A BOOK APART
Les livres de ceux qui font le web

No. 13

Scott Jehl

DESIGN WEB RESPONSIVE ET RESPONSABLE

PRÉFACE DE Ethan Marcotte

EYROLLES

Le responsive design a considérablement amélioré le design web de ces dernières années, mais compte tenu de l'évolution et de la diversité du Web mondial, il faut désormais aller plus loin et concevoir des sites responsables en veillant à leur utilisabilité, leur accessibilité, leur durabilité et leur performance.

Scott Jehl nous montre ici comment porter un regard critique sur nos créations en développant pour de nouveaux contextes et de nouvelles fonctionnalités, pour des réseaux rapides comme des réseaux plus lents et pour un public véritablement global. Il décrit comment proposer le bon contenu à différentes plateformes et comment optimiser les performances d'un site. Lisez cet ouvrage et concevez des sites et des applications utilisables pour les années à venir !

A BOOK APART

Les livres de ceux qui font le web

www.editions-eyrolles.com


1
HTML5 POUR LES WEB DESIGNERS


2
CSS3 POUR LES WEB DESIGNERS


3
STRATÉGIE DE CONTENU WEB


4
RESPONSIVE WEB DESIGN


5
DESIGN ÉMOTIONNEL


6
MOBILE FIRST


7
MÉTIER WEB DESIGNER


8
STRATÉGIE DE CONTENU MOBILE


9
LA PHASE DE RECHERCHE EN WEB DESIGN


10
SASS POUR LES WEB DESIGNERS


11
TYPOGRAPHIE WEB


12
WEB DESIGNER CHERCHE CLIENT IDÉAL

Scott Jehl

DESIGN WEB RESPONSIVE ET RESPONSABLE

EYROLLES

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

Traduction autorisée de l'ouvrage en langue anglaise intitulé *Responsible responsive design* de Scott Jehl (ISBN : 978-1-937557-1-7-1), publié par A Book Apart LLC

Adapté de l'anglais par Charles Robert

Dans la même collection

HTML5 pour les web designers - n°1, Jeremy Keith, 2010, 96 pages.
CSS3 pour les web designers - n°2, Dan Cederholm, 2011, 128 pages.
Stratégie de contenu web - n°3, Erin Kissane, 2011, 96 pages.
Responsive web design - n°4, Ethan Marcotte, 2011, 160 pages.
Design émotionnel - n°5, Aarron Walter, 2012, 118 pages.
Mobile first - n°6, Luke Wroblewski, 2012, 144 pages.
Métier web designer - n°7, Mike Monteiro, 2012, 156 pages.
Stratégie de contenu mobile - n°8, Karen McGrane, 2013, 164 pages.
La phase de recherche en web design - n°9, Erika Hall, 2015, 176 pages.
Sass pour les web designers - n°10, Dan Cederholm, 2015, 96 pages.
Typographie web - n°11, Jason Santa Maria, 2015, 160 pages
Web designer cherche client idéal - n°12, Mike Monteiro, 2015, 152 pages.

TABLE DES MATIÈRES

AVANT-PROPOS

On n'a jamais manqué de métaphores pour décrire le Web. Au début, c'était notre nouvelle presse à imprimer ; au fil du temps, c'est devenu notre terrain de jeu, puis notre marché global. Aujourd'hui, c'est nos albums photo, nos journaux intimes, nos blogs de voyage, nos instants partagés, nos vidéos, nos GIF, etc., etc. Et avec l'explosion des appareils portables offrant un accès permanent au Web, plus de gens y accèdent aujourd'hui qu'à aucun autre moment de sa courte existence. On peut convenir que le Web est bien plus que la somme de ses parties : d'un tas de câbles échangeant des paquets de données, il est devenu le lieu où nous publions, vendons, communiquons, travaillons et jouons.

Mais voilà le problème : le Web est bien plus fragile que nous ne voulions l'admettre. Il est truffé d'incertitudes - une connexion peut être coupée, la latence d'un réseau peut être trop élevée - qui font que des pans entiers de nos designs risquent de ne jamais parvenir à nos utilisateurs. Bien sûr, il est tentant de voir cela comme un problème temporaire, un problème qui se résorbera progressivement à mesure que les appareils et les réseaux s'amélioreront. Mais entre l'infrastructure vieillissante des économies développées et la popularité croissante des appareils mobiles, plus économiques et moins énergivores sur les marchés émergents, on a l'impression d'assister à l'avènement d'une nouvelle norme pour le Web - un support auquel on accède depuis les quatre coins de la planète, mais qui est également beaucoup plus lent que ce qu'on avait imaginé.

Cela pourrait nous angoisser, mais l'histoire ne s'arrête pas là.

Lorsque les « mobiles » sont entrés dans l'équation, nous avons eu une opportunité : au lieu de créer plusieurs sites séparés pour différents types d'appareils, nous nous sommes rendu compte que nous pouvions utiliser des mises en page flexibles et des *media queries* pour concevoir des sites *responsive* (réactifs), des mises en page qui peuvent potentiellement s'adapter à une infinité d'écrans de tailles différentes.

Et aujourd'hui, nous avons une autre opportunité : celle de nous assurer que nos mises en page ne soient pas simplement adaptatives, mais viables à long terme - qu'elles soient conçues

pour offrir un contenu attrayant et des interfaces riches, non seulement pour les appareils les plus récents et les bandes passantes les plus larges, mais pour tous les écrans du monde.

Par chance, Scott Jehl est là pour nous montrer la voie à suivre.

J'ai eu le plaisir de travailler avec Scott sur plusieurs projets de responsive design, et je n'ai jamais vu de web designer ayant autant de considération et de respect pour la fragilité du Web que lui. Et dans ce petit livre, Scott partagera cette expertise avec vous, cher lecteur, en vous apprenant à concevoir des interfaces agiles et légères qui sauront s'adapter à la volatilité du Web.

Ces dernières années, à force de pratiquer le responsive design, nous avons appris à nous défaire de notre besoin de contrôler la largeur et la hauteur de nos mises en page. Aujourd'hui, Scott Jehl nous montre l'étape suivante : construire des designs responsive de façon responsable, faire en sorte qu'ils soient non seulement adaptés à des écrans de différentes tailles, mais à la forme mouvante d'un Web universel qui ne fait pas de différence.

À vous de jouer.

Ethan Marcotte

INTRODUCTION

Début 2012, ma femme et moi avons loué un appartement à Siem Reap, au Cambodge. Elle était venue faire du bénévolat dans un hôpital pour enfants ; quant à moi, je continuais mon travail de web designer à distance avec mes collègues de Filament Group aux États-Unis. J'ai ainsi travaillé pendant plusieurs mois au gré de nos déplacements, alors que nous traversions certaines des régions les plus démunies du monde – le Laos, l'Indonésie, le Sri Lanka et le Népal. Chaque étape du parcours m'a donné l'occasion d'utiliser le Web dans les mêmes conditions, souvent limitées, que les gens qui vivaient là. Mes préjugés de designer et ma patience d'utilisateur ont volé en éclats.

Vous avez probablement déjà lu quelque part que les services mobiles étaient le principal mode d'accès à Internet dans certains pays en voie de développement, et mes observations personnelles le confirment. Des vitrines remplies d'appareils mobiles dont j'ignorais l'existence inondaient les marchés (et m'ont permis de remplir mon sac d'appareils à tester). Non seulement tout le monde semblait avoir un téléphone relié à Internet, mais j'ai été surpris de constater que les gens se servaient fréquemment des réseaux cellulaires pour connecter d'autres appareils au Web. Le moyen le plus courant pour connecter un ordinateur portable à Internet était d'acheter une carte SIM prépayée et une clé USB. C'est donc ce que j'ai fait.

Utiliser le Web de cette façon a mis ma patience à rude épreuve. Combien d'heures ai-je perdu à basculer entre plusieurs onglets en cours de chargement et à cliquer sur « Rafraîchir » dans l'espoir d'arriver enfin à ouvrir une application Web, grignotant petit à petit le quota de données limité de ma carte prépayée. En fervent défenseur de bonnes pratiques comme l'amélioration progressive *(progressive enhancement)* et le responsive design, je me plaisais parfois à penser que si ces sites avaient été « bien conçus », ces problèmes n'existeraient pas. Mais la vérité, c'est que beaucoup de ces bonnes pratiques ne fonctionnent pas aussi bien que prévu. Malheureusement, il m'est apparu que la promesse simple d'un accès universel au Web était loin de se concrétiser.

Je ne suis pas le premier à m'en rendre compte. Un article paru dans *Wired* en 2014 décrit l'expérience de cadres de Facebook voulant utiliser leur propre service au cours d'une visite au Nigeria, où plus de 30 % des utilisateurs d'Internet sont inscrits sur Facebook (http://bkaprt.com/rrd/0-01/) :

> *Nous avons lancé l'appli, et puis nous avons attendu… et attendu… Ça a pris un temps fou. Même des fonctionnalités simples, comme télécharger des photos – des choses que font la plupart des utilisateurs de Facebook – ne fonctionnaient tout simplement pas. Ça a été un coup dur pour nous. Nous avions développé une application pour des utilisateurs comme nous. Mais nous étions l'exception, pas la règle.*

Nous autres développeurs web sommes des gens exceptionnels. Notre travail demande des réseaux rapides et fiables pour transmettre d'énormes quantités de données, et nous avons accès aux appareils les plus récents et les plus puissants. Mais si la plupart d'entre nous travaillent dans des conditions relativement idéales, nous ne pouvons pas nous contenter de concevoir pour des utilisateurs comme nous ; nous ne pouvons pas oublier que pour la plus grande partie du monde, le Web ne fonctionne pas comme ça.

Vous vous dites peut-être, « mais ce n'est pas le public que je vise », et vous avez peut-être raison. Mais songez au fait que la majorité du trafic web mondial proviendra cette année d'appareils bon marché sur les marchés émergents (http://bkaprt.com/rrd/0-02/). Même dans certaines régions les plus développées, les connexions mobiles sont souvent lentes, intermittentes et instables, et les forfaits de données sont de plus en plus chers et limités. Une rapide recherche sur Twitter confirmera que le réseau cellulaire de Londres est toujours notoirement mauvais, et là où j'habite en Floride, j'ai rarement l'occasion d'avoir mieux qu'une connexion EDGE.

Bon nombre de nos voisins, de nos utilisateurs, de nos clients ne disposent pas d'un accès fiable et efficace au Web. En tant que Web designers, nous sommes bien placés pour améliorer cette situation. Si je mentionne les clients, c'est parce que promouvoir un meilleur accès n'est pas seulement une question

d'altruisme, c'est également un moyen d'étendre la portée de nos services et de les rendre plus robustes pour tous ceux qui y accèdent.

Ce livre traite d'accessibilité : élargir l'accès aux services que nous produisons sans compromettre les fonctionnalités qui font avancer le Web. La diversité est une caractéristique essentielle du Web, pas un bug. Nous devons nous efforcer de rendre notre contenu et nos services accessibles à tous les appareils qui en sont capables. Si cela semble difficile, eh bien, c'est parce que ça l'est parfois. Mais j'ai l'intention de vous convaincre que c'est possible, et que ça en vaut grandement la peine.

Commençons par un petit récapitulatif de ce que font nos utilisateurs.

Notre Web se diversifie

Les chiffres en attestent. Apple a vendu plus d'appareils iOS en 2011 que d'ordinateurs au cours des 28 années précédentes (http://bkaprt.com/rrd/0-03/). En 2013, l'usage mondial de données mobiles a crû de 81 % (http://bkaprt.com/rrd/0-04/). En janvier 2014, 58 % des Américains possédaient un smartphone et 42 % possédaient une tablette, quatre ans après le lancement de l'iPad (http://bkaprt.com/rrd/0-05/). La vitesse de cette croissance est époustouflante, mais il ne s'agit pas que des mobiles.

