http://codepen.io/raphaelgoetter/full/bdYQML

Calcul de flex-grow

Pour rappel, la propriété `flex-grow`, dont la valeur initiale est `0`, permet à un enfant flex-item de s'élargir pour occuper une proportion de l'espace restant.

La taille d'un élément doté de `flex-grow` est régie par les trois principes suivants :

- l'espace restant est réparti entre tous les flex-item disposant d'une valeur positive de `flex-grow` ;
- la répartition s'opère par pondération : plus la valeur de `flex-grow` est importante, plus l'élément absorbe de l'espace restant, proportionnellement aux autres éléments flexibles ;
- la portion récupérée par chaque élément flexible s'ajoute à la valeur de la propriété `flex-basis`. Le total représente la dimension finale de l'élément.

La formule magique

Si vous êtes féru de mathématiques, ou tout simplement curieux, voici la formule calculant exactement la largeur finale d'un élément flex-item dont le `flex-grow` n'est pas nul :

```
taille finale =
flex-grow / (somme des flex-grow) * (largeur du flex-container - somme des flex-
basis) + (flex-basis de l'élément)
```

Vous conviendrez avec moi qu'il s'agit d'une bien belle formule et que ses concepteurs ont très certainement dû passer quelques soirées bien studieuses (voire bien arrosées) pour la rédiger. Mais ça, c'est sans connaître la formule – bien plus alambiquée encore – de `flex-shrink` !

flex-grow n'est pas flex !

Cela ne paraît pas évident au premier abord, mais il est fondamental de comprendre que le fonctionnement de `flex-grow` et celui de `flex` peut s'avérer radicalement différent appliqué à un élément que l'on souhaite rendre flexible.

En effet, selon la propriété choisie, la valeur de `flex-basis` diffère :

```
.choucroute {
  flex: 1; /* flex-basis = 0% */
}
.knack {
  flex-grow: 1; /* flex-basis = … auto */
}
```

Même si vous n'avez pas appris la formule par cœur, vous savez que le calcul de la taille de l'élément tient compte de la valeur de `flex-basis` et vous conviendrez avec moi que `flex: 1` n'est bel et bien pas systématiquement identique à `flex-grow: 1`.

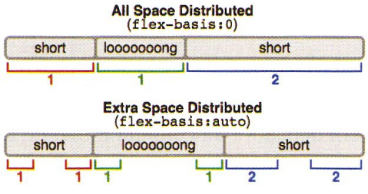


Figure 10–2
La différence entre flex-basis: 0 et flex-basis: auto
(source : W3C specifications)

Quelques exemples concrets

Pour être sûr d'avoir bien compris ce mécanisme important du module Flexbox, passons `flex-grow` à travers la moulinette de cinq cas concrets.

Le principe est celui-ci : pour chacun des exemples suivants, le flex-container a une largeur de 1 000 pixels et contient deux flex-item. Selon les propriétés appliquées aux enfants, tentons de déterminer leur taille finale.

Cas 1. Chaque enfant est doté de flex: 1

Figure 10–3
Deux enfants en flex: 1

La taille finale de chaque élément est, systématiquement, de 500 px :

- `flex: 1` est identique à `flex: 1 1 0%` ;
- pour chaque enfant, la valeur de `flex-basis` est donc de `0%`, la troisième valeur de la propriété raccourcie `flex` ;
- l'espace restant est donc de `(1000-0)=1000px` ;

- tous les éléments ayant une valeur de `flex-grow` identique, ils se répartissent équitablement cet espace restant, soit `+500px` chacun, qu'ils ajoutent à leur `flex-basis` ;
- la taille finale de chaque élément est donc `flex-basis+500px`, soit 500 px.

Cas 2. flex: 1 0 200px et flex: 1 0 400px

Figure 10–4
flex: 1 0 200px et flex: 1 0 400px

La taille finale du premier enfant est 400 px, celle du deuxième est 600 px :

- le total des `flex-basis` est de 600 px ;
- l'espace restant est donc de `(1000-600)=400px` ;
- chaque élément ayant la même valeur de `flex-grow`, ils se répartissent équitablement cet espace restant, soit `+200px` chacun ;
- la taille finale du premier enfant est donc `200+200`, soit 400 px, celle du deuxième est `200+400`, soit 600 px.

Cas 3. flex: 1 et flex-grow: 1

Figure 10–5
flex: 1 et flex-grow: 1

La taille finale des flex-item dépendra du contenu initial du second flex-item. Supposons que la largeur du contenu du second flex-item soit calculée à 126 px :

- `flex: 1` est identique à `flex: 1 1 0%`, tandis que `flex-grow: 1` est identique à `flex: 1 1 auto` ;
- pour le premier enfant, la valeur de `flex-basis` est `0%` ; pour le second, elle est `auto` (c'est-à-dire la largeur du contenu) ;
- l'espace restant est donc de `1000-(largeur contenu item 2)`, ici `1000-126=874px` ;
- chaque élément ayant une valeur de `flex-grow` identique, ils se répartissent équitablement cet espace restant, soit `+437px` chacun ;
- la taille finale du premier enfant est `437+0`, soit 437 px, celle du deuxième est `437+126`, soit 563 px.

Cas 4. flex: 1 0 200px et width: 200px

Figure 10–6
flex : 1 0 200px et width : 200px

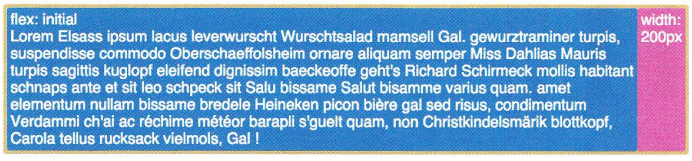
La taille finale du premier enfant est 800 px, celle du second, 200 px :

- pour le premier, la valeur de `flex-basis` est 200 px, pour le second elle est `auto` (c'est-à-dire la largeur du contenu) ;
- l'espace restant est donc de `1000-200-(largeur contenu item 2)`, ici `1000-200-200`, soit 600 px ;

- la taille finale du premier enfant est donc `200+600`, soit 800 px, et celle du second est 200 px.

Cas 5. Pas de flex et width: 200px

Figure 10–7
flex-basis: auto et width: 200px



La taille finale des éléments dépend de leur contenu. Si le premier est très large, le second occupe moins de 200 px :

- pour les deux flex-item, la valeur de `flex-basis` est `auto` (c'est-à-dire la largeur du contenu) ;
- l'espace restant est donc de `1000-(largeur des contenus) ;`
- la taille initiale du deuxième élément correspond à `width`, qui a le pouvoir d'écraser la valeur `auto` de `flex-basis`. Cependant, le contenu du premier item étant très important, l'espace restant est négatif ;
- les éléments pouvant se rétrécir par défaut et ne pouvant pas passer à la ligne, vont absorber cet espace négatif en se réduisant autant que possible, c'est-à-dire à la taille du mot le plus long pour le deuxième flex-item.

Calcul de flex-shrink

OUF, ça y est. Nous avons enfin passé le gros morceau de `flex-grow` et je crois que vous vous en êtes sortis indemnes !

Je vais devoir vous annoncer une mauvaise nouvelle : le calcul de `flex-shrink` est encore beaucoup plus alambiqué que celui de `flex-grow`…

Avant de vous perdre définitivement avec la formule, je vous rappelle que la propriété `flex-shrink`, dont la valeur par défaut est `1`, est un facteur de compression, grâce auquel un enfant flex-item se réduit automatiquement en absorbant l'espace restant quand celui-ci est négatif.

Également pour rappel : lorsque `flex-wrap` vaut `wrap` ou `wrap-reverse`, alors le comportement des éléments sera toujours de passer à la ligne avant d'appliquer `flex-shrink` (voir « principes fondamentaux », pages 60-61).

La formule magique

Comme vous allez le constater, le calcul de la largeur finale d'un élément `flex-shrink` est d'une simplicité élémentaire pour un docteur en physique quantique moléculaire ou pour un

rédacteur de spécifications. Toutefois, un être humain tel que vous ou moi risque d'y laisser quelques neurones au passage.

```
taille finale =
(flex-shrink * flex-basis) / (somme des (flex-shrink * flex-basis)) * (largeur du
flex-container - somme des flex-basis) + (flex-basis de l'élément)
```

Si je puis me permettre un conseil avisé : ne vous souciez pas vraiment de ce qui se passe derrière les rouages de `flex-shrink` ; employez cette propriété pour autoriser les éléments à se réduire, tout simplement.

Des ressources bien pratiques

Pour finir ce chapitre, je vous propose trois ressources en anglais sur `flex-grow` et `flex-shrink`. Elles vous permettront de vous immiscer encore plus loin dans la compréhension de ces deux propriétés.

- *Flexbox Adventures* : un article très détaillé sur Flexbox et en particulier sur les calculs de `flex-grow` et `flex-shrink`.
  http://bit.ly/flexbox-adventures
- *Understanding Flexbox* : `flex-grow` et `flex-shrink` poussés dans leurs retranchements.
  http://bit.ly/understanding-flexbox
- *Flexbox tester* : du même auteur que le lien précédent, un bac à sable interactif permettant de tester les largeurs finales obtenues avec toutes les explications nécessaires. Indispensable !
  http://bit.ly/flexbox-tester

11

Modèles de design

Où il est question de gouttières récalcitrantes, de rénovation de donjon et de redécoration intérieure « pour accueillir mes copines licornes ».

À présent que les arcanes des Flexbox n'ont plus de secrets pour nous, poursuivons sur notre lancée en nous attardant sur divers schémas de positionnement d'éléments. L'objectif avoué est de récolter diverses techniques pour obtenir des grilles de mise en page stables et propres.

À travers les modèles suivants, nous allons véritablement comprendre comment interagissent les propriétés de Flexbox :

- répartir tous les éléments sur une ligne sans retour à la ligne possible ;
- répartir un nombre défini d'éléments sur une ligne en forçant le retour ;
- répartir les éléments automatiquement sans se soucier de leur écart.

Pour chacune des situations, nous allons tenter de définir un écart volontaire entre les « colonnes », ce que l'on appelle une « gouttière ».

Répartition sur une ligne

Figure 11–1
Répartition des éléments sur une ligne

Principe et objectif

Je souhaite répartir un nombre inconnu d'éléments sur une unique ligne. Quel que soit leur nombre, tous les éléments doivent entrer dans la rangée, aucun retour à la ligne n'est possible.

Code nécessaire

Les styles CSS correspondant à notre besoin pourraient se résumer à ce qui suit :

```
.container {
  display: flex;
}
.container > * {
  flex: 1; /* idem flex: 1 1 0% */
}
```

Explications sommaires

Il est difficile de faire plus court : un simple `flex: 1` fait le travail.

Les éléments s'agrandissent ou se réduisent si nécessaire. Ils occupent tous la même surface en raison de leur `flex-basis: 0%` et ne peuvent pas passer à la ligne.

Pour information, il est inutile de préciser `width` : en effet, la répartition est forcément équitable à partir du moment où `flex-basis` est égal à zéro et où `flex-grow` est identique pour chaque élément.

Une gouttière ?

Trop facile !

Puisque tous les éléments sont alignés, il est plutôt aisé de cibler tous les enfants sauf le premier à l'aide d'un ancien sélecteur d'adjacence directe, puis d'appliquer une marge de la taille souhaitée.

```
.container > * + * {
  margin-left: 10px; /* taille de gouttière */
}
```

Kézako « * + * » ?

Le symbole * consiste en un sélecteur universel en CSS. Son effet est de cibler n'importe quel élément HTML. Associé au sélecteur d'adjacence +, il peut faire des miracles. La combinaison *+* signifie « tout élément suivant un autre élément », en clair « tous sauf le premier ».

Figure 11–2
Répartition des éléments sur une ligne, avec une gouttière

Question pour des champions comme vous

Si je vous posais la question suivante : dans ce contexte, `flex-wrap` fonctionne-t-il ou non ? Que me répondriez-vous ?

La réponse est très exactement « Non » parce que `flex-basis` vaut `0` (car nous avons affublé les flex-item d'un `flex: 1`). J'ai failli vous avoir, avouez.

Répartition en forçant le retour à la ligne

Nous pourrions souhaiter, à juste titre, que les éléments passent à la ligne plutôt que de se comporter comme des cellules de tableau. Ainsi, nous nous rapprocherions du concept de grille classique.

Il y a deux moyens de forcer le déclenchement de `flex-wrap`.

1. Imposer une largeur minimale aux éléments via `min-width`.
2. Passer en `flex-basis: auto` et imposer une largeur `width`.

La solution min-width

La première option impose une largeur minimale aux éléments, sans pour autant empêcher qu'ils s'agrandissent au besoin :

```
.flow-3 {
  display: flex;
  flex-wrap: wrap;
}
.flow-3 > * {
  flex: 1 1 33.3333%;
  min-width: 320px;
}
```

Dans ce code, la classe `.flow-3` désigne le flex-container et `.flow-3 > *`, l'ensemble de ses flex-item.

La propriété `min-width` prend le dessus sur la loi Flexbox concernant le passage à la ligne : les éléments ne pouvant pas se réduire en deçà de 320 px de largeur peuvent passer à la ligne.

Il s'agit du moyen le plus simple d'obtenir une disposition de trois éléments qui se répartiront de manière naturellement Responsive sans même nécessiter de Media Queries !

L'inconvénient de cette solution est que si la dernière ligne compte un nombre d'éléments différent de 3, par exemple 2 ou 1, `flex-grow` va absorber plus d'espace que souhaité et que sur les lignes précédentes, ce qui aura une incidence sur la taille des éléments.

Figure 11–3
Forcer le retour à la ligne avec min-width

La solution width

La seconde option, via `width`, empêche les éléments d'absorber l'espace restant :

```
.flow-3 > * {
  flex: initial /* idem flex: 0 1 auto; */
  width: 33.3333%;
}
```

`width` est opérationnel à partir du moment où `flex-basis` vaut `auto`. `flex-wrap` s'applique alors également et les éléments sont autorisés à passer à la ligne.

Figure 11–4
Forcer le retour à la ligne avec width

Marges et gouttières

À partir du moment où les éléments peuvent se répartir sur plusieurs lignes, cela devient tout de suite un peu plus sportif puisqu'un simple sélecteur `* + *` n'est plus du tout suffisant.

Il faut donc ruser habilement en jouant sur le fait que l'on connaît le nombre d'éléments par ligne, ici trois, et cibler une suite connue : le 4e, le 7e, le 10e, etc. pour leur supprimer la marge à gauche.

La formule `:nth-of-type(3n+1)` du sélecteur d'enfants est parfaite pour des lots d'éléments disposés par rangées de trois :

```
.flow-3 > * {
  flex: initial; /* idem flex: 0 1 auto; */
  width: 32%;
  margin-left: 2% /* taille de gouttière */
}
.flow-3 > *:nth-of-type(3n+1) {
  margin-left: 0;
}
```

Figure 11–5
Forcer le retour à la ligne avec width, avec gouttières

Le truc en plus : calc()

Notre gouttière entre éléments est ici de `2%`, d'où la valeur de `width: 32%` pour couvrir le total de 100 %.

Pourrait-on mettre en place des gouttières exprimées en unités variées telles que des `px`, des `em` ou des `rem` si nous en avions la contrainte ? C'est tout à fait possible avec la valeur-fonction `calc()`, reconnue par tous les navigateurs modernes et depuis IE 9.

Voici un exemple pour une gouttière de 40 pixels :

```
.flow-3 > * {
  width: calc(100% / 3 - 40px);
  margin-left: 40px; /* taille de gouttière */
}
```

Le navigateur se chargera de réaliser le calcul à votre place.

> **REMARQUE** **Une marge de trop ?**
>
> Il ne vous aura pas échappé qu'en l'état, il subsiste une marge à gauche du premier élément. Rassurez-vous : ce problème sera envisagé et résolu dans le prochain chapitre consacré aux grilles de mises en page.

Automatiser en production

Les différents cas de figure que nous venons de résoudre nécessitent de connaître à l'avance le nombre d'éléments par rangée et de prévoir une classe correspondante (`.flow-2`, `.flow-3`, `.flow-4`, etc.). Ce n'est bien évidemment pas idéal dans un environnement de production.

Sachez que des outils existent pour vous libérer de cette tâche ingrate : les préprocesseurs CSS tels que LESS ou Sass.

Traiter ces outils en profondeur dépasse les objectifs de ce livre, mais renseignez-vous sur le sujet : il est possible – et conseillé – d'automatiser tous les calculs et de générer dynamiquement les classes CSS grâce à ces métalangages que sont LESS ou Sass.

Voici un exemple d'automatisation à l'aide de Sass. Ici, la taille de gouttière ainsi que le nombre de « colonnes » sont exprimés à l'aide de variables intégrées au sein d'une fonction (appelée aussi *mixin*) :

```
// Mixin pour colonnes de largeurs égales

$number : 4 ;
$gutter : 1em ;
```

```
@mixin grid($number:$number,$gutter:$gutter) {
  & > * {
    width: calc(100% * 1 / #{$number} - #{$gutter});
  }
}
```

Pour vous permettre d'aller plus loin sur ce thème de l'automatisation à l'aide de préprocesseur, je vous invite à consulter le tutoriel *Une grille responsive avec CSS 3 Flexbox et LESS (ou Sass)*, que vous trouverez à l'adresse raccourcie http://kiwi.gg/grille.

Répartition automatique (Autoflow)

Figure 11–6
Répartition « autoflow » grâce à space-between

Principe et objectif

Cette fois-ci, je ne vais même pas définir la gouttière entre les éléments. Tout ce qui m'importe est de répartir un nombre inconnu d'éléments à raison de *N* par ligne, en autorisant le passage à la ligne et le plus simplement possible sans opérer de savants calculs.

Pour information, c'est le mode de positionnement principal employé sur mon site personnel goetter.fr.

Code nécessaire

Les styles appliqués évoquent les quelques expériences précédentes, mais nous allons cette fois tirer parti d'une valeur encore trop peu exploitée, `space-between` :

```
.autoflow-3 {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
}
.autoflow-3 > * {
  width: 32% /* ou flex: 0 0 32%; */
}
```

Explications sommaires

`width` impose une largeur fluide identique à chaque élément, ici `32%` afin de profiter d'un espace entre chaque élément. Cet espace à se répartir est exactement de `4%` (le détail du calcul étant `100%-32%*3`).

Grâce à `justify-content: space-between`, les éléments sont automatiquement répartis au sein de leur ligne, tel un texte dont le contenu serait justifié.

> **REMARQUE Un sélecteur plus générique ?**
>
> Le choix du sélecteur de classe `.autoflow-3` est relativement restrictif et vous impose de prévoir d'autres classes pour chaque cas de figure, par exemple `.autoflow-2`, `.autoflow-4`, etc. Une bonne façon de s'affranchir de cette fastidieuse étape est d'opter pour un sélecteur d'attribut, tel que `[class^="autoflow-"], [class*="autoflow-"]` qui se chargera tout seul de cibler les éléments dont la classe commence ou contient la chaîne « autoflow- ».

Voulez-vous rendre cela Responsive rapidement ? L'opération se réalise avec deux Media Queries :

```
/* petits écrans */
@media (max-width: 640px) {
  [id=main] > * {
    width: 49%;
  }
}

/* très petits écrans */
@media (max-width: 480px) {
  [id=main] > * {
    width: 100%;
  }
}
```

> **REMARQUE Un sélecteur > *, kézako ?**
>
> Le sélecteur employé dans cette portion de CSS, `[id=main] > *` désigne littéralement « tous les enfants directs de #main quels qu'ils soient ».

Limitations de cette technique

En vertu du choix d'alignement justifié, notre grille d'affichage n'est ici parfaite que lorsque le nombre total des éléments est un multiple de 3.

Dans le cas contraire, les éléments de la dernière ligne (qui peuvent être 1 ou 2) seront disposés différemment de leurs prédécesseurs, ce qui n'est pas forcément souhaité.

Et les gouttières ?

Les espaces entre les éléments répartis selon la valeur `space-between` sont forcément automatiques, ce qui est à la fois pratique et contraignant, car il ne sera pas possible de les fixer.

Figure 11–7
Répartition justifiée de la dernière rangée

L'astuce en plus

Il y a au moins deux moyens de pallier le problème d'alignement des éléments de la dernière ligne de contenu.

La première solution consiste à générer un pseudo-élément invisible mais qui occupera l'espace d'un élément de la grille :

```
.autoflow-3::after {
  content: "";
  flex: 0 0 32%;
}
```

Cette astuce ne permet de combler artificiellement qu'un seul « trou » et ne sera adaptée que pour des grilles de répartitions par 2 ou 3 éléments.

Pour des répartitions à plus large échelle, à partir de 4 éléments, il sera nécessaire d'opter pour une autre solution : celle d'ajouter manuellement au moins un élément HTML vide faisant office de bouche-trou.

```
...
<div>9</div>
<div>10</div>
<span></span> <!-- fake element -->
<span></span> <!-- fake element -->
<span></span> <!-- fake element -->
```

Notez que le nombre d'éléments bouche-trous importe peu. Cela fonctionnera même si vous en prévoyez plus que de raison, à condition de veiller à ce qu'ils n'occupent pas plus de place que nécessaire :

```
 .autoflow-3::after {
  content: "";
  flex: 0 0 32%;
  padding: 0;
  margin: 0;
  background: none;
}
```

12

TP : construction de grilles

Où l'on apprend que l'origine du grimoire à succès « Cinquante nuances de grilles » provient d'un malentendu sur les horaires d'ouverture des herses du donjon.

Nous parvenons à présent au dernier exercice décortiqué de notre ouvrage.

Après avoir parcouru toutes les possibilités techniques offertes par Flexbox, ainsi que les écueils et les limites de ce positionnement, nous devrions à présent pouvoir concevoir un modèle d'affichage que vous brûlez certainement de mettre en pratique : une grille de mise en page.

S'il vous est déjà arrivé de côtoyer des frameworks HTML/CSS tels que Bootstrap ou Foundation, la notion de grille d'affichage vous est forcément familière.

Principes généraux

Une grille se doit d'être la plus polyvalente possible afin de répondre à vos besoins, même lorsque ceux-ci deviennent particuliers. Elle le sera vraiment si elle tient compte de toutes les fonctionnalités suivantes :

- **répartition équitable ou non :** la taille de chaque colonne est automatique (toutes les colonnes sont égales) ou fixée indépendamment ;
- **gouttières :** les colonnes sont espacées par une gouttière dont la largeur est automatique ou fixée ;
- **pull et push (offset) :** un « vide » est laissé avant ou après une colonne ;

- **éléments « à la une » :** certains éléments sont mis en exergue et disposés avant les autres ;
- **imbrications :** une grille est susceptible d'être imbriquée dans une autre grille ;
- **responsive :** la grille s'adapte à toutes les surfaces d'affichage ;
- **simple :** le top du top est bien entendu de cumuler ces fonctionnalités tout en demeurant très simple à l'usage.

Des grilles, des grilles, des grilles

Si vous pensiez être le premier à envisager la construction de grilles de mise en page à l'aide de *Flexbox Layout*, j'ai une mauvaise nouvelle pour vous : des dizaines et des dizaines de webdesigners du monde entier ont déjà exploité ce filon avant vous avec plus ou moins de réussite.

Je vous ai concocté une petite liste non exhaustive des projets les plus intéressants à l'heure actuelle. N'hésitez pas à les décortiquer, voire à adopter celui qui saura vous convenir dans vos projets.