Nos appareils représentent une palette de plus en plus large de facteurs de forme, de fonctionnalités, de contraintes environnementales et d'utilisations (FIG 0.1). La variabilité de la taille des écrans à elle seule est stupéfiante – voyez plutôt ce graphi-

FIG 0.1 : Un échantillon des divers écrans que nous devons prendre en charge aujourd'hui

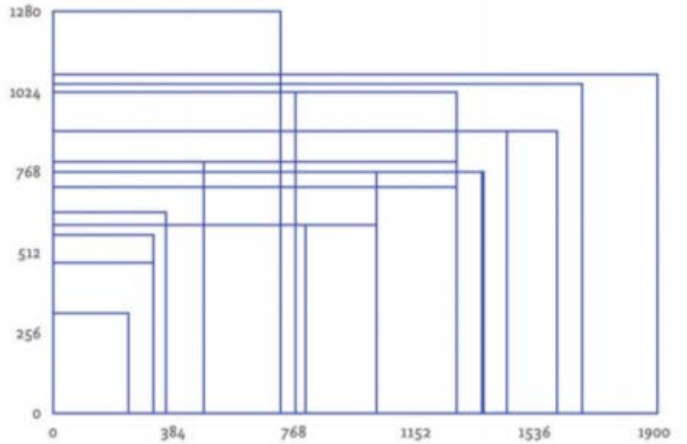


FIG 0.2 : Les différentes tailles d'écran des vingt appareils les plus répandus (http://bkaprt.com/rrd/0-06/)

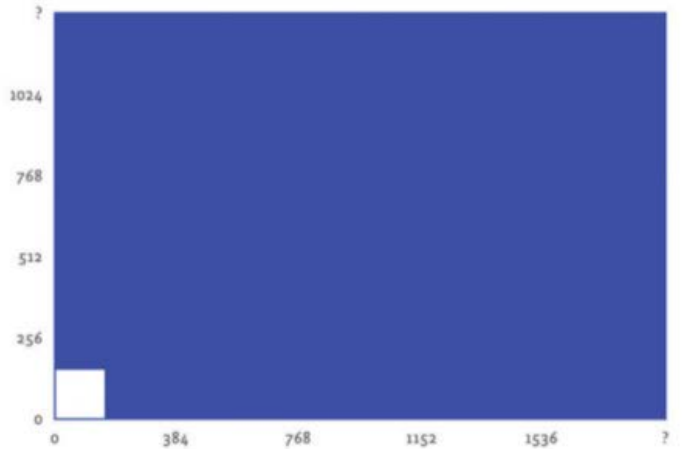


FIG 0.3 : Fragmentation de la taille des viewports sur le Web, redessinée à partir d'un tweet de Cennydd Bowles (http://bkaprt.com/rrd/0-07/)

que illustrant les dimensions des vingt écrans les plus répandus au début de l'année 2013 (FIG 0.2).

Les dimensions de l'écran ne sont pas un indicateur de la résolution de l'affichage, qui peut être plus élevée que la définition standard ; elles ne permettent pas non plus de prédire la taille de la fenêtre du navigateur, ou *viewport*, qui diffère souvent de la taille de l'écran. Comme le fait remarquer le designer Cennydd Bowles, étant donnée la variabilité quasi infinie du viewport du navigateur, les dimensions que nous devons prendre en charge sont bien plus nombreuses que ce que le classement des écrans ne laisse entendre (FIG 0.3).

Alors ça, c'est de la fragmentation ! Heureusement, nous avons plus ou moins résolu le problème de la conception d'un design qui s'adapte de manière fluide aux viewports de toutes tailles.

Responsive design : un point de départ responsable

> *Plutôt que de créer des designs déconnectés, conçus chacun pour un appareil ou un navigateur particulier, nous devrions les traiter comme les facettes d'une même expérience.*
>
> Ethan Marcotte, « Responsive web design », *A List Apart*

En 2010, Ethan Marcotte a inventé le terme « responsive web design » pour décrire une approche du web design combinant des grilles fluides, des images fluides et des media queries CSS3 afin de produire des sites web qui réagissent comme par magie à leur environnement (FIG 0.4).

Si le titre de ce livre vous a amené à croire que le responsive web design n'était pas responsable, permettez-moi de clarifier tout de suite. Un design responsive est un design responsable. Fin. Merci de votre attention.

Bon, restons sérieux.

Avec cette combinaison ingénieuse de plusieurs technologies standard du Web, Marcotte nous a offert un moyen durable de développer des mises en page visuelles capables de s'adapter à tous les appareils. Mais il serait le premier à insister sur le fait qu'une mise en page responsive n'est que l'une des nombreuses variables à prendre en compte lorsque nous concevons des sites

FIG 0.4 : Exemple donné par Ethan Marcotte d'une mise en page responsive dans son article paru sur *A List Apart* (http://bkaprt.com/rrd/0-08/)

et des applications multi-appareils. La mise en page n'est que le départ. Nous devons voir plus loin que la fenêtre du navigateur et envisager des moyens de prendre en charge la myriade de fonctionnalités qu'offrent différents appareils, déterminer comment préserver l'accessibilité des interfaces les plus complexes et comment transmettre nos ressources de manière optimale.

Comme l'écrit Trent Walton dans son essai « Device Agnostic » : « De même que les voitures sont conçues pour rouler par une chaleur extrême comme sur des routes verglacées, les sites web doivent être conçus pour faire face à la réalité de la variabilité inhérente au Web » (http://bkaprt.com/rrd/0-09/). Heureusement, être responsive du point de vue de la mise en page ne nous empêche pas d'être également responsive du point de vue des performances, de l'interactivité ou de l'accessibilité.

Responsive et responsable

Pour tenir notre promesse d'un Web accessible au plus grand nombre, plaisant à utiliser et durable, nous devons combiner le

responsive design avec quelques principes de design responsable. Un projet de design responsive et responsable doit prendre en compte les facteurs suivants à parts égales :

- **Utilisabilité** : la façon dont l'interface utilisateur d'un site web est présentée à l'utilisateur, et dont elle répond à différentes interactions et conditions de navigation.
- **Accès** : la possibilité pour les utilisateurs de tous les appareils, navigateurs et technologies d'accessibilité d'accéder aux fonctionnalités et au contenu de votre site et de les comprendre.
- **Durabilité** : la capacité pour les technologies d'un site web ou d'une application de fonctionner sur les appareils qui existent aujourd'hui et de continuer à être utilisables et accessibles par tous les utilisateurs, appareils et navigateurs à l'avenir.
- **Performances** : la vitesse à laquelle les fonctionnalités et le contenu d'un site sont transmis à l'utilisateur et l'efficacité avec laquelle ils fonctionnent au sein de l'interface utilisateur.

C'est plutôt complet, non ? Avant que je ne renomme ce livre *Bienvenue sur Internet par Scott*, examinons de plus près les défis que nous rencontrons pour répondre de manière responsable aux besoins de nos utilisateurs.

Concevoir pour une meilleure utilisabilité : capteurs, mécanismes de saisie et interactivité

Il est révolu le temps où les sites web que l'on concevait devaient seulement pouvoir être utilisés à l'aide d'une souris (si ce temps a jamais existé). Aujourd'hui, nous devons prendre en compte les interfaces tactiles, les claviers, les stylets, etc. que nous rencontrons sur un mélange d'appareils mobiles, de tablettes et d'ordinateurs portables. Les appareils les plus populaires du moment sont nombreux à prendre en charge l'interactivité tactile. Par exemple, le système d'exploitation Windows 8 supporte les interactions tactiles sur les ordinateurs portables comme sur les tablettes (FIG 0.5). Microsoft Kinect suit les mouvements des mains et des bras en temps réel (FIG 0.6). En réponse à ces nouveaux mécanismes de saisie, nous ne pouvons pas nous appuyer uniquement sur les traditionnelles interactions de la

FIG 0.5 : Le système d'exploitation Windows 8 fonctionne sur des appareils qui prennent en charge plusieurs modes de saisie, de l'écran tactile à la souris en passant par le clavier. Photographie de Kārlis Dambrāns (http://bkaprt.com/rrd/0-10/).

FIG 0.6 : Le Kinect de Microsoft suit les mouvements du corps entier, ce qui laisse peut-être entrevoir de futurs modèles d'interaction pour le Web. Photographie de Scott et Elaine van der Chijs (http://bkaprt.com/rrd/0-11/).

souris, comme le survol des liens ; nos interfaces doivent être prêtes à prendre en charge divers mécanismes de saisie au sein de notre univers multi-appareil.

Il existe souvent une disparité entre la puissance des applications natives et les API limitées que nous voyons sur le Web, et c'est un véritable obstacle à la conception d'applications web. Par chance, de plus en plus de navigateurs ont maintenant accès aux fonctionnalités natives du système d'exploitation, telles que le positionnement GPS, les contacts, le calendrier, les notifications, les systèmes de fichiers et la caméra. Ces interfaces standardisées nous permettent de communiquer avec les fonctions locales de l'appareil sans utiliser de plug-ins comme Flash ou Java, qui de toutes façons fonctionnent rarement sur certains appareils. En plus des données des API locales, de plus en plus de navigateurs sont capables d'accéder aux informations des capteurs de l'appareil, tels que le capteur de proximité, le GPS, l'accéléromètre, le niveau de batterie et même l'éclairage ambiant. Avec chaque nouvelle fonctionnalité, la plateforme web fait une nouvelle percée.

Concevoir pour l'accès : technologies d'accessibilité et continuité entre les appareils

Comme les technologies d'accessibilité sont de plus en plus souvent installées par défaut sur nos appareils, nous devons nous assurer que nos sites gardent un sens lorsqu'on les parcourt dans un contexte non visuel. Tous les ordinateurs Apple et les appareils iOS incluent désormais par défaut le logiciel de lecture d'écran VoiceOver, qui est intégré au navigateur et offre un système de navigation tactile tout en lisant la page à voix haute. Son système tactile rotatif « multitouch » permet par exemple de naviguer le Web en parcourant les titres et les liens, ce qui nous donne d'autant plus de raisons d'être vigilants quant aux balises que nous utilisons pour communiquer notre contenu (FIG 0.7).

Les technologies d'accessibilité ne sont pas uniquement destinées aux personnes handicapées ; la voix et l'audio peuvent être le mode d'interaction préféré et le plus sûr pour tous les utilisateurs dans certains contextes. Le lecteur d'écran le plus utilisé est probablement Siri d'Apple, qui est pratique pour les

FIG 0.7 : Rotor de VoiceOver sur l'iPhone (http://bkaprt.com/rrd/0-12/)

personnes qui ne peuvent temporairement pas regarder leur écran (au volant, par exemple), ou qui préfèrent l'aisance de l'interaction vocale à la saisie manuelle. Alors que les applications web continuent à faire des percées dans nos systèmes d'exploitation natifs, nous pouvons nous attendre à ce que les logiciels de ce genre soient de plus en plus répandus.

En plus d'offrir une expérience utilisable dans les contextes isolés, nous devons garder à l'esprit que les gens passent de plus en plus fréquemment d'un appareil à l'autre et s'attendent à pouvoir accéder à votre contenu en toutes circonstances. Une étude publiée par Google en 2012 intitulée *The Multi-Screen World* a révélé que beaucoup de gens utilisaient plusieurs appareils au cours d'une même journée, et bien souvent pour accomplir une même tâche (FIG 0.8). L'étude a notamment révélé que 65 % des acheteurs qui ajoutaient un article à leur panier d'achats sur un appareil mobile concluaient la transaction plus tard sur un ordinateur portable. Il est possible que la personne soit interrompue

FIG 0.8 : L'étude publiée par Google en 2012 intitulée The New Multi-Screen World (http://bkaprt.com/rrd/0-13/)

par un coup de fil ou préfère terminer la procédure de paiement sur un appareil pourvu d'un clavier. Quelle qu'en soit la raison, nous devons accueillir nos utilisateurs où qu'ils se trouvent.

Navigateurs : la tendance est au vintage

Alors que les navigateurs modernes tels que Google Chrome, Firefox et même Internet Explorer ne cessent de sortir de nouvelles fonctionnalités, beaucoup d'appareils dans la nature et dans les magasins utilisent un navigateur qui n'est plus en cours de développement. Par exemple, la version 2 du système d'exploitation Android est toujours très répandue dans le monde alors qu'il en est aujourd'hui à la version 5, et elle inclut un navigateur intégré qui n'a pas été mis à jour depuis 2011 (http://bkaprt.com/rrd/0-14/) ! Les développeurs chevronnés (c'est-à-dire vieux) comme moi se souviendront peut-être d'une situation similaire avec l'interminable fin de règne d'Internet Explorer 6. Malheureusement, à terme, le support des navigateurs finit toujours par être délaissé au profit d'une nouvelle plateforme ou des nouvelles priorités de l'entreprise, abandonnant ainsi tous les utilisateurs existants.

FIG 0.9 : Un graphique d'Opera illustrant comment le navigateur Opera Mini accède au Web (http://bkaprt.com/rrd/0-15/)

Ces contraintes des navigateurs constituent un défi, mais sont également une chance. Les utilisateurs recherchent souvent d'autres navigateurs pour leurs plateformes, dont certains offrent des fonctionnalités et des arguments de vente inhabituels. Par exemple, des millions de gens qui préfèrent que les pages web se chargent rapidement et consomment moins de données sur leur forfait (dingue, non ?) choisissent un navigateur comme Opera Mini, qui télécharge tout le contenu Web par l'intermédiaire de serveurs proxy qui optimisent la taille de téléchargement de chaque page (FIG 0.9). Les navigateurs qui fonctionnent ainsi prennent peu de JavaScript en charge, voire pas du tout ; ironiquement, des pratiques bénéfiques pour des navigateurs plus anciens, telles qu'un code HTML fonctionnel, sont également d'une aide précieuse pour les millions de personnes qui utilisent ces nouveaux navigateurs par proxy !

Optimiser les performances : réseaux, poids et incidence sur les performances

Les contraintes posées par les réseaux mobiles sont de plus en plus nuancées et difficiles à cerner, bien qu'elles tendent globalement à s'améliorer. Pour le moment, les connexions aux réseaux mobiles sont intermittentes et lentes, même dans les pays développés. Pour garantir de bonnes performances, nous devons revoir notre façon de transmettre notre contenu, réduire le poids et le volume de nos ressources et résoudre les dysfonctionnements qui empêchent d'y accéder.

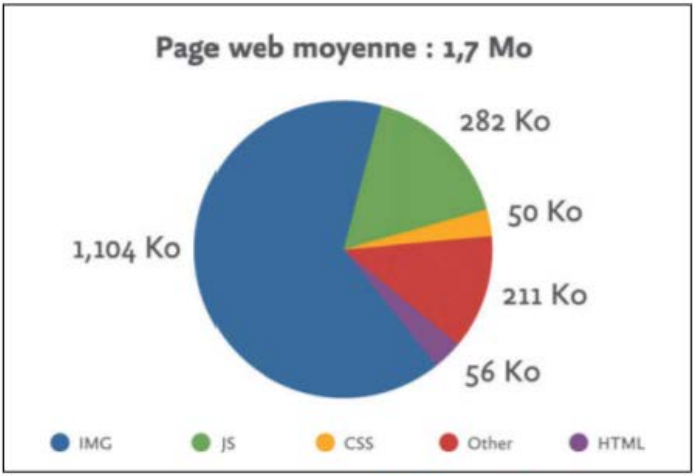


FIG 0.10 : Poids moyen des sites Web, avril 2014 (http://bkaprt.com/rrd/0-17/)

Toute portion de code inutilisée que nous transmettons est une perte de temps et d'argent pour nos utilisateurs, et nous avons une grande marge de manœuvre pour nous améliorer. Dans un billet publié en avril 2013 et intitulé « What are Responsible Websites Made Of ? », Guy Podjarny a analysé la taille des fichiers transmis par 500 sites web responsive et a constaté que 86 % d'entre eux transféraient des ressources similaires à tous les appareils (http://bkaprt.com/rrd/0-16/). Plutôt que de servir des images optimisées, par exemple, les sites fournissaient des images pour grand écran (comptant ainsi sur le navigateur pour les redimensionner à la taille de l'écran) ainsi que pléthore de CSS, de JavaScript et d'autres fonctionnalités qui n'étaient nécessaires que dans certains contextes.

Bien sûr, le problème du poids du Web n'est pas l'exclusivité du responsive design. Il prolifère depuis longtemps sur le Web de bureau ; en avril 2014, un site web pesait en moyenne 1,7 Mo (FIG 0.10).

Des sites web lourds et mal optimisés entraînent des temps de chargement plus longs pour les utilisateurs. Une étude réalisée

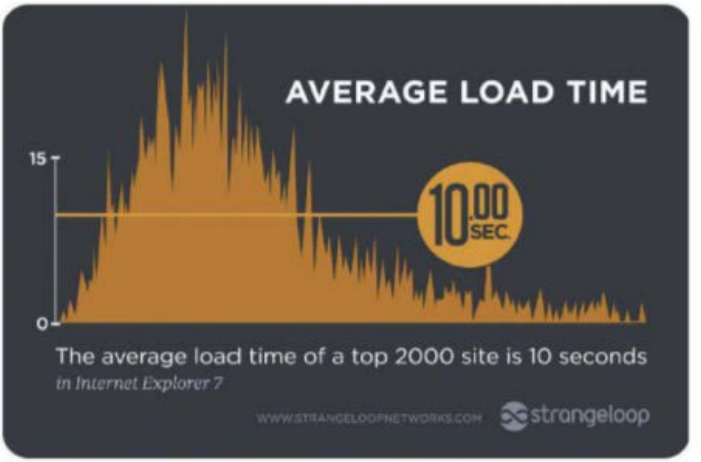


FIG 0.11 : La taille de transfert d'un site affecte sa vitesse de chargement (http://bkaprt.com/rrd/0-19/)

en 2012 par StrangeLoop sur les sites du Top 2 000 d'Alexa a établi que le temps de chargement moyen d'un site était de six à dix secondes sous Internet Explorer 7 avec une connexion wi-fi, alors ne parlons même pas de ce que ce serait sur une connexion mobile (FIG 0.11) ! De mauvaises performances ont une incidence directe sur les utilisateurs, et par conséquent sur l'activité commerciale. En 2012, Walmart a constaté que pour chaque seconde de temps de chargement en moins, le taux de conversion augmentait de 2 % ; ses recettes augmentaient d'un pour cent pour chaque diminution de 100 millisecondes (http://bkaprt.com/rrd/0-18/).

Par ailleurs, alors que la vitesse et la fiabilité des réseaux cellulaires limitent toujours l'usage de données mobiles, le coût des données elles-mêmes est devenu prohibitif. En 2014, si vous achetiez un iPhone dans un Apple Store aux États-Unis, le forfait le moins cher de Verizon à 60 $ par mois comprenait seulement 250 Mo de données. Avec une taille de page moyenne à

1,7 Mo, vous comprenez mieux pourquoi votre quota mensuel s'épuise aussi vite !

Comme un usage responsable du réseau est vital pour les performances, ce livre consacre plusieurs chapitres à la minimisation des données transférées, mais également à l'optimisation de la transmission de notre code afin que nos sites soient utilisables aussi rapidement que possible.

Accepter l'imprévisible

Le Web a toujours été un support de design hostile. Alors que l'usage d'appareils multiples est en hausse, les scénarios tels qu'une faible bande passante, un écran de petite taille, une orientation imprévisible de l'écran ou une navigation non visuelle ne sont plus des exceptions ; ce sont des contextes quotidiens. Il n'est plus possible de concevoir selon un ensemble de conditions inflexibles, car les interfaces que nous concevons et développons sont de plus en plus utilisées dans des circonstances que nous ne pouvons ni prévoir ni contrôler.

Pour répondre aux besoins de tous nos utilisateurs sur le Web aujourd'hui, nous devons rendre nos designs responsive dans le moindre détail et préparer notre code pour parer à toute éventualité. Et pour cela, nous devons prendre en compte à la fois les modes d'utilisation actuels et potentiels. Nous devons créer des sites responsive qui mettent l'accent sur les performances, l'accessibilité, l'utilisabilité et la durabilité, mais il n'est pas simple de parvenir à ces objectifs. Dans ce livre, nous explorerons les défis auxquels nous devons faire face pour implémenter un design responsive. En suivant des pratiques et des modèles responsables, nous pouvons éliminer bon nombre de problèmes d'accessibilité et de performance avant qu'ils ne se présentent, pour offrir une expérience adaptée et optimisée quelles que soient les fonctionnalités et les contraintes du navigateur.

Notre objectif est de créer des expériences plaisantes et inclusives (rien que ça), alors poursuivons et familiarisons-nous avec quelques manières d'aborder les défis que nous devons relever.

1 DESIGN RESPONSABLE

Mon amour du responsive design repose sur l'idée que mon site web vous accueillera où que vous soyez – sur un portable, un ordinateur de bureau ou tout autre appareil.

Trent Walton, « Fit To Scale » (http://bkaprt.com/rrd/1-01/)

Le responsive design, avec ses principes fondamentaux (grilles fluides, images fluides et media queries), constitue une avancée considérable vers un Web plus universel qui transcende les différences entre appareils. Cependant, il repose sur des fonctionnalités qui sont susceptibles de ne pas marcher comme prévu – voire pas du tout. Nos sites doivent réagir à toutes sortes d'imprévus, qu'il s'agisse du comportement de nos utilisateurs, des conditions du réseau ou de scénarios de compatibilité uniques.

Dans ce chapitre, nous allons explorer deux principes responsables : l'utilisabilité et l'accessibilité. Nous aborderons des considérations d'ordre général avant d'étudier des exemples de code concrets et conçus pour durer que vous pourrez implémenter dès maintenant. Pour commencer, parlons de design.

CONCEVOIR POUR L'UTILISABILITÉ

Lorsque nous pensons à l'utilisabilité d'un design responsive, nous cherchons un moyen de présenter le contenu et les fonctionnalités du design sur différents appareils et tailles d'écran. Les composants de l'interface doivent-ils faire place au contenu lorsque la surface de l'écran est réduite ? Ces composants fonctionnent-ils de manière intuitive en réponse aux différents modes de saisie ? Le contenu et la hiérarchie sont-ils clairement balisés ? La longueur des lignes permet-elle une lecture agréable sur les écrans de toutes tailles ?

Passez rapidement au navigateur

> *Ne parlons plus de « concevoir dans le navigateur », mais de « décider dans le navigateur ».*
> Dan Mall, The Pastry Box Project (http://bkaprt.com/rrd/1-02/)

Chez Filament Group, nous démarrons la plupart de nos projets sous Adobe Illustrator, avec lequel nous itérons des concepts de design visuel généraux. Nous essayons ensuite de passer au code aussi vite que possible. À ce stade, nous visons à concevoir le strict nombre de variations nécessaire pour communiquer un plan pour la mise en page et l'interactivité sur différents viewports - de simples suggestions pour différents types d'appareils. Nous ne cherchons pas encore à tirer parti des différents mécanismes de saisie et des fonctionnalités de chaque navigateur, ni à établir quel appareil recevra quelle variante de la mise en page. L'objectif est de passer au navigateur aussi vite que possible pour prendre des décisions de design et d'interaction en contexte, ce qui se traduira par des recommandations plus informées pour nos clients.