Frameworks CSS

- Bootstrap 4+ : http://blog.getbootstrap.com/2015/08/19/bootstrap-4-alpha/
- pureCSS : http://purecss.io/
- Foundation 6+ : http://foundation.zurb.com/
- Flexboxgrid : http://flexboxgrid.com/
- Ionic Framework : http://ionicframework.com
- Scriptura : http://scriptura.github.io/Pages/GridLayout.html
- Juiced : http://juicedcss.com/
- KNACSS 4+ : http://knacss.com

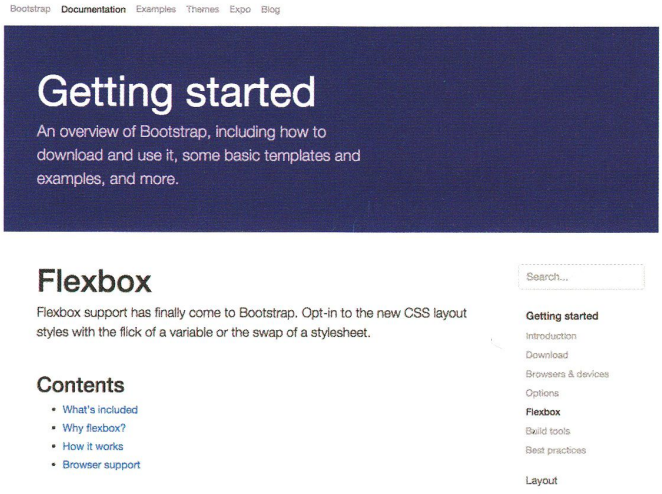


Figure 12–1
L'arrivée de Flexbox dans la documentation Bootstrap 4

Grilles autonomes

Les ressources suivantes se chargent (uniquement) de réaliser des grilles à l'aide de Flexbox :

- o-grid du Financial Times : http://registry.origami.ft.com/components/o-grid
- GridLex : http://gridlex.devlint.fr/
- Batch : https://martskin.github.io/batch
- Core : http://splintercode.github.io/core-flex-grid/
- Lost Grid : https://github.com/corysimmons/lost
- Chewing Grid : https://github.com/tzi/chewing-grid.css
- flexgrid : http://ptb2.me/flexgrid/
- griddleCSS : http://studio51.github.io/gridlecss/
- Levity : https://github.com/w-jerome/levity
- unGrid : http://sherbrow.github.io/ungrid/flex.html
- FlexSassCandy : https://github.com/meerita/flexsasscandy
- Topcoat Grid : https://github.com/topcoat/grid
- Flex Grid Framework : http://flexgridframework.com/

Les bases de la grille

Mon objectif est le suivant : seul le conteneur sera affublé d'une classe et ses enfants (directs) en bénéficieront automatiquement, sans avoir à les nommer spécifiquement. C'est pour l'instant le choix que nous avons fait au sein de notre agence lorsque nous avons construit notre grille et framework associé KNACSS.

Par exemple, le code imaginaire suivant pourrait générer une grille de 4 colonnes où tous les enfants directs occuperaient un quart de l'espace disponible s'il n'y avait pas de gouttière :

```
.container-grid-4 {
  /* ici les styles du conteneur */
}
.container-grid-4 > * {
  /* ici les styles des enfants */
}
```

Ébauche de grille en Flexbox

Commençons simplement par une grille de répartition sur 4 colonnes de largeur égale, via Flexbox et sans définir de gouttière.

Structure HTML

```
<div class="container-grid-4">
  <div>un div ou n'importe quoi d'autre</div>
  <div>un 2è div ou n'importe quoi d'autre</div>
  <div>un 3è div ou n'importe quoi d'autre</div>
  <div>etc.</div>
  <div>etc.</div>
</div>
```

Styles associés

```
/* box-sizing c'est la vie */
* {box-sizing: border-box;}

/* styles du conteneur */
/* affichage horizontal et passage à la ligne */
.container-grid-4 {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
}
/* styles des enfants */
/* largeur fixée à 25% */
.container-grid-4 > * {
  flex: 0 0 auto;
  width: 25%;
}
```

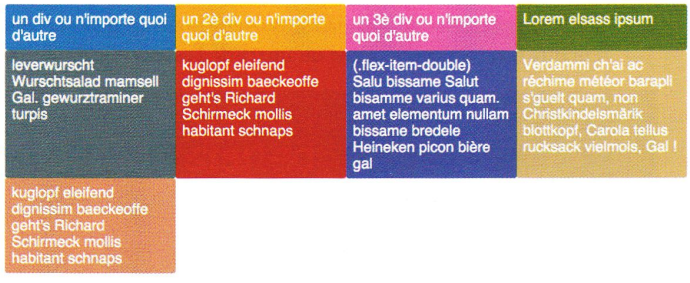


Figure 12–2
Notre grille de 4 colonnes, déjà opérationnelle

Gérer la gouttière

À l'heure actuelle, gérer l'espace entre les éléments d'une grille n'est toujours pas aussi intuitif que cela devrait être. Nous sommes encore obligés de passer par des artifices pour cela.

Une astuce devenue un grand classique, la combinaison `margin-left` négatif sur le parent associé à un `padding-left` sur chaque enfant, est certainement la plus prisée. Cependant, grâce

au modèle particulier de Flexbox, nous allons pouvoir nous affranchir du `padding` et ne conserver comme critère que le `margin`.

Supposons que je souhaite créer quatre colonnes espacées de 20 px. Cette astuce consiste à :

- conférer une largeur (`width`) de un quart du parent à chacun des enfants moins la gouttière : `width: calc(25% - 20px)` (j'ai choisi `width` plutôt que `flex-basis` pour des raisons de compatibilité actuelle) ;
- attribuer un `box-sizing: border-box` pour être sûr que les boîtes ne s'agrandiront pas en ajoutant du `padding` ou un `border` ;
- appliquer un `margin-left` de `20px` sur chaque enfant ;
- appliquer un `margin-left` de valeur négative (`-20px`) sur le conteneur, ce qui aura pour effet d'étirer ses enfants et d'absorber le `margin` du premier enfant.

```
.container-grid-4 {
  margin-left: -20px;
}
.container-grid-4 > * {
  width: calc(25% - 20px);
  margin-left: 20px;
}
```

Variante avec :nth-child()

Si, pour une raison ou une autre, vous ne souhaitez pas appliquer de marge externe négative sur le conteneur principal, il est possible de cibler uniquement les éléments nécessitant une gouttière à l'aide d'une formule `:nth-child(4n+1)` dans le cas où votre grille est prévue pour quatre colonnes.

Cela complique toutefois un peu la perception des largeurs des éléments car il faudra leur appliquer à chacun les trois quarts de la gouttière :

```
.container-grid-4 > * {
  flex: 0 0 auto;
  width: calc(25% - (20px * 3 / 4));
  margin-left: 20px;
}
.container-grid-4 > *:nth-child(4n+1) {
  margin-left: 0;
}
```

Comme vous le voyez, c'est vraiment enfantin si vous avez opté pour la version « simple ». Flexbox se prête parfaitement à ce genre d'exercice car il n'a pas les inconvénients de `float` ou `inline-block` (sortie du flux, espaces « whitespace » indésirables, etc.).

Il nous suffit à présent de dupliquer ce code CSS pour l'adapter à toutes les répartitions possibles : par 2, par 3, par 5, etc.

Figure 12–3
4 colonnes, avec gouttière de 20 pixels

Automatiser avec un préprocesseur

Parvenu à ce stade, un nouveau palier pourrait être franchi : celui d'automatiser la construction de la grille quel que soit le nombre de colonnes souhaité et quelle qu'en soit la valeur de gouttière.

C'est à présent le travail des préprocesseurs tels que LESS ou Sass. J'ai choisi LESS pour notre démonstration.

Créons un *mixin* `.grid(x,y)` que l'on pourrait appliquer à n'importe quel élément HTML conteneur et dont les paramètres `x` et `y` représenteraient respectivement le nombre de colonnes et la largeur de la gouttière. La largeur des enfants en pourcentage serait calculée selon le nombre de colonnes via la formule `calc(100% * 1 / @{number} - @{gutter});`.

Code LESS

```
// config :
@gutter: 1em;
@number: 4;

[class*="grid-"] {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: space-between;
  margin-left: -@gutter;
}

[class*="grid-"] > * {
  flex: 0 0 auto;
  display: block; /* IE fix */
  width: ~'calc(100% * 1 / @{number} - @{gutter})';
  margin-left: @gutter;
}

.grid(@number:@number, @gutter:@gutter) {
  & > * {
    width: ~'calc(100% * 1 / @{number} - @{gutter})';
  }
}
```

Bonus : éléments double ou triple taille

Certains éléments doivent être mis en exergue et occuper deux fois plus d'espace que leurs voisins ? Rien n'est plus simple avec Flexbox : il suffit d'adapter la valeur de `width` voire de l'automatiser avec LESS pour que cette valeur s'adapte quel que soit le nombre de colonnes.

```
.flex-item-double {
  width: ~'calc(100% * 2 / @{number} - @{gutter})';
}
.flex-item-triple {
  width: ~'calc(100% * 3 / @{number} - @{gutter})';
}
```

Figure 12–4
Ajout d'un élément de taille double au sein de notre grille

Bonus : « à la Une »

À présent, pourquoi ne pas continuer avec les bonnes choses ? Grâce à la propriété `order` du positionnement Flexbox, il est très simple d'intervertir l'ordre d'affichage des éléments et de faire remonter un bloc avant les autres. Il suffit pour cela de lui attribuer une valeur de `order` inférieure à la valeur par défaut qu'est `0` :

```
/* sera affiché au début */
.flex-item-first {
  order: -1;
}
/* sera affiché à la fin */
.flex-item-last {
  order: 2;
}
```

Figure 12–5
Ajout d'un élément « à la Une » au sein de notre grille

Bonus : pull et push

Les marges automatiques, que l'on a traitées précédemment, expriment ici tout leur potentiel. En effet, un simple `margin-left: auto` appliqué sur un élément de la grille va le pousser, lui et ses frères suivants, à droite sur leur ligne. A contrario, un `margin-right: auto` va tout pousser à gauche de la ligne. Cela permettra de générer volontairement des espaces vides entre certaines colonnes, ce que l'on appelle des *offsets*.

```
/* poussé à gauche */
.pull {
  margin-right: auto;
}
/* poussé à droite */
.push {
  margin-left: auto;
}
```

Figure 12–6
Un groupe d'éléments poussés à droite de leur rangée

Grille Responsive

Une étape non négligeable à présent consiste à rendre notre construction Responsive. Pour nous conformer à notre philosophie initiale, les informations liées aux tailles des colonnes devront être apportées sur le conteneur et non sur les enfants.

J'ai choisi de définir le nombre de colonnes selon les tailles d'écrans à l'aide des mots-clés suivants :

- `-small-*` qui définit le nombre de colonnes lorsque le point de rupture atteint la valeur de la variable `small-screen` ;
- `-tiny-*` qui définit le nombre de colonnes lorsque le point de rupture atteint la valeur de la variable `tiny-screen`.

La grille suivante s'affichera en 4 colonnes sur grand écran, puis en 3 colonnes sur un écran réduit, puis en une seule colonne sur petit écran :

```
<div class="grid-4-small-3-tiny-1">
  <div>un div ou n'importe quoi d'autre</div>
  <div>un 2è div ou n'importe quoi d'autre</div>
  <div>un 3è div ou n'importe quoi d'autre</div>
  <div>etc.</div>
</div>
```

Styles CSS appliqués

```
/* Grille sur petits écrans */
@media (max-width: 640px) {
  [class*="-small-4"] > * {
    width: calc(100% * 1/4 - 30px);
  }
  [class*="-small-4"] > .flex-item-double {
    width: calc(100% * 1/2 - 30px);
  }
  [class*="-small-3"] > * {
    width: calc(100% * 1/3 - 30px);
  }
  [class*="-small-3"] > .flex-item-double {
    width: calc(100% * 2/3 - 30px);
  }
  ...
}

/* Grille sur très petits écrans */
@media (max-width: 480px) {
  ...
  [class*="-tiny-1"] > * {
    width: calc(100% - 30px);
  }
```

```
    [class*="-tiny-1"] > .flex-item-double {
      width: calc(100% - 30px);
    }
}
```

Résultat final

Nous voici donc arrivés à la fin de notre (agréable) périple. Notre grille est prête et fonctionnelle. Il est même parfaitement possible d'imbriquer une grille dans une grille (mais n'en abusez pas !).

J'espère que le résultat (et sa simplicité) vous convaincra.

Figure 12–7
Notre grille vue sur écran large

Figure 12–8
Notre grille vue sur écran de taille moyenne

Figure 12–9
Notre grille vue sur petit écran

Pour rappel, cette grille est celle actuellement employée sur le framework KNACSS. Nous y avons apporté quelques modifications mineures, mais vous y retrouverez tous les principes décrits dans cet exercice.

13

Encore plus loin avec Flexbox

Où l'on se rend compte qu'il est bien possible que les scénaristes aient prévu de nouveaux rebondissements, voire un nouvel épisode à cette histoire palpitante.

À ce stade, Flexbox n'a presque plus aucun secret à vous révéler. Il demeure toutefois un certain nombre de caractéristiques propres à ce module très particulier qu'il est grand temps de dévoiler.

Des propriétés non appliquées

Quelques propriétés ou pseudo-éléments sont incompatibles avec le modèle d'affichage de Flexbox, généralement parce qu'ils sont rendus inutiles en son sein. Ces particularités peuvent affecter les flex-container ou les flex-item.

- Sur un flex-container :
  - aucune propriété de la famille `column-` (des spécifications « CSS3 columns ») n'a d'effet ;
  - les pseudo-éléments `::first-line` et `::first-letter` ne s'appliquent pas non plus.
- Sur les flex-item :
  - un flex-item ne peut pas être flottant (pas de `float` possible) ;
  - les propriétés `clear` et `vertical-align` ne s'appliquent pas.

Figure 13–1
Pas de flex-item flottant

Un contexte de formatage particulier

Le modèle de boîte flexible dispose intrinsèquement de sérieux atouts lorsqu'il s'agit de dompter ces flottants qui débordent tout le temps.

Les éléments flex-container et flex-item construisent un *Block Formatting Context* (BFC), ou plus exactement un *Flex Formatting Context (FFC)*, dont les avantages sont similaires et décrits dans un article d'Alsacréations.

- Un flottant ne déborde pas d'un flex-container ni d'un flex-item.
- Un flex-container ou un flex-item ne s'écoule pas autour d'un flottant (ils demeurent l'un à côté de l'autre).
- La fusion de marges n'opère pas au sein d'un flex-container.

- http://kiwi.gg/bfc
- http://kiwi.gg/fusion

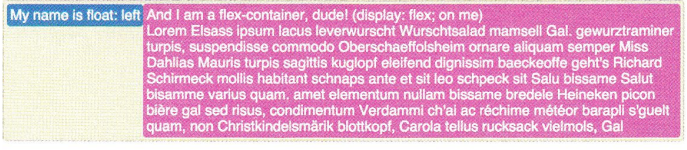


Figure 13–2
Illustration de FFC :
le flex-container ne s'écoule pas autour du flottant.

Des pseudo-éléments embarrassants

Lorsque vous affublez un conteneur d'un `display: flex`, le contexte de Flexbox est appliqué automatiquement à chacun de ses enfants directs… même à certains dont vous ne soupçonniez peut-être pas l'existence.

Parmi ces enfants « non désirés » peuvent se trouver des éléments censés être invisibles (en `display: none`) tels que des balises `<script>`, ou encore plus classiquement des pseudo-éléments comme `::before` ou `::after` qui, bien qu'inexistants dans le DOM, comptent parmi les enfants flex-item.

En devenant des flex-item, tous ces éléments héritent des propriétés Flexbox et des comportements attenants ; leur valeur de `flex` devient par exemple `0 1 auto` et tutti quanti.

Si par le plus grand des hasards, vous avez recours à un framework CSS basé sur des grilles en `float`, ou à une classe `.clearfix` pour échapper des flottants, il y a de fortes chances que votre conteneur génère un pseudo-élément de ce type :

```
.clearfix::after {
  content: "";
  display: table;
  clear: both;
}
```

La présence du pseudo-élément `::after` parmi les enfants flex-item a des conséquences visuellement perceptibles sur certains navigateurs.

Supprimer purement et simplement cet élément ou cette classe n'est pas forcément souhaitable, surtout si vous comptez proposer une alternative flottante aux navigateurs ne reconnaissant pas Flexbox.

Il faut donc ruser et faire disparaître cet élément tout en le laissant actif dans un environnement de `float`. La méthode la plus simple est de forcer `flex-basis` à une taille de `0` :

```
.clearfix::after {
  flex-basis: 0;
  max-width: 0;
}
```

Largeur minimale intrinsèque

Depuis la spécification Flexbox, les propriétés `min-height` et `min-width` bénéficient d'une nouvelle valeur qui est `auto` et qui n'existait pas en CSS 2.1. C'est d'ailleurs la valeur par défaut des éléments flex-item, et non plus `0` comme c'était le cas précédemment.

Cela se traduit par le fait que les flex-item, de par leur `min-width` intrinsèque, ont une largeur par défaut qui ne peut pas être inférieure à celle de leur contenu minimal (`"min-content"`), même avec un `flex-basis: 0`, ou un `flex-grow: 0`.

Figure 13–3
Largeur minimale de contenu préservée même avec un flex-basis: 0

Flexbox et z-index

Les éléments flex-item créent un contexte d'empilement, à l'instar de ce que l'on peut reproduire à l'aide des propriétés `position`, `transform` ou `opacity`.

Cette particularité leur apporte une prise en charge native de la propriété d'empilement `z-index`, ce qui signifie par exemple qu'il est parfaitement inutile de leur appliquer une `position` pour modifier leur ordre d'empilement, comme c'est le cas pour les autres éléments habituels.

Figure 13–4
Un flex-item simplement doté d'un z-index:1

Flexbox et les espaces en pourcentage

Dans le modèle de boîte classique, les valeurs en pourcentage de `margin` et de `padding` présentent une particularité historique : elles se rapportent toujours à la *largeur* du parent.

S'il est assez cohérent de constater qu'un `margin-left: 10%` et un `padding-left: 10%` correspondent à 10 % de la largeur du parent, il est bien plus déroutant de se rendre compte qu'un `margin-top: 10%` et un `padding-top: 10%`, qui sont des verticales, se rapportent aussi à la *largeur* du parent et non à sa hauteur ! Et pourtant, c'est bien ainsi que cela se passe sur l'ensemble des éléments de type bloc habituels.

Les spécifications Flexbox reviennent sur cette apparente aberration et établissent un calcul plus cohérent. Elles disent clairement que les `margin-top`, `margin-bottom`, `padding-top` et `padding-bottom` en pourcentage sur des flex-item doivent se référer à la hauteur de leur parent et non à leur largeur.

À l'heure où ce livre est écrit, tous les navigateurs ne respectent pas encore à la lettre ces directives ; il est donc probable que vous assistiez à des différences d'affichage entre certains d'entre eux. Pour information, Mozilla Firefox a été le premier à adopter l'algorithme correct d'affichage.

Figure 13–5
Les marges en pourcentage selon le type du parent

Flexbox et les animations

Si vous êtes férus d'animations et de transitions CSS en tous genres, vous serez heureux d'apprendre que certaines propriétés de Flexbox peuvent être animées lorsqu'un événement modifie leur valeur.

Voici la liste des propriétés CSS animables :

- `order` ;
- `flex-grow` ;
- `flex-shrink` ;
- `flex-basis` (sauf la valeur `auto`).

Les propriétés suivantes, en revanche, ne peuvent pas être animées :

- `flex-direction` ;
- `flex-wrap` ;
- `justify-content` ;
- `align-items` ;
- `align-self` ;
- `align-content`.

Puisque `flex-grow` et `flex-shrink` peuvent bénéficier des transitions CSS, il est assez simple de réaliser certaines navigations animées dont voici un exemple en figure 13-6.

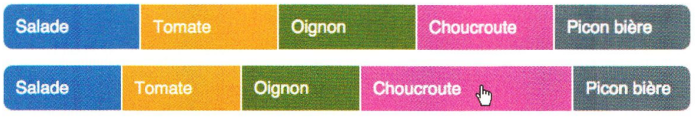


Figure 13–6
Exemple de navigation animée lors du survol

Le code associé pourrait être aussi basique que celui-ci :

```
nav a {
  flex: 1;
}
nav a:hover,
nav a:focus {
  flex: 2;
}
```

Flexbox et accessibilité

L'usage de la propriété `order` implique de grandes responsabilités en termes d'accessibilité. En effet, comment doit se comporter une assistance technique, par exemple une synthèse vocale, lorsque les éléments d'une navigation ont été « mélangés » visuellement via `order` ? Doit-elle se référer à l'ordre du code HTML ou à l'ordre visuel souhaité par le webdesigner ?

C'est le genre de réflexions que mène le groupe de travail sur l'accessibilité au sein du W3C.

Les spécifications dans leur état actuel ont décidé que la propriété `order` ne devait pas être prise en compte par les assistances auditives, ni modifier l'ordre de la navigation au clavier (via la tabulation par exemple).

Plus clairement, `order` ne doit servir qu'à modifier l'ordre dans un but purement décoratif et ne doit pas être employé pour des changements logiques ou de structure.

Je vous invite à régulièrement tester l'accessibilité de vos intégrations en vous souvenant que les assistances pour personnes affectées d'un handicap ne tiendront compte que de l'ordre défini dans le DOM du HTML.

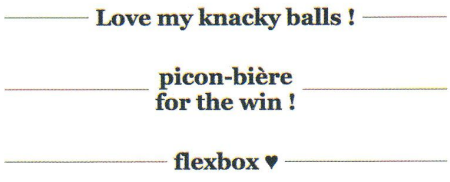
Quelques astuces pratiques

Parce que je ne peux pas m'empêcher de vous dévoiler toutes les petites astuces que je connais, je vous invite à découvrir une sélection de petits modules très simples à réaliser à l'aide de Flexbox : des titres décorés, des points de conduite, un tunnel d'achat et une fenêtre modale centrée.

Décoration de titre

L'idée générale est d'afficher une barre horizontale décorative à gauche et à droite du texte d'un titre de largeur variable, la barre occupant toute la largeur.

Figure 13–7
Liseré de décoration sur des titres



Lien pour tester en ligne :
▸ cdpn.io/GgZQKx

```
h1 {
  display: flex;
}
h1::before,
h1::after {
  display: block; /* correctif IE 10 */
  content: "";
  flex: 1;
  align-self: center;
  border-bottom: 1px solid CurrentColor;
}
```

Points de conduite

Les « points de conduite » désignent les fameux pointillés entre des éléments de tailles différentes.

Figure 13–8
Points de conduite au sein d'un tableau de données

Kiwi	13337€
Banane flambée	42€
Citron	1€
Choucroute	1010€
Picon bière	16516516€

Lien pour tester en ligne :
▸ cdpn.io/yyOXmE

```
tr {
  display: flex;
}
td {
  display: flex;
  flex: 1;
}
td::before {
  display: block; /* correctif IE 10 */
  content: "";
  flex: 1;
  align-self: center;
  border-bottom: 1px dotted #aaa;
  height: 4px;
}
```

Tunnel d'achat

Voyons comment réaliser un tunnel d'achat en cinq étapes à l'aide d'une simple liste numérotée (`<ol>`, `<li>`). L'étape en cours est marquée d'une classe `.is-current` et toutes les étapes suivantes apparaîtront semi-opacifiées.

Figure 13–9
Habillage d'un tunnel d'achat à partir d'une simple liste ordonnée

Lien pour tester en ligne :
▸ cdpn.io/MaJzyB

```
/* ol est le flex-container */
.tunnel {
  counter-reset: progress;
  padding-left: 0;
  display: flex;
}

/* li est flex-item et flex-container */
.tunnel li {
  display: flex;
  align-items: center;
  list-style-type: none;
  counter-increment: progress
}

/* tous flexibles sauf le premier */
.tunnel li+li {
  flex: 1 0 auto;
}

/* masquer les étapes suivantes */
.tunnel .is-current ~ li {
  opacity: .3;
}

/* afficher le compteur */
.tunnel li::after {
  content: counter(progress);
  display: inline-block;
  box-sizing: border-box;
  width: 2em;
  height: 2em;
  line-height: 2em;
  background: hotpink;
  border-radius: 50%;
  text-align: center;
  color: #fff;
}

/* afficher la barre sauf devant le premier */
.tunnel li+li::before {
  content: "";
  flex: 1;
  display: block; /* IE 10 fix */
  border-bottom: 2px solid hotpink;
}
.tunnel .is-current ~ li::before {
  border-bottom-style: dotted;
}
```

Fenêtre modale

Voici une fenêtre modale *pop-in* centrée verticalement au sein de la fenêtre du navigateur bien entendu.