Trouvez vos points de rupture

Les points de rupture sont les dimensions auxquelles nous passons d'une mise en page fluide à une autre à l'aide de media queries. Voici deux exemples :

```
/* premier point de rupture */
@media (min-width: 520px){
  ...styles pour les viewports de 520 px de largeur et
  plus
}
/* deuxième point de rupture */
@media (min-width: 735px){
  ...styles pour les viewports de 735 px de largeur et
  plus
}
```

S'il est tentant de choisir des points de rupture tôt dans le processus de design, basés par exemple sur les dimensions d'appareils courants que nous voulons prendre en charge, la vérité, c'est qu'il ne faut pas les choisir. Nous devons plutôt les trouver en nous laissant guider par notre contenu.

> *Commencez d'abord par une fenêtre de petite taille, puis élargissez-la jusqu'à ce que ça ait l'air moche. C'est le moment d'ajouter un point de rupture !*
> Stephen Hay, http://bkaprt.com/rrd/1-03/

Le design et le contenu de la page doivent modeler et influencer les points de rupture. Comme Hay le fait remarquer, le moyen de plus simple de trouver des points de rupture est simplement de redimensionner la fenêtre du navigateur jusqu'à ce que le contenu devienne bizarre (c'est le terme technique) à utiliser ou à lire – et *presto*, un point de rupture !

En plus de votre instinct, vous pouvez vous fier à des recommandations un peu plus objectives. D'après le livre de Richard Rutter écrit en hommage à Robert Bringhurst, *The Elements of Typographic Style Applied to the Web* (http://bkaprt.com/rrd/1-05/), la mesure (nombre de caractères par ligne dans une colonne de texte) optimale pour une lecture immersive est généralement donnée entre 45 et 75 caractères, espaces compris (FIG 1.1). Lorsque vous élargissez votre fenêtre, faites attention au moment où une colonne de texte approche cette plage : c'est probablement là que vous devrez ajuster votre mise en page.

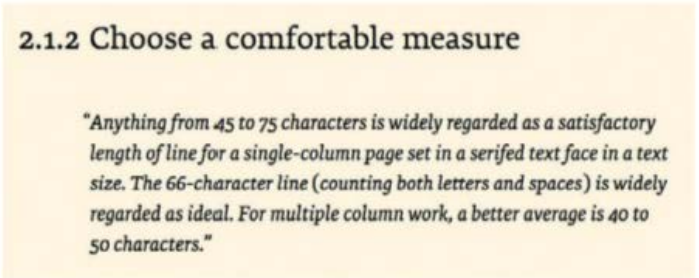


FIG 1.1 : Dans cet exemple, la mesure est de soixante-dix caractères par ligne, ce qui permet une lecture confortable (http://bkaprt.com/rrd/1-04/).

Lorsque vous travaillerez sur des projets de responsive design complexes, vous vous apercevrez que ces points de rupture se situent souvent à différentes tailles pour différentes portions d'une mise en page, et que certains sont plus importants que d'autres.

Les points de rupture majeurs marquent des changements importants, consistant généralement à ajouter des colonnes ou à modifier radicalement la présentation de plusieurs éléments ; les points de rupture mineurs impliquent généralement de légers ajustements du design (comme de changer l'attribut `font-size` d'un élément pour éviter le retour à la ligne) qui exploitent les espaces restants entre les points de rupture majeurs. Généralement, les points de rupture majeurs sont décidés tôt dans le développement, tandis que les points de rupture mineurs sont ajoutés à titre de touche finale. Moins nous utiliserons de points de rupture et plus le design sera simple à entretenir.

Intéressons-nous à un exemple. Sur le site web du *Boston Globe*, la mise en page comporte deux ou trois points de rupture majeurs, mais les composants plus complexes varient plus fréquemment. L'encadré d'en-tête du site comporte quatre points de rupture majeurs, plus quelques ajustements mineurs pour éviter les retours à la ligne (FIG 1.2).

Majeur

Premier point de rupture : les options de navigation et de recherche se déroulent d'un clic.

Majeur

Deuxième point de rupture : le logo passe à gauche pour partager la largeur avec les icônes de navigation.

Majeur

Troisième point de rupture : le logo repasse au centre, la boîte de recherche est visible à tout moment.

Majeur

Quatrième point de rupture : la boîte de recherche passe à droite du logo, la barre de navigation s'élargit.

Mineur

Point de rupture final : la boîte de recherche s'élargit, la liste des petites annonces est visible à tout moment dans le coin supérieur gauche.

FIG 1.2 : Points de rupture majeurs et mineurs de l'en-tête du Boston Globe

Concevez de façon modulaire

Comme dans l'exemple d'en-tête qui précède, je trouve utile de compiler les différentes variations de chaque composant séparément ; ainsi, je peux tester son utilisabilité et documenter ses variations en un seul endroit. Le développeur Dave Rupert, de Paravel, explore ce concept dans un billet intitulé « Responsive Deliverables » (http://bkaprt.com/rrd/1-06/). Rupert écrit : « Des livrables responsive doivent être des systèmes pleinement fonctionnels du style Twitter Bootstrap (http://bkaprt.com/rrd/1-07/),

taillés sur mesure pour les besoins de vos clients. » En d'autres termes, nous devons construire et documenter nos composants du plus petit au plus grand, comme les pièces d'un puzzle qui s'emboîtent parfaitement.

Même contenu, moins de bruit

Vous êtes parvenu à trouver des points de rupture horizontaux pour des viewports de différentes tailles. Comment allez-vous faire tenir tout ce contenu sur les plus petits écrans sans que tout soit confus ? Le responsive design essuie parfois des critiques à cause de sites qui essaient d'éviter les situations confuses en dissimulant une partie de leur contenu aux utilisateurs – les empêchant d'accéder à un contenu qui était suffisamment important pour qu'on l'inclue à la base. Si c'est un contenu utile, il est utile pour tout le monde. Comme l'indique Luke Wroblewski dans *Mobile first*, plutôt que de dissimuler le contenu qui n'est pas pratique à afficher, mieux vaut réorganiser le design de sorte à préserver l'utilisabilité sur les plus petits écrans.

Par chance, nous disposons de nombreux modèles de design qui permettent de s'adapter aux contraintes des petits écrans de façon intéressante, intuitive et responsable.

Dévoilement progressif

L'un de ces modèles est le dévoilement progressif (*progressive disclosure*), qui consiste à afficher le contenu à la demande. En clair, il est acceptable de masquer le contenu du moment que vous laissez à vos utilisateurs un moyen d'y accéder. L'idée du dévoilement progressif est simple : masquez des portions du contenu, mais offrez une interface intuitive pour que vos utilisateurs puissent l'afficher s'ils le souhaitent (FIG 1.3).

Le dévoilement progressif est le plus souvent un simple jeu d'afficher-masquer comme dans l'exemple précédent, mais nous avons bien d'autres moyens pour basculer visuellement d'un contenu à un autre. Par exemple, cette annonce immobilière effectue une rotation 3D pour révéler des informations supplémentaires sur la propriété, comme son adresse et son emplacement sur une carte (FIG 1.4). Pour les navigateurs qui ne prennent pas en charge les animations 3D en CSS, les utilisateurs peuvent ouvrir la carte sans transition animée, tandis

FIG 1.3 : L'interface de navigation du Boston Globe utilise un système de dévoilement progressif sur les petits écrans.

que les navigateurs les plus basiques affichent la carte en toutes circonstances, sous les informations du bien.

La « mise en page hors canvas », terme inventé par Luke Wroblewski dans son article « Off-Canvas Multi-Device Layouts », décrit une autre approche notable pour réduire la complexité sur les petits écrans (http://bkaprt.com/rrd/1-08/). Wroblewski décrit plusieurs modèles de design consistant à placer les composants non prioritaires de l'interface en dehors de l'écran jusqu'à ce que l'utilisateur les appelle en touchant une icône ou autre élément similaire ; le contenu qui se trouvait en dehors de l'écran vient alors se superposer ou déplacer le contenu principal (FIG 1.5). Cette approche « à la demande » est de plus en plus répandue sur les petits écrans.

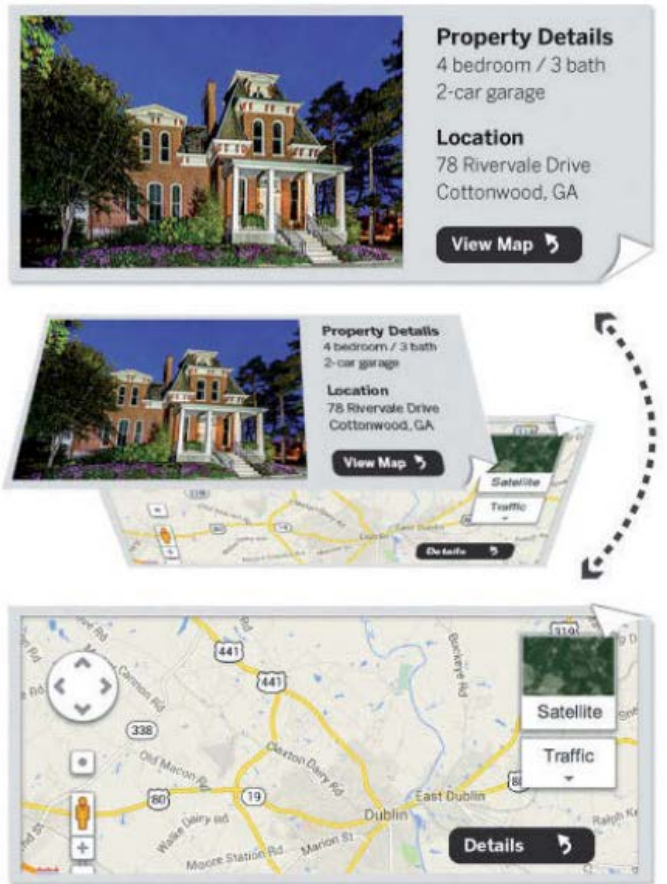


FIG 1.4 : Dévoilement progressif du contenu à l'aide d'une rotation 3D

FIG 1.5 : Lorsque vous touchez l'icône du menu, la barre de navigation hors canvas de Facebook apparaît sur la gauche de l'écran.

Tableaux responsive

Les tableaux sont l'un des types de contenu les plus difficiles à présenter sur petit écran. Il est souvent essentiel que l'utilisateur voie les légendes des lignes et des colonnes associées à une cellule du tableau, mais le nombre de lignes et de colonnes que nous pouvons afficher est limité (FIG 1.6).

Chez Filament, nous avons beaucoup expérimenté et nous avons découvert deux modèles qui marchaient suffisamment bien pour être inclus dans le framework jQuery Mobile. Le premier modèle, Reflow (http://bkaprt.com/rrd/1-09/), reformate le tableau d'une vue multicolonnes à une vue en liste ; chaque cellule devient sa propre ligne, avec la légende de la ligne à sa gauche (FIG 1.7).

Pour cela, Reflow utilise CSS pour donner à chaque cellule du tableau l'attribut `display: block`, créant ainsi une nouvelle ligne, et JavaScript pour récupérer les légendes de chaque colonne du tableau et les insérer dans les cellules qui serviront de légende (tout en masquant les légendes supplémentaires pour les lecteurs d'écran). Reflow convient bien aux tableaux simples

FIG 1.6 : Les tableaux de grande taille peuvent poser des problèmes d'utilisabilité sur les petits écrans.