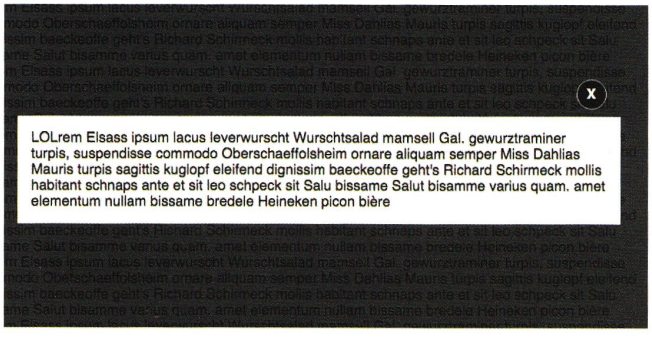


Figure 13–10
Une fenêtre modale centrée grâce à Flexbox

Lien pour tester en ligne :

- cdpn.io/jPZMqv

```
.overlay {
  position: fixed;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  top: 0; bottom: 0; left: 0; right: 0;
  background-color: rgba(0,0,0,0.8);
}
.modal {
  background: #fff;
  width: 80%;
  padding: 1em;
}
```

14

Performances et compatibilité

Où l'on aperçoit nos dragons et trolls déguisés en Internet Explorer pour Halloween. Qu'ils sont taquins !

Performances de Flexbox

La rapidité d'affichage du modèle Flexbox est un sujet de débat passionnant depuis quelques années et plusieurs expérimentations ont tenté de comparer des mises en page à l'aide de flottants, de Flexbox ou d'autres schémas de positionnement.

Les premiers tests, vers 2012, ont conclu qu'un gabarit réalisé via Flexbox était généralement plus lent à s'afficher qu'en employant la bonne vieille méthode des flottants. C'est un comble !

Ceci dit, comme toutes les études sur des cas isolés (ou comme les discours politiques), les résultats de ces tests sont à prendre avec des pincettes et du recul. En effet, les conclusions ont été établies sur une version ancienne de la spécification (`display: box`) et non sur la version finale (`display: flex`). Or, il a été démontré depuis que la version actuelle de Flexbox est environ 2,3 fois plus rapide qu'avant.

▸ http://bit.ly/1LZERh2

Depuis lors, plusieurs études tentent à démontrer que Flexbox demeure lent dans certaines conditions d'affichage, ou bien comparent Flexbox à un autre modèle récent : Grid Layout.

Il est bien entendu très difficile de comparer des modèles d'affichage qui dépendent de spécifications qui évoluent, de navigateurs qui s'améliorent, de la complexité du gabarit testé et de chaque cas particulier au sein de ce gabarit.

Voici ce que l'on peut conclure sur ce point précis de la performance d'affichage de Flexbox dans la construction de gabarits :

- l'ancienne version de Flexbox était bien plus lente que l'actuelle ;
- Flexbox est globalement (un peu) plus rapide que les flottants ;
- Grid Layout est plus rapide que Flexbox (il est fait pour cela) ;
- les éléments flexibles (par exemple `flex: 1`) sollicitent et ralentissent le navigateur jusqu'à ce que leur dimension soit calculée.

En bref, et en attendant que Grid Layout soit reconnu par une majorité de navigateurs, Flexbox demeure la meilleure solution pour la conception de gabarits de mise en page. Évitez toutefois de rendre flexibles tous vos éléments et concentrez-vous uniquement sur ceux qui le nécessitent absolument. Votre navigateur vous en remerciera.

Rappelons enfin que Flexbox est originellement prévu pour le design des composants, des modules, des widgets, et non pour la mise en page globale, même s'il se charge de cette mission avec brio.

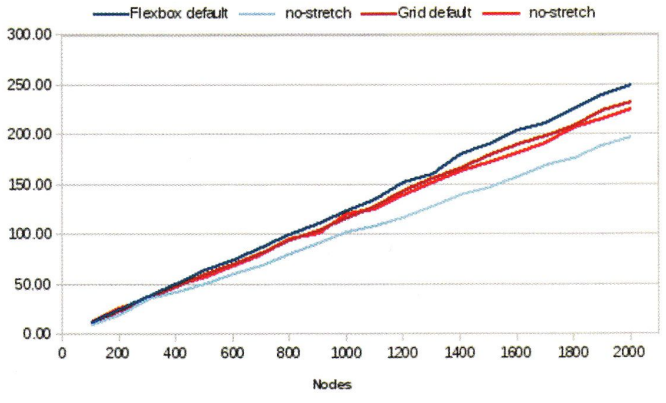


Figure 14–1
Différence d'affichage (ms) entre Flexbox et Grid Layout (source : http://blogs.igalia.com/)

Compatibilité des navigateurs

N'ayons pas peur des mots : le module Flexbox est plutôt très bien reconnu par les navigateurs, même certains glorieux anciens et même sur les mobiles en général, comme en témoigne l'excellente ressource Caniuse.com que vous devriez déjà connaître par cœur.

Le seul retardataire est Internet Explorer, qui ne reconnaît pas cette spécification pour ses anciennes versions (inférieures à IE 10). Cela demeure éventuellement bloquant pour certains projets destinés à des ordinateurs de bureau.

Statistiques d'usage en France et dans le monde

À la fin de l'année 2015, le modèle d'affichage Flexbox était reconnu, statistiquement, par plus de 95 % de vos visiteurs potentiels en France et dans le monde.

Ces chiffres sont bien évidemment extrêmement encourageants, mais je ne vous apprendrai pas que les meilleures statistiques demeurent celles de vos visiteurs au sein de vos projets. Penchez-vous sur vos outils d'analyse de trafic et décortiquez-les : vous aurez sans doute l'heureuse surprise de découvrir que les anciennes versions d'Internet Explorer deviennent chez vous aussi négligeables.

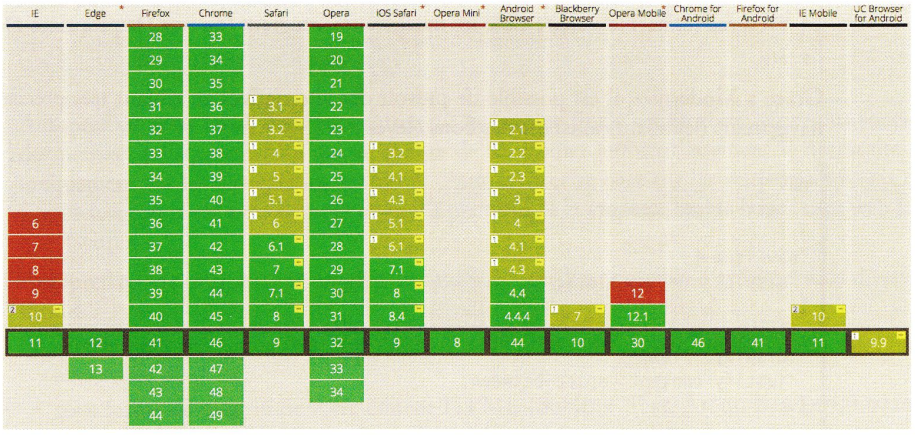


Figure 14–2 Compatibilité de Flexbox Layout (fin 2015)

Les outils et alternatives à Flexbox

Dans le cas où vous ne pourriez vraiment pas vous permettre d'abandonner la prise en charge des navigateurs dinosaures, voici quelques astuces et outils qui pourront grandement vous faciliter la tâche.

Modernizr

Modernizr est une bibliothèque JavaScript très connue qui détecte les fonctionnalités HTML, CSS et JavaScript prises en charge par un navigateur.

▸ https://modernizr.com/

Le principe de cet outil est d'ajouter automatiquement des classes sur l'élément `<html>`. Ces classes servent à cibler les navigateurs selon les fonctionnalités souhaitées.

Par exemple, si Flexbox est pris en charge par le navigateur, Modernizr écrira (entre autres) `<html class="flexbox">` et, dans le cas contraire, `<html class="no-flexbox">`.

La personnalisation de Modernizr est telle qu'il est possible de cibler les navigateurs ayant implémenté une version antérieure de la spécification.

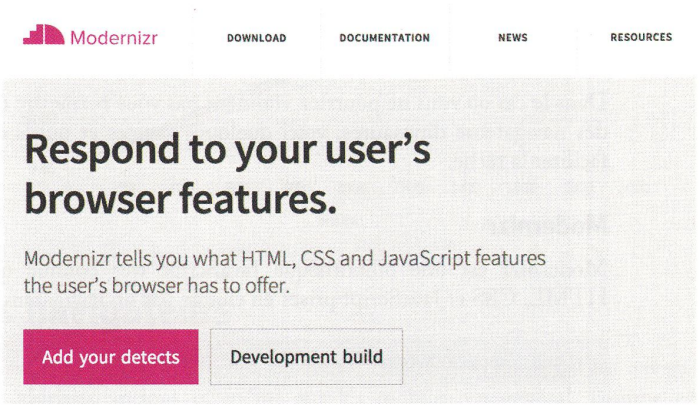
Voici une liste non exhaustive de classes prises en compte par cet outil :

- `flexbox` (et `no-flexbox`) : cible les navigateurs reconnaissant (ou non) la version finalisée de Flexbox ;
- `flexboxlegacy` (et `no-flexboxlegacy`) : cible les navigateurs implémentant (ou non) la première version datant de 2009 ;
- `flexboxtweener` (et `no-flexboxtweener`) : cible les navigateurs prenant en charge (ou non) la version intermédiaire de 2012, soit uniquement IE 10 ;
- `flexwrap` (et `no-flexwrap`) : cible les navigateurs intégrant (ou non) la propriété `flex-wrap`. ;
- etc.

Grâce à Modernizr, il est possible de prévoir des alternatives en ciblant très précisément les navigateurs défaillants ainsi que les bons élèves :

```
nav {
  text-align: center;
}
.nav-item {
  display: inline-block;
}
.flexbox nav {
  display: flex;
  justify-content: space-between;
}
```

Figure 14–3
modernizr.com



Autoprefixer

L'état de l'art de la standardisation de « Flexible Box Layout Module » est pour le moins pittoresque tant il a connu de rebondissements. Au cours des différentes versions de chaque pro-

priété, le nombre de préfixes CSS (`-moz-`, `-webkit-`, `-ms-`, `-o-`) s'est vu multiplié et est rapidement devenu problématique.

Aujourd'hui encore, certains navigateurs ont toujours besoin de ces préfixes, même s'ils se raréfient :

- Internet Explorer 10 (plus besoin de préfixes à partir d'IE 11) ;
- Safari 8 et inférieur ;
- Android 4.3 et inférieur.

Est-ce grave ou bloquant ? Certainement pas, à condition de prendre quelques précautions à l'aide de l'outil génial qu'est Autoprefixer.

▸ https://github.com/postcss/autoprefixer

Autoprefixer vous laisse vous concentrer sur la syntaxe finale de Flexbox, sans vous soucier des préfixes puisque c'est lui qui se charge de les ajouter automatiquement si nécessaire et uniquement sur les navigateurs concernés. Autoprefixer se base sur les statistiques issues du site Caniuse.com et est actuellement conseillé par Google et employé sur de larges sites web tels que Twitter.

Autoprefixer est initialement un plug-in JavaScript qu'il est possible d'exécuter sous diverses formes :

- directement en ligne via des bacs à sable tels que http://pleeease.io/play/ ou http://autoprefixer.github.io/ ;
- sous forme de plug-in de votre éditeur HTML (Sublime Text, Atom, Brackets, Dreamweaver, Visual Studio, Notepad++, etc.) ;
- via une tâche Grunt, Gulp, Webpack, Brunch, Jekyll ;
- combiné à des préprocesseurs tels que LESS, Sass ou Stylus ;
- au sein de compilateurs graphiques tels que Prepros.io ou CodeKit.

Grâce aux nombreuses formes que revêt Autoprefixer, vous trouverez sans aucun problème une méthode pour le mettre en œuvre dans vos projets web et vous oublierez très vite les inconvénients des préfixes navigateurs.

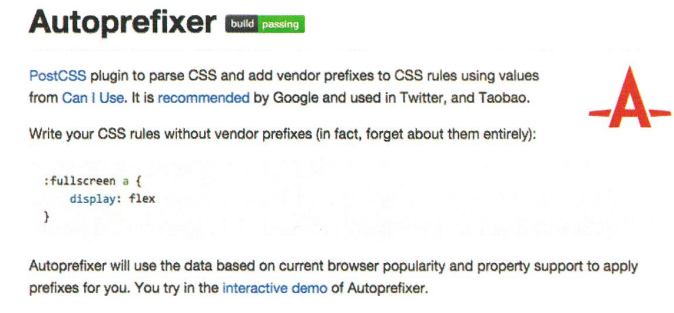


Figure 14–4
La page Autoprefixer sur https://github.com/postcss/autoprefixer

Les préprocesseurs

Les préprocesseurs, que j'ai déjà évoqués dans un chapitre précédent concernant les grilles de mise en page, sont des métalangages destinés à automatiser un certain nombre de tâches répétitives, notamment. Or, ajouter des préfixes CSS à la main *est* une tâche répétitive.

Les préprocesseurs tels que LESS, Sass ou encore Stylus embarquent des fonctions personnalisables appelées *mixins* permettant ce genre de mission.

En langage Sass, voici à quoi une telle *mixin* peut ressembler :

```
@mixin vendor-prefix($name, $argument) {
  -webkit-#{$name}: #{$argument};
  -ms-#{$name}: #{$argument};
  -moz-#{$name}: #{$argument};
  -o-#{$name}: #{$argument};
  #{$name}: #{$argument};
}
p {
  @include vendor-prefix(hyphens, auto)
}
```

Même si ce genre de fonction peut vous sembler attrayant, je vous invite à y réfléchir à deux fois. En effet, le CSS généré à chaque opération comportera systématiquement tous les préfixes existants (`-moz-`, `-webkit-`, `-ms-`, `-o-`), c'est-à-dire cinq lignes de déclarations… même lorsque la moitié, voire la totalité, des préfixes n'est plus nécessaire.

En recourant aux *mixins* de préprocesseurs, votre feuille de styles va considérablement s'allonger, souvent pour des déclarations devenues obsolètes depuis fort longtemps.

L'outil Autoprefixer évoqué dans la partie précédente est, selon moi, une bien meilleure ressource pour ce type d'opération.

Les polyfills

Les polyfills sont tout simplement des alternatives JavaScript consacrées à la simulation de fonctionnalités pour les navigateurs déficients ou anciens.

Flexie est un polyfill de Flexbox que vous trouverez à l'adresse https://github.com/doctyper/flexie.

En pratique, il rend Flexbox Layout compatible avec des antiquités telles que Internet Explorer 6 !

Toutefois, Flexie est bâti sur la première spécification de Flexbox, datant de 2009, et vétuste depuis plusieurs années déjà. Pour mettre en action Flexie, il vous serait nécessaire de rédiger toutes les propriétés de Flexbox dans un langage du temps passé.

Aujourd'hui, le développeur de Flexie déconseille lui-même d'employer ce polyfill. Il ajoute qu'il travaille sur une version remaniée de son outil, nommée Reflexie. À ce jour toutefois, la dernière modification apportée à ce projet date d'il y a plus de trois années.

En clair, si j'étais vous, j'éviterais de prendre le risque de mettre en production ce genre de jouet, plus ou moins laissé à l'abandon, qui de surcroît pose de réels problèmes de performances sur les anciens navigateurs.

Bogues connus et solutions

À l'instar de toutes les spécifications jeunes, Flexbox Layout n'est pas exempt de défaillances sur certains navigateurs, notamment ceux ayant implémenté ce module avant qu'il ne soit finalisé.

Pour vous faire gagner du temps sur certains comportements inexplicables de votre intégration CSS, j'ai rassemblé au sein de ce chapitre les principales aberrations Flexbox de nos chers navigateurs.

Pour commencer, notez et conservez l'adresse du projet Flexbugs de Philippe Walton : https://github.com/philipwalton/flexbugs.

Sur cette page collaborative Github sont collectés la plupart des bogues connus de Flexbox. Il est certain que vous allez y trouver des informations profitables.

Figure 14–5
Flexbugs

README.md

Flexbugs

This repository is a community-curated list of flexbox issues and cross-browser workarounds for them. The goal is that if you're building a website using flexbox and something isn't working as you'd expect, you can find the solution here.

As the spec continues to evolve and vendors nail down their implementations, this repo will be updated with newly discovered issues and remove old issues as they're fixed or become obsolete. If you discover a bug that's not listed here, please report it so everyone else can benefit.

The bugs and their workarounds

1. Minimum content sizing of flex items not honored
2. Column flex items set to `align-items:center` overflow their container
3. `min-height` on a flex container won't apply to its flex items
4. `flex` shorthand declarations with unitless `flex-basis` values are ignored
5. Column flex items don't always preserve intrinsic aspect ratios
6. The default `flex` value has changed
7. `flex-basis` doesn't account for `box-sizing:border-box`
8. `flex-basis` doesn't support `calc()`

La valeur par défaut de flex a changé (IE 10)

Internet Explorer 10 est le seul navigateur à avoir implémenté la version intermédiaire (« tweener ») de Flexbox datant de 2012.

À l'époque de cette spécification, la valeur par défaut de `flex` était `none`, c'est-à-dire l'équivalent de `0 0 auto` (la valeur actuelle est `initial`, soit l'équivalent de `0 1 auto`). La différence réside dans la valeur initiale de la propriété `flex-shrink` et signifie que, par défaut sur IE 10, un flex-item n'est pas contractable.

La meilleure solution pour éviter ce genre de mauvaise surprise est de toujours préciser explicitement le facteur de rétrécissement :

- soit en indiquant cette valeur sur chaque flex-item via `flex-shrink: 1` ;
- soit en optant pour la propriété raccourcie `flex` en précisant toutes les valeurs, par exemple `flex: 0 1 auto`.

Bogue de min-height non appliqué (IE 10-IE 11)

Sur un flex-container, la propriété `min-height` n'est pas correctement prise en compte par IE 10 et IE 11 (c'est corrigé sur Edge). Cependant la propriété `height` fonctionne bien.

Voici un code d'exemple déficient sur ces versions de navigateurs :

```
.flex-container {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}
```

Pour résoudre ce problème, l'une des solutions est d'appliquer un *patch* sur le parent de l'élément problématique et de le rendre lui-même flex-container via un `display: flex`.

```
/* correctif min-height pour IE 10-11 */
.parent > .flex-container {
  display: flex;
  flex-direction: column;
}
.flex-container {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}
```

Propriété flex non reconnue sur des éléments inline (IE 10-IE 11)

Sur IE 10 et IE 11, les éléments de type *inline* (`<span>`, `<a>`, `<label>`, etc.), de même que les pseudo-éléments générés en CSS (`::before` et `::after`), ne sont pas correctement traités et affichés tels des flex-item. Ce bogue est corrigé sur Edge.

Par exemple, le code suivant n'aura aucun effet sur ces navigateurs :

```
.container {
  display: flex;
}
span {
  flex: 1;
}
```

L'astuce consiste à modifier le type de rendu de ces éléments censés être des flex-item et de leur appliquer par exemple un `display: block` :

```
.container {
  display: flex;
}
span {
  display: block; /* correctif IE pour éléments inline */
  flex: 1 ;
}
```

Pas de box-sizing sur flex-basis (IE 10-IE 11)

La propriété `box-sizing` ne s'applique pas sur une largeur d'élément définie via `flex-basis` sur les navigateurs IE 10 et IE 11 (corrigé sur Edge).

Pour comprendre, prenez le bout de code suivant :

```
.flex-item {
  box-sizing: border-box;
  flex-basis: 100px;
  padding: 10px;
}
```

Au final, notre flex-item dispose d'une largeur totale de 120 pixels (la largeur de `flex-basis` à laquelle on ajoute les `padding`), ce qui ne nous arrange guère en général.

La solution est plutôt évidente : tant que ce bogue existe et si vous usez de `box-sizing`, il demeure plus pertinent de fixer les dimensions des flex-item à l'aide des propriétés `width` ou `height` :

```
.flex-item {
  box-sizing: border-box;
  width: 100px;
  padding: 10px;
}
```

Ainsi, quel que soit le navigateur, notre élément mesure 100 pixels de large comme le lui demande la propriété `box-sizing`.

Pas de calc() sur flex (IE 10-IE 11)

Sur IE 10 et IE 11, la valeur-fonction `calc()` est incompatible avec la propriété raccourcie `flex`.

En d'autres termes, le raccourci `flex: 0 0 calc(100% / 3)` est inopérant... En revanche, la version séparée `flex-basis: calc(100% / 3)` fonctionne parfaitement !

Vous aurez facilement deviné quelle syntaxe vous serez amené à employer s'il vous faut cumuler les deux fonctionnalités.

max-width: 100% non appliqué sur les images ?

En plein test de mise en page Responsive, et encore tout excité de pouvoir enfin appliquer Flexbox en production, vous serez surpris de constater un phénomène étrange lorsque vous tenterez d'utiliser la célèbre technique suivante pour rendre vos images fluides à l'intérieur de leur parent flex-item :

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

Non seulement, l'image ne s'adapte pas à la taille de son parent comme vous l'aviez prévu, mais pire : c'est son parent qui s'adapte aux dimensions initiales de l'image !

Il ne s'agit (malheureusement) pas d'un bogue mais de l'observation stricte des récentes spécifications de Flexbox. Pour rappel, les flex-item ont dorénavant par défaut une valeur de `min-width` (et `min-height`) définie à `auto`, c'est-à-dire la taille de leur contenu. Les spécifications sont claires sur ce point.

> http://www.w3.org/TR/css-flexbox-1/#min-size-auto

Certains éléments dits « remplacés », c'est-à-dire les éléments `<img>`, `<video>`, `<picture>`, `<canvas>`, `<textarea>`, `<input>`, etc. possèdent des dimensions intrinsèques. Leur parent flex-item, en vertu de sa nouvelle valeur de `min-width`, calcule sa taille en fonction de cette dimension par défaut. C'est pourquoi `max-width: 100%` n'aura aucun effet... puisque c'est l'image elle-même qui détermine la taille de son parent.

Il existe trois méthodes simples pour éviter ce piège et rendre nos images fluides :

- **`min-width: 0`** sur le parent flex-item de l'image - En effet, muni d'une valeur explicite de `min-width`, il n'adopte plus la valeur par défaut `auto`, qui était problématique ;
- **`flex-direction: column`** sur le flex-container - En basculant dans un axe vertical, c'est la hauteur minimale (`min-height`) qui est alors calculée en fonction du contenu et non plus la largeur ;
- **`overflow: hidden`** (ou `auto` ou `scroll`) sur le flex-item - Cela crée un Block Formatting Context (BFC) et confère un superpouvoir à l'élément, notamment celui de redéfinir un nouveau « niveau de flux » et de ne plus subir la largeur de ses enfants.

15

Flexbox contre Grid Layout ?

Où la princesse marâtre est finalement vaincue elle aussi et où le barde chante une toute nouvelle geste pleine d'aventures extraordinaires.

Les limites de Flexbox ?

Nous voici parvenus à la fin de ce livre dédié à Flexbox. Au vu de toutes les fonctionnalités admirables que l'on a pu manipuler avec bonheur tout au long de notre aventure, l'on peut à présent se poser une question bien à-propos : y a-t-il des limites à l'usage de Flexbox ?

Eh bien figurez-vous que oui.