Rank	1
Movie Title	Citizen Kane
Year	1941
Rating	100%
Reviews	74

Rank	2
Movie Title	Casablanca
Year	1942
Rating	97%

Rank	Movie Title	Year	Rating	Reviews
1	Citizen Kane	1941	100%	74
2	Casablanca	1942	97%	64
3	The Godfather	1972	97%	87
4	Gone with the Wind	1939	96%	87
5	Lawrence of Arabia	1962	94%	87
6	Dr. Strangelove Or How I Learned to Stop Worrying and Love the Bomb	1964	92%	74
7	The Graduate	1967	91%	122
8	The Wizard of Oz	1939	90%	72
9	Singin' in the Rain	1952	89%	85
10	Inception	2010	84%	78

FIG 1.7 : Exemple du modèle de tableau Reflow dans le framework jQuery Mobile, avec le même tableau présenté dans deux largeurs différentes

FIG 1.8 : Exemple du modèle de tableau Column Toggle dans le framework jQuery Mobile, avec le même tableau présenté dans deux largeurs différentes

qui agissent comme des listes formatées, mais sa présentation sur petit écran est peu pratique lorsque vous avez besoin de comparer des données sur plusieurs lignes.

Le modèle Column Toggle (http://bkaprt.com/rrd/1-10/) vient répondre à ce problème. Il fonctionne en affichant sélectivement les colonnes du tableau dans les limites de l'espace horizontal disponible. S'il n'y a pas assez de place, CSS masque les données des colonnes, mais un menu permet à l'utilisateur d'afficher tout de même les colonnes, et il est alors possible de faire défiler le tableau horizontalement (FIG 1.8).

Ce ne sont que deux des nombreux modèles possibles pour présenter le contenu de vos tableaux de manière responsable. Pour plus d'exemples, consultez le projet Responsive Patterns de Brad Frost (http://bkaprt.com/rrd/1-11/). Vous y trouverez toutes sortes de solutions, des composants de navigation horizontaux qui s'escamotent dans des menus lorsque l'espace est limité aux grilles CSS Flexbox pour les mises en page complexes.

CONCEVOIR POUR LE TACTILE (ET TOUT LE RESTE)

Une mise en page responsive n'est que la première étape. Même si votre site s'adapte harmonieusement à toutes les tailles d'écran, vous ne faites pas votre travail si quelqu'un ne parvient pas à l'utiliser. L'interaction tactile n'est pas seulement l'apanage des petits écrans ; de nombreux appareils comportent un écran tactile en plus d'autres mécanismes de saisie. Le nombre de personnes ayant accès à un appareil tactile est en pleine explosion, et nous sommes aujourd'hui tenus d'ajouter le tactile à notre arsenal d'interactions courantes telles que la souris, le focus et le clavier. Les subtilités de l'interaction tactile peuvent sembler intimidantes, mais nul besoin de revoir complètement nos designs pour les interfaces tactiles : l'une des joies du responsive design, c'est qu'il peut être développé avec nos outils de tous les jours. Pour garantir l'utilisabilité d'une interface existante sur les appareils tactiles, deux mesures essentielles :

- Assurez-vous que le contenu offrant une interactivité avec la souris (comme les interactions `hover`) est également accessible dans les navigateurs qui ne comportent pas de pointeur de souris.
- Ne comptez pas sur le fait que le tactile sera utilisé, mais concevez comme s'il allait l'être. Voyons quelle sera l'influence de ces facteurs avec les considérations suivantes.

Réservez les interactions de survol aux raccourcis

L'absence d'interactions `mouseover` (ou de survol) est l'un des plus gros changements à prendre en compte lorsque vous souhaitez prendre en charge les appareils tactiles. De fait, l'absence d'interactions de survol est la raison principale pour laquelle de nombreux sites conçus pour le web de bureau sont inadaptés aux contextes tactiles, produisant des problèmes d'utilisabilité qui empêchent les utilisateurs d'accéder à certaines fonctionnalités. Vous ne pouvez pas compter sur le survol pour des interactions vitales du design, mais vous pouvez l'utiliser comme un moyen alternatif et optionnel d'atteindre un contenu accessible d'une autre façon.

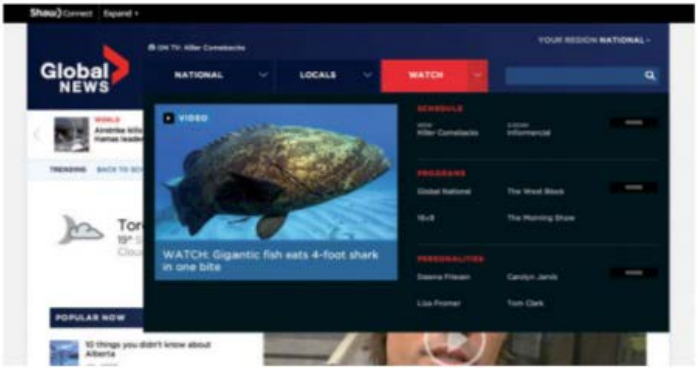


FIG 1.9 : Les menus déroulants de GlobalNews.ca fonctionnent au survol comme sur les écrans tactiles.

Prenez par exemple l'interface de navigation du site web de Global News Canada, conçu par Upstatement et développé par l'équipe du Filament Group (FIG 1.9). La barre de navigation générale amène les utilisateurs vers la page d'accueil des sections National, Locals et Watch lorsqu'ils cliquent sur les liens ou les touchent sur un écran tactile. Ces liens comportent également des menus déroulants qui basculent d'une section à l'autre au survol de la souris. Sur un écran tactile, les utilisateurs sont directement envoyés vers la page d'accueil de la section s'ils touchent le lien, alors nous avons élaboré un mécanisme alternatif pour ouvrir les menus déroulants et prendre en compte tous les points de rupture. C'est le rôle des boutons fléchés situés à droite de chaque lien de la barre de navigation, qui permettent d'ouvrir les menus déroulants d'un clic de souris ou d'une pression du doigt.

Pensez tactile

Une règle générale : les appareils qui accéderont à votre site ne seront pas tous équipés d'écrans tactiles, mais concevez toujours comme si c'était le cas. Les doigts ne sont pas aussi précis qu'un curseur de souris, aussi devons-nous élargir les boutons et les liens pour qu'ils soient plus simples à toucher. Dans quelle

FIG 1.10 : Illustrations de l'article du *Smashing Magazine* (http://bkaprt.com/rrd/1-13/)

mesure ? La discussion reste ouverte, mais les directives d'Apple suggèrent une taille minimale de 44 × 44 pixels pour des boutons utilisables. En se basant sur les travaux du Touch Lab du MIT (http://bkaprt.com/rrd/1-12/), Anthony T suggère dans un article intitulé « Finger-Friendly Design : Ideal Mobile Touchscreen Target Sizes » paru dans *Smashing Magazine* de créer des cibles un peu plus grandes, entre 45 et 57 pixels, et de 72 pixels pour les boutons destinés à être utilisés au pouce, comme ceux qui se trouvent en bas de l'écran d'un appareil mobile (FIG 1.10).

N'oubliez pas les blancs ! L'espace entre les éléments tactiles est aussi important que leur taille. Un bouton plus petit entouré d'espace vide peut être tout aussi simple à utiliser qu'un élément plus grand, et la taille d'un bouton au sein de son empreinte tactile devient alors une question de contraste visuel.

Les gestes courants

Les écrans tactiles offrent un potentiel d'interaction bien plus riche que le simple « tap » - de nombreux gestes tactiles se sont répandus, particulièrement dans les applications natives. Le schéma de Craig Villamor, Dan Willis et Luke Wroblewski illustre quelques gestes courants en matière d'interaction tactile (FIG 1.11).

Vous connaissez probablement la plupart de ces gestes, qui sont utilisés par les systèmes d'exploitation de plusieurs appareils (dont iOS). Dans les navigateurs, ces gestes sont souvent associés à des comportements pratiques par défaut qui varient d'un appareil à l'autre ; certains gestes partagent le même

FIG 1.11 : Illustration des interactions tactiles courantes (http://bkaprt.com/rrd/1-14/)

comportement. Par exemple, un double tap, un pincement ou un étirement sous Safari permettent d'agrandir ou de réduire une zone particulière de la page. Un glisser-déplacer (drag) ou un balayage (flick) dans n'importe quelle direction fera défiler la page, et une pression longue révélera souvent un menu contextuel semblable à ce que vous verriez en cliquant avec le bouton droit de la souris.

Ces gestes natifs ont toutes sortes d'implications sur notre façon de concevoir de manière responsable pour les écrans tactiles. Les utilisateurs s'attendent à ce que les fonctions tactiles de leur appareil se comportent d'une certaine façon, aussi mieux vaut-il éviter de désactiver ou de modifier ces fonctions natives si nous pouvons l'éviter. Bien que les navigateurs nous permettent d'utiliser des événements tactiles tels que `touchstart`, `touchmove` et `touchend` (ou les nouveaux événements de curseur standard `pointerdown`, `pointermove`, `pointerup`, etc.) pour spécifier des gestes avec JavaScript, comment faire pour ne pas entrer en conflit avec les comportements tactiles natifs ?

Gestes sûrs : existent-ils ?

Compilons une liste des gestes que nous pouvons utiliser en toute sécurité sur le Web (spoiler : cette liste est courte). En nous basant sur les gestes natifs des appareils les plus répandus à l'heure actuelle, nous avons le tap, le tap à deux doigts, le drag horizontal et le flick horizontal. Mais même au sein de cette courte liste, il y a des conflits potentiels. Par exemple, Chrome sur iOS et Android permet aux utilisateurs d'effectuer un balayage horizontal pour passer d'un onglet à l'autre, tandis que Safari sur iOS utilise le même geste pour revenir à la page précédente ou suivante, ce qui signifie que notre usage de ces gestes risque de produire des comportements inattendus. Par exemple, si le contenu d'une page s'agrandit au-delà de la largeur du viewport, ce qui se produit souvent, un déplacement horizontal sera généralement utilisé pour faire défiler la page vers la droite ou vers la gauche. Nous devons donc faire attention à ce que nos gestes tactiles personnalisés n'interfèrent pas avec les gestes par défaut.

N'oubliez pas que si j'estime que ces gestes sont sûrs, c'est seulement parce que je ne connais aucun navigateur tactile qui les utilise - pour le moment. Le jour où iOS implémentera le tap à deux doigts, tout ce que nous aurons conçu pour ce geste sera susceptible d'entrer en conflit avec ce comportement natif, ce qui n'est pas *future friendly* du tout. Cela ne veut pas dire que nous devions éviter de développer des gestes personnalisés, mais cela souligne à quel point il est important de concevoir pour de nombreux modes de saisie. Si l'un d'entre eux cesse de fonctionner pour une raison quelconque, nous aurons des moyens alternatifs d'accéder à notre contenu.

En pratique, cela consiste à s'assurer que l'interaction est toujours basée sur une interface souris-clavier. Par exemple, le carrousel des unes sur le site du *Boston Globe* offre plusieurs options interactives (FIG 1.12). Vous pouvez cliquer sur les flèches situées sous le carrousel, cliquer sur les couvertures à droite ou à gauche de l'image centrale, utiliser les flèches directionnelles de votre clavier, ou faire défiler le carrousel sur un écran tactile. Voyez les gestes tactiles comme une amélioration optionnelle venant s'ajouter à des modes de saisie largement pris en charge.

FIG 1.12 : Le carrousel pour modes de saisie multiples sur le site du Boston Globe

L'un des problèmes potentiellement plus importants que posent les gestes tactiles, c'est leur découverte, car ils manquent souvent l'indication visuelle de leur présence. Nous avons rencontré ce dilemme en développant la fonction « articles sauvegardés » du *Boston Globe*, qui permet de sauvegarder des articles sur son compte pour pouvoir les lire plus tard. Sur les petits écrans, le bouton Sauvegarder est masqué par défaut, mais peut être affiché à l'aide d'un tap à deux doigts (FIG 1.13). Bien sûr, il n'y a aucun moyen simple de le savoir à moins de consulter la section d'aide et de lire les instructions !

Scripter les interactions tactiles

Les navigateurs pour écrans tactiles sont généralement capables d'utiliser des composants conçus pour la souris ; certes, vous devrez vous adapter au tactile du point de vue du design, mais vous n'aurez pas forcément besoin de faire quoi que ce soit de spécial avec JavaScript pour prendre en charge les interactions tactiles. Il existe cependant des événements tactiles que vous pouvez scripter pour enrichir et améliorer votre design. Lorsque vous développez des composants, par exemple, il peut être particulièrement utile d'écrire du code pour détecter les événements tactiles parce qu'ils réagissent immédiatement aux interactions tactiles. En comparaison, dans beaucoup de navigateurs tactiles, les événements de souris tels que `click` et `mouseup` se

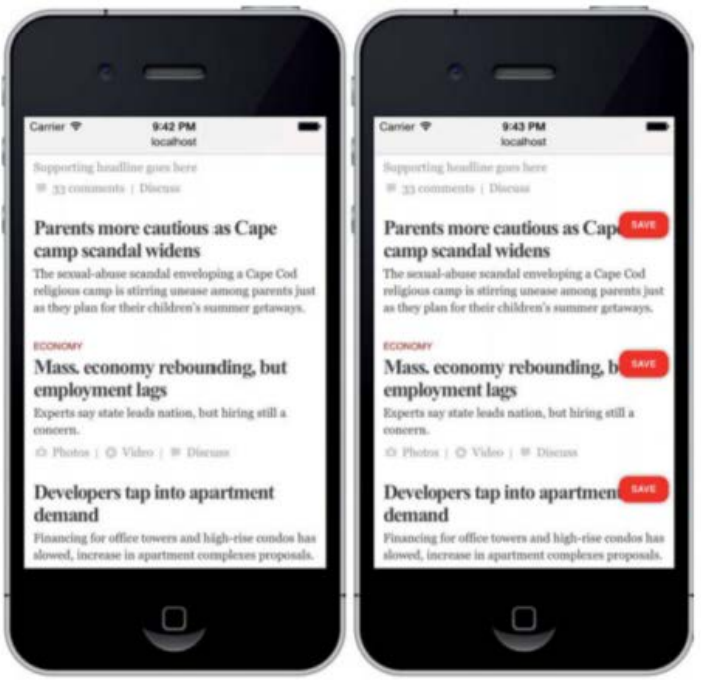


FIG 1.13 : Les boutons de sauvegarde du Boston Globe apparaissent à l'aide d'un tap à deux doigts.

déclenchent généralement 300 millisecondes ou plus après que l'utilisateur a touché l'écran (l'appareil veut s'assurer qu'il ne s'agit pas d'un double tap avant de gérer le `click`), si bien qu'un site uniquement codé pour répondre aux événements de souris souffrira d'un retard mineur, mais notable. Cela étant, il peut être délicat de scripter des gestes tactiles, car la plupart des navigateurs qui les prennent en charge émettent à la fois des événements tactiles et des événements de souris lorsqu'un geste tactile est utilisé. Pour compliquer encore la chose, les navigateurs utilisent parfois des noms différents pour les mêmes

événements tactiles (tels que le très répandu `touchstart` plutôt que le standard émergent, `pointerdown`).

Quelles que soient les optimisations que nous apportons pour les écrans tactiles, il est crucial de ne pas entraver la capacité des gens à interagir avec le contenu par des mécanismes de saisie non tactiles tels que le clavier et la souris. Une approche responsable répandue pour s'assurer que les interactions tactiles fonctionnent aussi rapidement que possible consiste à configurer des détecteurs d'événements pour la souris et le tactile. Au cours d'une interaction, le code gère alors le type d'événement qui se produit en premier et ignore l'autre pour éviter d'exécuter deux fois le même script. Ça a l'air simple, mais ça ne l'est pas. Pour cette raison, je vous recommande d'utiliser une bibliothèque JavaScript open source éprouvée qui fera le gros du travail à votre place. J'utilise Tappy.js (http://bkaprt.com/rrd/1-15/), un script que j'ai créé pour exécuter des événements `tap` personnalisés en écrivant du code jQuery. Un exemple de Tappy à l'œuvre :

```
$( ".myBtn" ).bind( "tap", function(){
  alert( "tap!" );
});
```

En coulisse, cet événement `tap` détecte les interactions tactiles, au clavier ou à la souris pour appliquer un comportement spécifique. (Dans ce cas, il exécute une alerte qui dit « tap ! ». Je suis sûr que vous lui trouverez de meilleures utilisations.)

Pour une bibliothèque qui offre un ensemble de fonctions tactiles plus avancées, jetez un œil à FastClick (http://bkaprt.com/rrd/1-16/), créée et entretenue par la talentueuse équipe du *Financial Times*.

CONCEVOIR POUR L'ACCESSIBILITÉ

Nous avons abordé quelques aspects importants en matière d'utilisabilité, comme concevoir pour différentes variantes d'écrans, trouver des points de rupture et gérer les modes de saisie de manière inclusive. Mais pour que les composants soient utilisables sur tous les appareils, nous devons faire en

sorte qu'ils soient accessibles dans les navigateurs qui ne prennent pas en charge notre présentation ou notre comportement idéal, et pour les utilisateurs qui parcourent le Web à l'aide de technologies d'accessibilité. Pour ces raisons et bien d'autres, vous ne pourriez pas rendre de meilleur service à vos utilisateurs qu'en commençant par une bonne vieille base de HTML. L'un des points forts du HTML, c'est sa rétrocompatibilité inhérente, grâce à laquelle toutes les pages, même développées avec la toute dernière itération, sont accessibles à partir de pratiquement n'importe quel appareil qui supporte l'HTML.

Si les documents HTML sont particulièrement accessibles à la base, ils ne le restent pas toujours : une application imprudente de CSS et de JavaScript peut rendre un contenu complètement inutilisable, et les utilisateurs préféreront encore l'expérience dépouillée initiale. Par exemple, songez à un menu déroulant dont le contenu est masqué à l'aide de `display: none;`. Sauf exception, les lecteurs d'écran relaieront uniquement le contenu présenté à l'écran ; par conséquent, si vous ne prenez pas de précautions, le contenu de ce menu sera non seulement masqué visuellement, mais également auditivement. Nous devrons alors donner des indices pour alerter tous les utilisateurs - pas seulement ceux qui parcourent le Web visuellement - que le contenu du menu existe et peut être affiché (ou lu) au besoin.

À mesure que nous poussons HTML vers de nouvelles interactions, il est essentiel de considérer que l'accessibilité risque à tout moment d'être perdue, et que nous devons la préserver tout au long du processus de développement.

Garantir l'accès grâce à l'amélioration progressive

L'idée que le Web est né accessible va de pair avec le concept d'amélioration progressive, qui consiste à commencer par une base de HTML fonctionnel et sémantique, puis à ajouter une discrète couche de présentation (CSS) et de comportement (JS) pour une expérience utilisateur plus riche et plus dynamique.

Un grand pouvoir implique de grandes responsabilités. Chaque fois que vous vous aventurez au-delà d'un simple rendu de code HTML pour concevoir votre présentation et votre interactivité propres, vous êtes responsable de l'accessibilité. Cela demande une once de planification. En tant que développeurs,

FIG 1.14 : Vue des contrôles natifs sous-jacents (à gauche) derrière une interface utilisateur améliorée (à droite)

nous devons être capables de « voir au travers » de nos designs visuels pour découvrir leur signification sous-jacente en HTML.

Dans le livre *Designing with Progressive Enhancement* du Filament Group, nous appelons ce processus « perspective aux rayons X » (**FIG 1.14**) :

> *La perspective aux rayons X est une méthodologie que nous avons développée pour évaluer un design de site complexe, le découper en modules élémentaires et le reconstruire de sorte qu'une seule page de code fonctionne aussi bien sur les navigateurs pleinement fonctionnels que sur les navigateurs et les appareils qui ne comprennent que l'HTML de base.*

Passer les différentes parties d'un design aux rayons X peut demander un certain degré de réflexion créative : tout dépend du niveau de similitude entre le contrôle personnalisé et son équivalent natif. Certains sont plutôt transparents : par exemple un bouton qui fait office d'`input` de type case à cocher. Dans ce cas, il suffira d'un peu de CSS pour produire un bouton comme illustré à la **FIG 1.15**, à l'aide des balises `label` et `input`.

FIG 1.15 : Un élément input et label standard stylé comme un bouton

```
<label class="check">
  <input type="checkbox">Bold
</label>
```

Une approche utilisant uniquement CSS présente trois avantages. Elle est simple, légère et surtout, elle utilise des éléments de formulaire HTML natifs qui garantissent que le contrôle est accessible aux utilisateurs de dispositifs d'accessibilité. En d'autres termes, les technologies d'assistance comme le lecteur d'écran intégré VoiceOver d'Apple liront le contrôle natif à haute voix comme si les améliorations visuelles n'existaient pas : « bold, case non cochée » par défaut et « bold, case cochée » lorsque la case est cochée.

Facile, non ? Cependant, il peut être difficile de préserver ce niveau d'accessibilité avec des composants personnalisés plus complexes.

Amélioration responsable d'un contrôle complexe

Portons nos rayons X sur quelque chose de plus abstrait, comme un curseur (FIG 1.16).

La spécification HTML5 apporte un lot de nouveaux types de champs de formulaires comme `number`, `color` et `search`. Vous pouvez utiliser ces types dès aujourd'hui en toute sécurité pour offrir une interactivité plus spécialisée aux navigateurs qui les prennent en charge ; les navigateurs qui ne les comprennent pas rendront simplement les données comme si elles étaient de type `text` standard.

Voici un exemple de balisage pour un champ de type `color` :

```
<label for="color">Choose a color:</label>
<input type="color" id="color">
```

FIG 1.16 : Un curseur personnalisé avec un champ de saisie numérique

FIG 1.17 : Un champ de saisie de couleur dans Google Chrome

La **FIG 1.17** en illustre le rendu dans Google Chrome, un navigateur compatible et la **FIG 1.18** dans iOS 7, un système non compatible.

Un autre de ces nouveaux types de champs est `range`, qui affiche un curseur dans la plupart des navigateurs. Mais le curseur généré de manière native laisse à désirer du point de vue du design et de l'utilisabilité. Pour commencer, son apparence est délicate - parfois impossible - à personnaliser. Sur certains navigateurs, le curseur natif ne comporte pas de légende pour afficher la valeur sélectionnée, ce qui le rend inutile pour choisir des valeurs précises. Par exemple, la **FIG 1.19** illustre le rendu d'un champ `range` avec une plage de valeurs possibles de 1 à 10 sous iOS 7 Safari.

```
<label for="value">Choose a value:</label>
<input type="range" id="value" min="0" max="10">
```

FIG 1.18 : Le champ color est rendu par défaut en un champ de saisie de texte sous iOS 7.

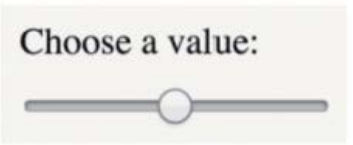


FIG 1.19 : Un champ range rendu sous iOS 7 Safari, qui n'affiche ni le minimum, ni le maximum, ni la valeur actuelle.

À moins que nous ne concevions un contrôle de volume, ce curseur ne nous sera pas d'une grande utilité. Si nous voulons créer un curseur utilisable et agréable à utiliser sur les écrans tactiles, nous devrons le construire nous-mêmes. Faisons-le d'une façon qui fonctionne pour tout le monde.

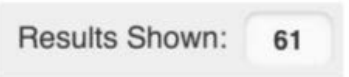


FIG 1.20 : Notre balisage HTML de base rendu dans un navigateur

La première étape - et la plus importante - consiste à commencer avec notre bon vieil ami, HTML. En fin de compte, un curseur est la visualisation d'une échelle numérique, alors commençons par une balise `input` de type `number`, un autre type de champ HTML5 qui sera interprété comme un champ `text` par les navigateurs non compatibles. Le type de champ `number` a l'avantage de nous permettre d'utiliser plusieurs attributs standard complémentaires qui modèlent les contraintes du contrôle : `min` et `max`. Nous utiliserons ces attributs comme point de départ de notre HTML (FIG 1.20) :

```
<label for="results">Results Shown:</label>
<input type="number" id="results" name="results" »
  value="60" min="0" max="100" />
```

Maintenant que nous avons nos fondations, nous pouvons utiliser JavaScript pour créer un curseur qui manipulera la valeur du champ `input` lorsque l'utilisateur actionnera la manette.