Pour commencer, Flexbox a été créé dans l'optique de résoudre la plupart des situations « pénibles » dans le quotidien d'un intégrateur (alignements verticaux, hauteurs identiques, flexibilité, etc.), mais **n'a pas pour vocation de remplacer tous les autres types de positionnements existants.**

Flexbox remplit parfaitement son rôle dès lors qu'il s'agit de styler des composants ou widgets, répartis au sein de conteneurs d'une seule dimension (horizontale ou verticale). Même s'il s'en sort avec les retours à la ligne grâce à des astuces comme `flex-wrap`, on lui trouvera rapidement des limites, que l'on a pu constater dans l'un des chapitres dédiés aux modèles de design.

En bref, Flexbox est prévu pour des modules dans une dimension, pas vraiment pour des gabarits complets ou des grilles à plusieurs dimensions.

Figure 15–1
Un gabarit qui semble simple, mais loin d'être évident à réaliser via Flexbox sans conteneur supplémentaire

Et c'est là qu'intervient un modèle de positionnement encore plus récent, **CSS Grid Layout**.

Grid Layout

Le module de positionnement Grid Layout est une spécification du W3C à l'état de brouillon *(Working Draft)* dont les premiers jets datent de 2004.

Sa documentation officielle est actuellement maintenue par trois contributeurs, dont une personne de Microsoft et une personne de Google.

Le positionnement par grille

Le concept général de Grid Layout (ou « positionnement par grille ») est de diviser virtuellement l'espace en zones majeures dans une page ou une application. Concrètement et schématiquement, il s'agira de découper en lignes et en colonnes comme nous le ferions pour un tableau de mise en page.

On y trouve d'ailleurs de nombreuses références d'affichage « tabulaire » avec lignes et colonnes, `rowspan` et `colspan`. En cela, ce schéma de positionnement est très similaire aux tableaux HTML ou aux rendus de type `display: table`, `display: table-cell` et autre `display: table-row`.

La différence la plus flagrante est que la grille consiste en une construction de l'esprit et ne nécessite aucun élément HTML ni balisage pour être élaborée. Aucune charpente physique telle que `<table>`, `<tbody>`, `<tr>`, `<td>` ou `<th>` n'est nécessaire, ce qui en facilite l'adaptation à différentes tailles d'écrans et de périphériques : inutile d'intervenir sur l'ordre, la nature ou la « sémantique » des éléments HTML, il suffit de modifier la grille initiale en CSS pour que tous les éléments s'y adaptent.

Les propriétés usuelles de Grid Layout

De nombreuses propriétés sont décrites dans les spécifications du module de grilles. Parmi les plus courantes, nous trouvons :

- `grid`, `inline-grid` : déclaration d'un contexte de grille (création d'un `grid-container`) ;
- `grid-template-areas` : déclaration de cellules nommées (optionnel) ;
- `grid-template-rows`, `grid-template-columns` : déclaration des dimensions de lignes et colonnes ;
- `grid-row`, `grid-column` : placement d'un élément `grid-item` dans une ligne ou une colonne ;
- `grid-gap`, `grid-column-gap`, `grid-row-gap` : espaces entre colonnes ou entre rangées (gouttières) ;
- `align-items`, `justify-items` : alignement horizontal ou vertical ;
- `align-self`, `justify-self` : alignement horizontal ou vertical d'éléments distincts.

Mise en œuvre

On crée un « contexte de grille » tout simplement en appliquant la déclaration `display: grid` (ou `inline-grid`) à un élément conteneur qui constituera la trame générale ; il deviendra un `grid-container` et ses enfants directs des `grid-item`. Cette trame sera définie par un schéma virtuel formé de lignes et de colonnes.

Tous les enfants directs de ce conteneur général seront automatiquement affectés par ce contexte particulier et pourront se placer au sein de la trame globale.

Exemple 1 (affichage de deux blocs sur une ligne)

Partie HTML

```
<body>
  <nav>...</nav>
  <section>...</section>
</body>
```

Partie CSS

```
body {
  display: grid;
  grid-template-columns: 200px 400px;
}
nav {
  grid-column: 1; /* placement en colonne 1 */
}
section {
  grid-column: 2; /* placement en colonne 2 */
}
```

De la sorte, nous définissons deux colonnes de respectivement 200 px et 400 px, dans lesquelles viennent se loger les éléments `nav` et `section`.

Figure 15–2
Mise en œuvre de Grid Layout

Exemple 2 (grille de 4 emplacements)

Partie HTML

```
<body>
  <nav>...</nav>
  <section>...</section>
  <article>...</article>
  <aside>...</aside>
</body>
```

Partie CSS

```
body {
  display: grid;
  grid-template-columns: 250px 400px;
  grid-template-rows: 100px 300px;
}
nav {
  grid-column: 1; grid-row: 1;
}
section {
  grid-column: 2; grid-row: 1;
}
article {
  grid-column: 1; grid-row: 2;
}
aside {
  grid-column: 2; grid-row: 2;
}
```

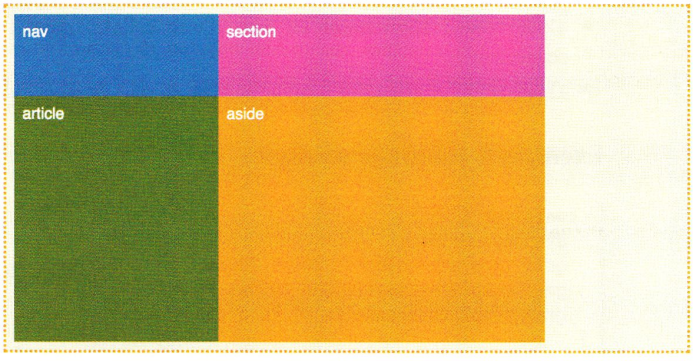


Figure 15–3
Des rangées et des colonnes au sein de Grid Layout

Variante : la syntaxe de templates

Grid Layout autorise la visualisation sous forme de zones, en nommant explicitement les emplacements de la grille à l'aide de chaînes de caractères ou de simples lettres.

Exemple 3 (template)

```
#inGrid {
  display: grid;
  grid-template-areas: "h h"
                       "n c"
                       "f f";
}
nav {
  /* placement de <nav> dans l'emplacement "n" */
  grid-area: n;
}
```

Figure 15–4
Positionnement à l'aide de zones nommées

Centrer les éléments

Par défaut, les éléments `grid-item` s'étirent pour occuper tout l'espace de leur cellule. Cependant, Grid Layout permet d'aligner les contenus verticalement ou horizontalement à l'aide des propriétés suivantes :

- `justify-items` : alignement au sein d'une cellule (dans l'axe principal). S'applique au conteneur ;

- `justify-self` : alignement d'un `grid-item` au sein de sa cellule. S'applique au `grid-item` ;
- `align-items` : alignement au sein d'une cellule (dans l'axe secondaire). S'applique au conteneur ;
- `align-self` : alignement d'un `grid-item` au sein de sa cellule. S'applique au `grid-item`.

Exemple 4 (centrage multiple)

```
.container {
  display: grid;
  grid-template-columns: 200px 200px;
  grid-template-rows: 200px 200px;
  grid-template-areas: "a b" "c d";
}
header {
  grid-area: a;
  justify-self: center;
}
nav {
  grid-area: b;
  align-self: center;
}
article {
  grid-area: c;
  justify-self: center;
  align-self: center;
}
footer {
  grid-area: d;
}
```

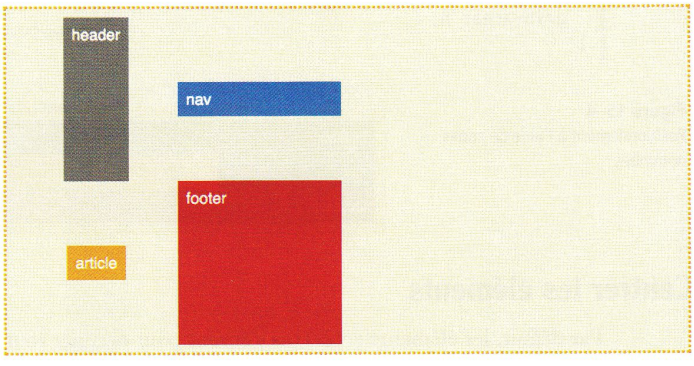


Figure 15–5
Différents types de centrage illustrés

Occuper plusieurs emplacements

Tels les célèbres attributs `rowspan` et `colspan` dédiés aux tableaux HTML, le module Grid Layout introduit la possibilité pour un élément de s'étaler sur plusieurs emplacements, à la fois horizontalement et verticalement.

Le descripteur CSS `span` est chargé d'organiser cette distribution sur plusieurs lignes ou colonnes. Par défaut, un `grid-item` dispose d'un `span` de valeur `1` (euh, rien à voir avec l'élément HTML `<span>`, n'est-ce pas ?).

Exemple 5 (column span)

```
.container {
  display: grid;
  grid-template-columns: 10em 1fr;
  grid-template-rows: min-content 1fr min-content;
  height: 300px;
}
header {
  grid-column: 1  / span 2;
  grid-row: 1;
}
nav {
  grid-column: 1;
  grid-row: 2;
}
article {
  grid-column: 2;
  grid-row: 2;
}
footer {
  grid-column: 1  / span 2;
  grid-row: 3;
}
```

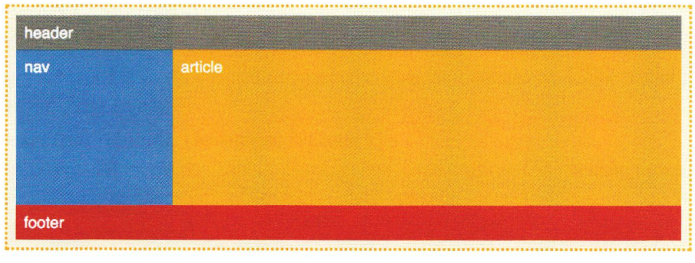


Figure 15–6
L'en-tête et le pied de page occupent chacun deux emplacements.

Et bien d'autres choses encore !

La spécification de Grid Layout ne se limite pas aux quelques fonctionnalités rapidement exposées dans ce chapitre, vous vous en doutez bien.

Parmi les autres avantages prévus dans ce module de positionnement, retenons plus particulièrement :

- de nouvelles unités telles que `min-content`, `max-content`, `fr` (pour désigner une fraction de l'espace restant) ;
- des motifs de répétition, à l'aide de la notation `repeat()` ;
- une répartition automatique des éléments, horizontale ou verticale ;
- un pré-disposition au Responsive : seule la définition de grille de départ nécessite d'être modifiée, il est inutile d'intervenir sur la structure HTML ou le positionnement des éléments.

En bref, le module Grid Layout constitue une méthode de positionnement robuste, très agréable et parfaitement adaptée à la construction de gabarits d'affichage.

Au moment où ce livre est rédigé, Internet Explorer reconnaît le module Grid Layout avec des préfixes et Chrome, Firefox, Opera et Safari sont en train de le prendre en charge. Ce n'est qu'une question de mois avant qu'il soit disponible sur l'ensemble du parc de navigateurs actuel.

Patientons encore un peu le temps que son implémentation se soit démocratisée et nous aurons enfin deux superbes outils de positionnement en CSS bien propres sur eux : Flexbox pour les modules et Grid Layout pour les mises en page. Que rêver de mieux ?

Pour en savoir plus sur la spécification Grid Layout (qui mériterait à elle seule un ouvrage dédié), je vous invite à parcourir au moins ces deux ressources sur le sujet :

- gridbyexample.com : un site dédié à Grid Layout, alimenté par Rachel Andrew ;
- kiwi.gg/gridlayout : l'article d'Alsacréations sur le thème de Grid Layout.

16

Ressources utiles

Où l'on atteint le générique de fin et où le casting des figurants défile sous nos yeux encore embrumés.