Le script à écrire dépasse le cadre de ce livre, mais j'aborderai quand même le balisage généré et comment s'assurer que le curseur ne pose pas de problème d'accessibilité. Pour commencer, la nouvelle balise générée en gras :

```
<label for="results">Results Shown:</label>
<input type="number" id="results" name="results" »
  value="60" min="0" max="100" />
<div class="slider">
  <a href="#" class="handle" style="left: 60%;"></a>
</div>
```

Détaillons les changements à apporter. Pour créer la « poignée » et la « glissière » de notre curseur, nous devons utiliser un élément qui est sélectionnable par un clavier, en l'occurrence un élément `a`, auquel j'ai attribué la classe `handle` à titre

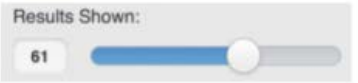


FIG 1.21 : Notre div de curseur, avec l'élément input affiché à gauche

de référence. Nous avons également besoin d'un conteneur `div` pour que la classe `.handle` soit stylée visuellement comme une glissière. Quand l'utilisateur déplace la poignée ou utilise les flèches de son clavier, nous utilisons JavaScript pour manipuler le positionnement `left` de la poignée en CSS avec un pourcentage qui reflète la distance que l'utilisateur a parcourue, et mettre à jour la valeur de notre contrôle `input` par la même occasion. J'ai inclus le balisage de notre nouveau curseur en gras (FIG 1.21) :

```
<label for="results">Results Shown:</label>
<input type="number" id="results" name="results" »
  value="61" min="0" max="100" />
<div class="slider">
  <a href="#" class="handle" style="left: 61%;"></a>
</div>
```

Styles CSS à part, voilà l'essentiel du comportement d'un curseur basique. Mais notre travail n'est pas fini. Notre page était accessible au départ, mais avec JavaScript, nous avons introduit une balise qui joue un rôle anormal – un élément `a` avec la classe `.handle`. Lorsqu'un lecteur d'écran rencontrera cet élément, il le lira à voix haute comme un « lien numérique » parce qu'il lui semblera être un lien ordinaire avec une valeur `href` de `#`.

Pour éviter que cette balise induise l'utilisateur en erreur, nous avons deux options : nous pouvons soit cacher le curseur pour les lecteurs d'écran (puisqu'il existe déjà un champ de saisie textuel `input`) ou trouver une solution pour que le curseur lui-même ait un sens pour les lecteurs d'écran. Pour ma part, je préfère masquer le nouveau contrôle ; il suffit d'ajouter l'attribut `aria-hidden` dans le bloc `div`, qui instruit aux lecteurs d'écran d'ignorer le contenu de cet élément lorsque la page est lue à haute voix :

```
<label for="results">Results Shown:</label>
<input type="range" id="results" name="results" »
  value="61" min="0" max="100" />
<div class="slider" aria-hidden>
  <a href="#" style="left: 61%;"></a>
</div>
```

Et aussi simplement que ça, nous avons appliqué une amélioration progressive à notre champ pour offrir une meilleure présentation visuelle sans compromettre l'accessibilité. « Mais... ARIA quoi ? », vous demandez-vous peut-être. Pour faire court, la spécification ARIA (Accessible Rich Internet Applications) du W3C est un ensemble d'attributs HTML qui permettent de changer la définition sémantique d'éléments HTML qui jouent un rôle non natif - qu'il s'agisse d'un `a` faisant office de bouton de menu (qui emploiera l'attribut `role="button"` d'ARIA) ou d'un `ul` servant de composant d'arborescence navigable (l'attribut `role="tree"`), comme si vous parcouriez une liste de fichiers dans l'explorateur d'un système d'exploitation. Il existe même un rôle ARIA pour décrire un curseur, si nous voulions utiliser cette méthode avec notre exemple précédent : `role="slider"`. En plus de ces attributs de rôle, ARIA comporte des attributs d'état qui permettent de décrire l'état du contrôle, tels que `aria-expanded`, `aria-collapsed` et `aria-hidden` (utilisé ci-dessus), et même des attributs pour décrire la valeur actuelle et les valeurs possibles d'un curseur personnalisé. Pour explorer les possibilités d'ARIA, consultez le site du W3C (http://bkaprt.com/rrd/1-17/).

Visualisations de données accessibles

Les visualisations de données telles que les tableaux et les graphiques sont souvent données sous un format qui n'a pas beaucoup de sens pour les personnes qui utilisent des technologies d'accessibilité. Prenons par exemple un graphique complexe tiré d'un article du *New York Times* et rendu via une balise `img` (FIG 1.22).

Pour un lecteur d'écran, toutes les informations contenues dans ce graphique sont invisibles. Un développeur responsable pourrait (ce serait la moindre des choses !) ajouter un attribut

FIG 1.22 : Ce graphique à courbes complexe a été rendu via une balise `img` (http://bkaprt.com/rrd/1-18/).

`alt` pour décrire les données du graphique, mais ces données sont parfois impossibles à décrire convenablement dans une seule chaîne de texte.

```
<img src="chart.png" alt="Economic winners and losers, »
  Change since...">
```

Comment pouvons-nous mieux communiquer ces informations ? Sortez les rayons X. Comme nous l'avons fait avec le curseur, nous pouvons choisir un point de départ plus sémantique à partir duquel créer ce graphique. Prenez par exemple le diagramme en camembert à la **FIG 1.23**. Comment pouvons-nous le construire d'une façon plus porteuse de sens pour les lecteurs d'écran qu'une balise `img` ?

Nous pouvons commencer par rédiger une base de HTML sémantique à l'intention de tous les utilisateurs et présenter le

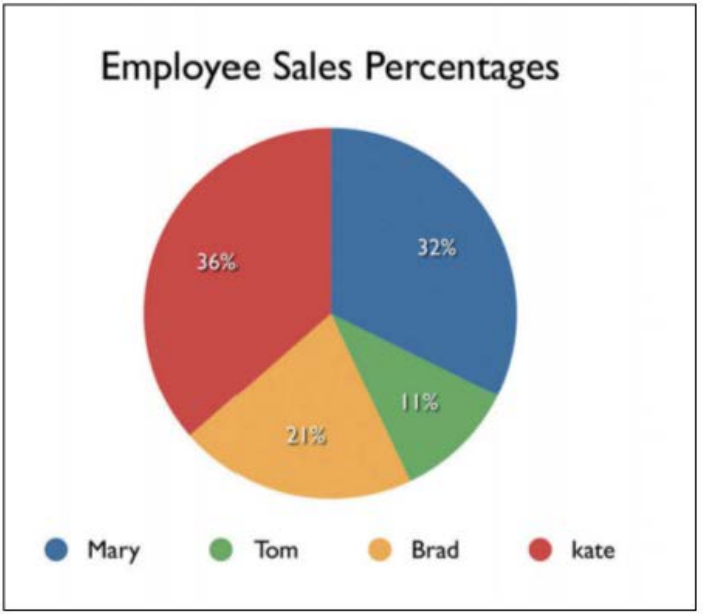


FIG 1.23 : Comment transmettre le sens de graphiques complexes à des lecteurs d'écran ?

graphique comme une amélioration. En étudiant le graphique pour en déterminer le message sous-jacent, nous pourrions par exemple déterminer qu'il est possible de le décrire sous la forme d'un tableau HTML. Nous pourrions alors introduire le balisage HTML ci-dessous dans une fonction JavaScript pour dessiner le tableau de façon dynamique avec une technologie telle que `canvas` d'HTML5 ou SVG. Une fois le graphique généré, nous pourrions même choisir de masquer le tableau en le plaçant en dehors de l'écran, considérant le graphique comme une amélioration visuelle du tableau qu'il remplace (FIG 1.24).

```
<table>
  <summary>Employee Sales Percentages</summary>
  <tr>
```

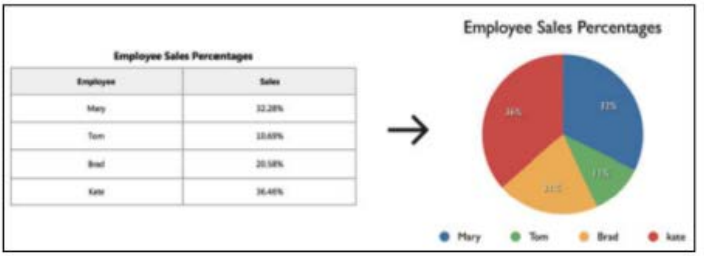


FIG 1.24 : Visualisation du tableau de gauche générée à l'aide de canvas

```
      <th>Employee</th>
      <th>Sales</th>
    </tr>
    <tr>
      <td>Mary</td>
      <td>32.28%</td>
    </tr>
    <tr>
      <td>Tom</td>
      <td>10.69%</td>
    </tr>
    <tr>
      <td>Brad</td>
      <td>20.58%</td>
    </tr>
    <tr>
      <td>Kate</td>
      <td>36.46%</td>
    </tr>
  </table>
```

Nous n'avons vu qu'un aperçu de tous les facteurs à prendre en compte pour développer des interfaces complexes et accessibles. Mais de manière générale, vous avez tout intérêt à composer une base de HTML sémantique qui sera valide, accessible et fonctionnelle sur pratiquement sur tous les appareils, puis ensuite seulement à ajouter des couches d'amélioration.

Une amélioration peut vite devenir une gêne, et c'est à nous, développeurs responsables, de nous assurer que cela ne se produise pas.

Il est clairement bénéfique pour l'accessibilité de concevoir de cette façon, mais la planification de cette approche peut représenter un défi lorsqu'il s'agit de communiquer ces attentes à nos clients et à nos testeurs d'assurance qualité. Il est peut-être nécessaire d'ajuster légèrement notre façon de définir le support...

Stratégie de support améliorée

Dans l'article « Grade Components, Not Browsers », j'ai repris une brillante idée de ma collègue Maggie Wachs, qui définit le support de manière modulaire pour chaque composant du site (plutôt que d'attribuer une note à un navigateur dans son ensemble, comme il est courant de le faire avec des approches telles que le système Graded Browser Support de Yahoo) (http://bkaprt.com/rrd/1-19/). La documentation que nous partageons avec nos clients classifie chaque composant en fonction de ses principaux degrés d'amélioration.

À titre d'exemple, l'image suivante présente les différents degrés d'amélioration pour la page de détails d'un bien immobilier (FIG 1.25). Le degré d'amélioration qu'un navigateur recevra dépendra de plusieurs conditions, notamment du support de fonctionnalités telles qu'Ajax et les transformations 3D de CSS3.

Cette documentation accomplit plusieurs choses. Tout d'abord, elle nous permet de détailler à nos clients les conditions particulières qui permettent à certaines portions de leur site de fonctionner de manière enrichie, de sorte que tout le monde (designers, clients et testeurs d'assurance qualité) sache à quoi s'attendre. Elle sert également à rappeler que certains composants peuvent recevoir une note plus élevée que d'autres selon le navigateur utilisé. En d'autres termes, le support des fonctionnalités est variable même sur les navigateurs modernes, et il est possible qu'un navigateur reçoive une expérience enrichie notée A pour un certain composant et une expérience moins riche notée B pour un autre composant.