Références

- Spécifications officielles du W3C, *CSS Flexible Box Layout Module Level 1* :
  http://www.w3.org/TR/css-flexbox-1/ (en anglais)
- *Le petit Flexbox illustré*, une excellente ressource synthétique et en français, pour mieux comprendre Flexbox :
  http://www.vincent-valentin.name/articles/le-petit-flexbox-illustre
- *A complete guide to flexbox* (CSS-tricks) :
  https://css-tricks.com/snippets/css/a-guide-to-flexbox/ (en anglais)
- *Flexbox, un guide complet* (La Cascade), traduction en français de l'article de CSS-tricks :
  https://la-cascade.io/flexbox-guide-complet/
- *Using CSS flexible boxes* (par Mozilla Developer Network) :
  https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Using_CSS_flexible_boxes (en anglais)
- *Flexbox Layout Module* (Alsacréations) :
  http://www.alsacreations.com/tuto/lire/1493-css3-flexbox-layout-module.html (en français)

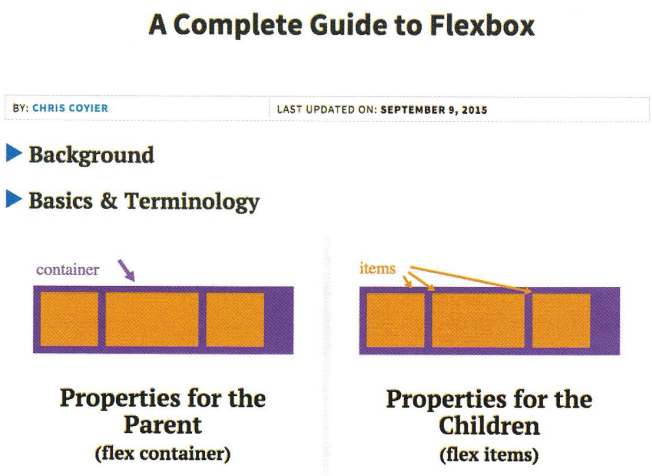


Figure 16–1
« A complete guide to flexbox », CSS-tricks

Tutoriels et articles

- *Flexbox Froggies*, un jeu très sympa pour apprendre Flexbox : flexboxfroggy.com (en anglais)
- *Jack in the Flexbox*, recueil de cas concrets résolus grâce à Flexbox : http://jackintheflexbox.tumblr.com (en français)
- *CSS Flexbox*, une compilation de ressources en anglais : http://cssflexbox.com/
- *Flexy boxes*, un bac à sable pour tester toutes les propriétés Flexbox en direct, en anglais : http://the-echoplex.net/flexyboxes/
- *Flexplorer*, un autre genre de bac à sable en ligne et en anglais : http://bennettfeely.com/flexplorer/
- *Flex for fantasy*, un rassemblement de ressources en français : http://4design.xyz/module-css3-flexbox
- *Building Flexbox website*, un tutoriel pas à pas en anglais : http://www.flexbox.website/

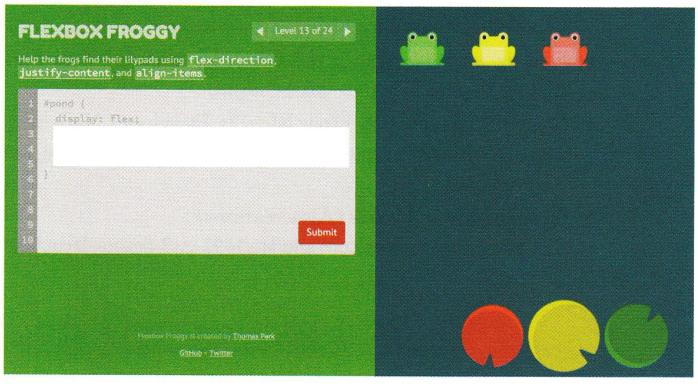


Figure 16–2
Le jeu « Flexbox Froggies »

Vidéos

- *What The Flexbox !* une vingtaine de tutoriels vidéo pas à pas gratuits : http://www.flexbox.io (en anglais)
- *CSS Flexbox Essentials* : https://www.youtube.com/watch?v=G7EIAgfkhmg (en anglais)

Conférences

- *Advanced Layouts with Flexbox* (en anglais) : http://www.mediacurrent.com/blog/advanced-layouts-flexbox
- *Leveling Up With Flexbox* (en anglais) : http://fr.slideshare.net/zomigi/leveling-up-with-flexbox-smartwebconf140923
- *Enhancing Responsiveness with Flexbox* (en anglais) : http://fr.slideshare.net/zomigi/enhancing-responsiveness-with-flexbox-css-conf-eu-2015

Livres

- *Flexbox website*, format e-book (en anglais) : https://leanpub.com/flexbox-website
- *Unraveling Flexbox*, format e-book (en anglais) : unravelingflexbox.com
- Le livret *Flexbox* du Train de 13h37 (en français), à paraître

Autres

- *Flexbox App*, une application Rails permettant de concevoir visuellement des interfaces via Flexbox (en anglais) : http://flex.rocks
- *CSS layout*, une newsletter entièrement dédiée au positionnement CSS (en anglais) : http://csslayout.news/

ANNEXE

Mémo des propriétés

Où l'on se dit qu'elle était bien jolie cette histoire et qu'on a hâte de mettre tout ça en pratique, non ?

Tableau A-1 L'ensemble des propriétés du module Flexbox

Propriété	Valeurs possibles	Valeur initiale	S'applique à	Animable
`display`	`flex` | `inline-flex`	selon l'élément	flex-container	non
`flex-direction`	`row` | `row-reverse` | `column` | `column-reverse`	`row`	flex-container	non
`flex-wrap`	`nowrap` | `wrap` | `wrap-reverse`	`nowrap`	flex-container	non
`flex-flow`	cf. `flex-direction` et `flex-wrap`	`row nowrap`	flex-container	voir propriétés individuelles
`justify-content`	`flex-start` | `flex-end` | `center` | `space-between` | `space-around`	`flex-start`	flex-container	non
`align-items`	`flex-start` | `flex-end` | `center` | `baseline` | `stretch`	`stretch`	flex-container	non
`align-content`	`flex-start` | `flex-end` | `center` | `space-between` | `space-around` | `stretch`	`stretch`	flex-container multilignes	non
`flex-grow`	Nombre positif	`0`	flex-item	oui
`flex-shrink`	Nombre positif	`1`	flex-item	oui
`flex-basis`	Largeur	`auto`	flex-item	oui

Tableau A-1 L'ensemble des propriétés du module Flexbox *(suite)*

Propriété	Valeurs possibles	Valeur initiale	S'applique à	Animable
`flex`	cf. `flex-grow`, `flex-shrink` et `flex-basis`	`0 1 auto`	flex-item	voir propriétés individuelles
`order`	Nombre entier	`0`	flex-item et élément postionné en absolute	oui
`align-self`	`auto` | `flex-start` | `flex-end` | `center` | `baseline` | `stretch`	`auto`	flex-item	non

Index

.clearfix 101
::after 86, 100
::before 100
::first-letter 99
::first-line 99
:not() 67

A

accessibilité 103
align-content 14
alignement 7
align-items 13
align-self 16
animations 103
ARIA 34
assistance technique 34
Autoprefixer 112

B

block 7, 15
Block Formatting Context 100, 118
bogue 115
Bootstrap 87
box-sizing 117

C

calc() 83, 118
Caniuse.com 4, 110, 113
centrer verticalement 27, 29
clear 99
clearfix 28
column 99
compatibilité 110
currentColor 38

D

display 8
distribution 7
DOM 24, 100

E

espace restant 27, 73

F

flex 19, 71
Flex Formatting Context 100
flex-basis 18
flex-direction 9
flex-flow 11
flex-grow 17, 74
Flexible Layout Module 1
flex-shrink 17, 77
flex-wrap 10
float 7
fluidité 7
formulaire 63
Foundation 87
fusion de marges 100

G

galerie d'images 49
gouttière 79, 83, 89, 90
Grid Layout 110, 119
grille 84, 87, 119

I

IFTTT IX
inline 7, 15

J

justify-content 12

K

KNACSS 88, 97

L

ligatures 33
ligne de flottaison 25

M
Media Queries 24, 34, 85
min-width 61, 81, 101, 118
mixin 83
modale 107
modèle de design 79
Modernizr 111

N
navigation 33

O
object-fit 55
object-position 56
order 15, 23, 37, 46, 68, 93, 104

P
performance 110
polyfill 114
préprocesseur 83, 92, 113, 114
pseudo-élément 86, 100
pull 94
push 94

R
ratio 52
répartition 80
Responsive Webdesign VII, 24, 46, 68, 95
RSS IX

S
Saint-Graal 41
space-between 84

T
tweener syntax 2
Twitter X

V
vertical-align 99
vh 44, 53
viewport 44

Z
z-index 101

N° d'éditeur : 9640
Dépôt légal : février 2016
Imprimé en Italie

CSS 3 FLEXBOX

Flexbox est en passe de révolutionner de manière profonde et pérenne notre façon de concevoir des designs et des composants en CSS. Pourquoi y consacrer un ouvrage ? Car, bien utilisé, Flexbox permet de faciliter la tâche de bon nombre de développeurs web.

Un ouvrage de référence pour les webdesigners et intégrateurs

« Flexible Box Layout Module », mieux connue sous le nom de « Flexbox », est une spécification CSS 3 du W3C qui définit un nouveau modèle de boîte et de positionnement jusqu'alors inédit. Il intègre une gestion naturelle de la fluidité des éléments et du Responsive, et rend caduc l'usage de grilles d'affichage complexes, voire de frameworks – véritables usines à gaz où l'on n'exploite qu'à peine 10 % de l'outil. Oubliez donc tout ce que vous aviez appris sur CSS et, comme moi, tombez amoureux de Flexbox !

Outre la simplification des schémas de positionnement, Flexbox apporte des solutions parfaites à une problématique bien ancrée dans notre époque : le Responsive Webdesign. Le design d'éléments flexibles, la réorganisation des blocs ainsi que la faculté à basculer très aisément d'un mode d'affichage horizontal vers un mode vertical en font un allié formidable dans nos projets d'adaptation aux tablettes et smartphones.

Une spécification CSS 3 novatrice

Flexbox a été pensé et optimisé pour faire table rase de toutes les techniques bancales historiques de positionnement et des contournements de propriétés qui pullulent dans nos projets web. Les « anciennes » méthodes encore (mal) utilisées de nos jours pour aligner ou placer des éléments ne sont souvent rien d'autre que du bricolage empirique : « tiens, à quoi peut bien servir ce position : relative ? », « pas grave, je mets une classe .clearfix partout ! », « oh ! mais pourquoi ça ne veut pas rentrer ? », etc.

Flexbox est conçu pour mettre de l'ordre dans tout ce maelström de bidouilles et revenir enfin aux bases d'un positionnement propre et adapté aux besoins actuels. Cet ouvrage vous permettra de découvrir et de tirer parti de toutes les nouveautés apportées par cette spécification CSS 3, notamment à travers six travaux pratiques...

À qui s'adresse cet ouvrage ?

- Aux webdesigners et intégrateurs avancés ou experts qui souhaitent appréhender et maîtriser en production ce nouveau modèle de positionnement excitant.
- Aux développeurs et chefs de projet web soucieux de découvrir les rouages d'une technologie qui facilitera assurément la vie de leurs projets.
- À quiconque est impatient de laisser tomber toutes les bidouilles ancestrales de CSS et d'ouvrir ses horizons à des modèles bien plus intuitifs.

Webdesigner et gérant d'une agence web strasbourgeoise, **Raphaël Goetter** partage ses connaissances à travers son site Alsacreations.com et s'intéresse de près aux domaines des normes du Web et de l'accessibilité. Il est également consultant et formateur en langages HTML, CSS 3 et Responsive Webdesign auprès de groupes nationaux et internationaux.

Sommaire

Une brève histoire de Flexbox • Les bases du design avec Flexbox • TP : réordonner des éléments • Trois astuces utiles • TP : une navigation Responsive • TP : un gabarit simple • TP : une galerie d'images • Les principes fondamentaux • TP : un formulaire fluide • La propriété flex en détail • Modèles de design • TP : construction de grilles • Encore plus loin avec Flexbox • Performances et compatibilité • Flexbox contre Grid Layout ? • Ressources utiles • Annexe - Mémo des propriétés

19,90 €
www.editions-eyrolles.com
Groupe Eyrolles | Diffusion Geodif

Conception de couverture : © Jérémie Barlog

Code éditeur : G14363
ISBN : 978-2-212-14363-8