FIG 1.25 : Documentation des différents degrés d'amélioration d'une fonctionnalité dont la présentation varie selon le navigateur

En documentant le support de cette façon, nous nous intéressons moins au navigateur qu'à ses fonctionnalités et à ses contraintes. Nous commençons à moins considérer le support comme une fonction binaire, ou même une échelle, et plus comme un diagramme de dispersion. Dans ce système, n'importe quel navigateur qui comprend l'HTML est pris en charge et peut accéder aux fonctionnalités et au contenu principal du site. Comme le fait remarquer Jeremy Keith : « C'est à nous qu'il revient d'expliquer comme le Web fonctionne… et pourquoi la nature inégalement distribuée des capacités des navigateurs n'est pas un bug, mais une fonctionnalité » (http://bkaprt.com/rrd/1-20/).

En parlant de fonctionnalités, nous avons besoin de moyens fiables et durables de détecter celles-ci et ce, quel que soit l'appareil. Nous allons maintenant voir pourquoi et comment faire.

2 DÉTECTION DURABLE

Au vu de la nature diverse des navigateurs actuels, la capacité à détecter leurs fonctionnalités et leurs contraintes est essentielle pour offrir une expérience utilisateur appropriée. Nous avons de nombreux moyens d'aborder cette détection, et certains sont plus responsables que d'autres.

DÉTECTION D'APPAREIL : L'ÉVOLUTION D'UN PALLIATIF

Parmi les sujets de débat dans le domaine du développement web, le plus controversé reste probablement la pratique de la détection d'appareil. Sa simple évocation lors d'une réunion entre collègues me donne des frissons à l'idée du déferlement d'opinions qui nous attend. En vérité, la détection d'appareil est parfois nécessaire pour certains codebases complexes et multi-appareils, mais à chaque nouveau projet que j'aborde, je trouve de moins en moins de raisons de l'utiliser.

C'est une très bonne chose, car toute approche qui inclut des logiques spécifiques pour différents appareils est une menace

pour la durabilité de notre codebase à long terme. Voyons maintenant pourquoi.

À tout vouloir détecter...

Lorsqu'un utilisateur demande à accéder à une page pour la première fois, nous savons bien peu de choses sur son environnement de navigation. Nous ne connaissons pas la taille de l'écran de son appareil, et il se peut même qu'il n'ait pas d'écran du tout. Nous ne connaissons pas les capacités de son navigateur. Nous pouvons détecter ces caractéristiques après avoir transmis notre code au navigateur, mais dans certains cas, c'est déjà trop tard.

S'il y a une chose que nous pouvons détecter de façon universelle lors de la première visite, c'est le user-agent du navigateur, qui est inclus dans chaque requête qu'un navigateur - ou *user-agent* - envoie au serveur. Cette chaîne de caractères comprend diverses informations, notamment le nom et la version du navigateur, comme Firefox 14 ou Chrome 25, ainsi que son système d'exploitation, comme Apple iOS. Des développeurs ingénieux ont rapidement compris qu'en recueillant des données sur divers navigateurs et leurs capacités et en les enregistrant sur leur serveur (dans ce que l'on appelle une base de données d'appareils), ils pouvaient demander ces informations lorsqu'un utilisateur visitait leur site pour se faire une bonne idée du type de navigateur utilisé. Ce processus est appelé *user-agent sniffing*, ou plus généralement détection d'appareil.

... on fait fausse route

La critique la plus courante formulée à l'encontre de la détection du user-agent, c'est que les informations fournies par le navigateur ne sont pas toujours fiables. Les navigateurs, les réseaux et parfois les utilisateurs eux-mêmes modifient les informations du user-agent pour une multitude de raisons, et il est donc difficile de savoir avec certitude à quel navigateur vous vous adressez. Commençons par les préférences de plusieurs navigateurs mobiles répandus : le navigateur par défaut d'Android, Opera Mini, le navigateur BlackBerry et bien d'autres offrent un moyen de changer le nom sous lequel le navigateur se présente. Vous verrez parfois cette option déguisée sous la

FIG 2.1 : Réglage du user-agent dans le navigateur d'Android, Opera et Firefox

mention « Demander le site de bureau » ou avec des réglages plus détaillés comme dans le navigateur Android. En modifiant leur user-agent, les utilisateurs peuvent contourner les limitations de certains sites qui fournissent des fonctionnalités et un contenu différents selon le navigateur (FIG 2.1).

De même, la chaîne user-agent par défaut de certains navigateurs mentionne de nombreux autres navigateurs afin que ses utilisateurs ne reçoivent pas une version limitée de certains sites. Par exemple, en plus de quelques informations appropriées, le user-agent de mon navigateur actuel (Chrome 34) mentionne Mozilla, Webkit, KHTML, Gecko et Safari - des navigateurs qui ne sont pas basés sur l'architecture de Chrome :

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_5)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/34.0.1847.131 Safari/537.36
```

Certains navigateurs vont encore plus loin et dissimulent délibérément les informations du user-agent pour forcer les sites à leur fournir l'expérience que reçoivent d'autres navigateurs. La chaîne user-agent d'Internet Explorer 11, une version pourtant grandement améliorée, ne fait aucune mention d'Internet Explorer ! Au lieu de ça, elle tente de faire croire aux bibliothèques de détection d'appareil qu'il s'agit de Firefox ou de Webkit, que les développeurs reconnaissent aujourd'hui comme les seuls navigateurs qui prennent en charge les fonctionnalités avancées nécessaires pour offrir une expérience enrichie. (Avec les versions plus récentes d'IE, ce n'est heureusement plus le cas.) Dans un article publié sur *A List Apart* et

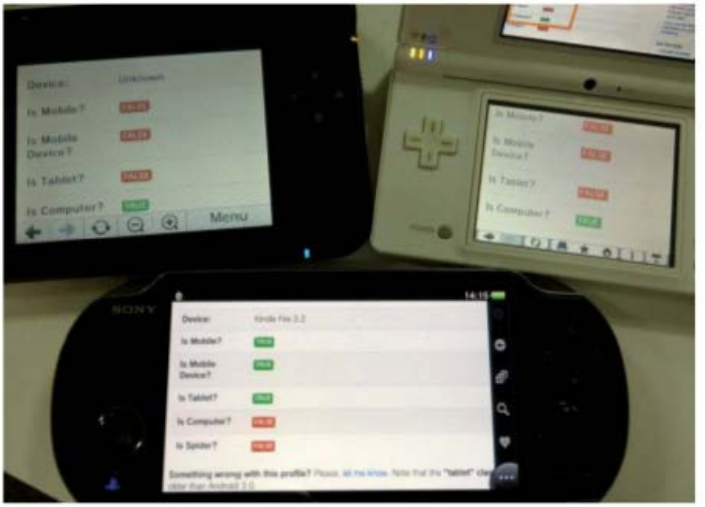


FIG 2.2 : Résultats de détection d'appareil inadéquats sur un appareil mobile (http://bkaprt.com/rrd/2-02/)

intitulé « Testing Websites in Game Console Browsers », Anna Debenham note une situation similaire avec le navigateur de la PlayStation Vita de Sony : « Le navigateur de la Vita est une version de NetFront basée sur WebKit. Étrangement, son user-agent l'identifie comme Silk, qui est le navigateur du Kindle Fire d'Amazon. » (http://bkaprt.com/rrd/2-01/) (FIG 2.2)

Les développeurs de navigateurs ont tout intérêt à garantir la pérennité de leur logiciel. Ironiquement, plus les développeurs Web chercheront à fournir des fonctionnalités et du contenu différents selon le user-agent, moins les informations de ces user-agents seront fiables.

Une solution peu durable

Mais le manque de fiabilité de cette approche est un problème mineur comparé à son manque de viabilité. Nous pouvons uniquement écrire des logiques de détection pour les navigateurs et les appareils qui existent aujourd'hui, ce qui rend la détection

FIG 2.3 : Suppositions à ne pas faire en se basant sur le user-agent d'un utilisateur

d'appareil complètement inutile pour prendre en charge les navigateurs à venir.

Mais surtout, en nous fiant trop à la détection d'appareil, nous risquons de formuler des hypothèses dangereuses basées sur des informations qui ne sont pas toujours à jour. La détection d'appareil permet au mieux d'obtenir des informations génériques sur un appareil ou un navigateur, et toutes les optimisations que nous effectuerons en nous basant sur ces données statiques risquent de ne pas refléter la nature vivante et dynamique du véritable environnement de navigation d'un utilisateur.

Voici quelques exemples de variables qu'une base de données d'appareils ne pourra jamais indiquer précisément (FIG 2.3) :

- **Taille du viewport**. Une base de données d'appareils peut apporter des informations relativement fiables sur l'écran d'un appareil, mais la taille de l'écran diffère souvent de la taille du viewport du navigateur. Pour des mises en page responsive, c'est la taille du viewport qui nous intéresse. Nous ne devons pas non plus essayer de déduire la vitesse de la connexion de la taille de l'écran – les smartphones sont couramment utilisés sur des connexions wi-fi rapides, tandis

FIG 2.4 : Site web du Boston Globe vu sur le même appareil dans deux orientations différentes

que les ordinateurs portables et les tablettes peuvent être connectés via un réseau cellulaire lent.

- **Orientation de l'appareil**. Ces considérations sur la taille du viewport se compliquent encore plus lorsque vous voulez prendre en compte les différences d'affichage entre les orientations portrait et paysage (FIG 2.4). Même si nous connaissons les dimensions d'un écran, nous n'avons aucun moyen de savoir (du côté serveur) quelle est l'orientation de l'appareil. Nous devons fournir une feuille de style qui s'adapte à cette variabilité du viewport.
- **Taille de la police**. La pratique courante consistant à utiliser des unités basées sur le cadratin (em) pour les media queries signifie que c'est la taille de police par défaut de l'utilisateur qui détermine la mise en page qu'il reçoit. Ainsi, le navigateur d'un ordinateur portable avec une police par défaut plus grande devra peut-être recevoir la page pour smartphone. (Comme nous le verrons par la suite, les media queries CSS gèrent ce problème naturellement.)
- **Préférences personnalisées**. Il arrive couramment que les utilisateurs modifient les paramètres par défaut de leur navigateur et désactivent des fonctionnalités sur leur téléphone.

FIG 2.5 : Un appareil Android 2.3 avec plusieurs mécanismes de saisie

Bien qu'un navigateur prenne une fonctionnalité en charge, le serveur n'a aucun moyen de savoir si cette fonctionnalité a été désactivée par l'utilisateur.

- **Modes de saisie**. Une base de données d'appareils peut généralement nous dire si un appareil est doté d'un écran tactile. Mais comme vous le savez, ce n'est pas parce qu'un appareil est équipé d'un écran tactile qu'il prend en charge les événements tactiles, ou que le tactile est le seul mécanisme de saisie qu'offre l'appareil (FIG 2.5). Et c'est sans compter que les interactions tactiles sont désormais intégrées dans des appareils dotés de grands écrans, comme l'ordinateur portable Chromebook de Google ; il est donc impossible d'établir une quelconque corrélation entre la prise en charge des interactions tactiles et la taille de l'écran.

Ainsi, lorsque nous construisons des expériences multi-appareils, nous devons être attentifs à ces facteurs et nous garder de toute supposition basée sur les conditions d'usine des appareils. La détection d'appareil est un pari risqué, et elle va le devenir de plus en plus.

BONNE NOUVELLE : NOUS AVONS LE CONTRÔLE

L'uniformisation du code entre les différents navigateurs a été longue, mais le support et les outils permettant de prendre des décisions durables et basées sur les fonctionnalités et les contraintes se sont radicalement améliorés au cours des dernières années, et ils ne cessent de progresser de jour en jour. Des technologies côté client telles que HTML, CSS et JavaScript nous permettent de voir ce qui se passe vraiment dans cet environnement de navigation dynamique et de prendre des décisions plus contextuelles et plus appropriées. En un mot : plus responsables.

Pensez fonctionnalités et contraintes, pas appareils

> *À trop mettre l'accent sur le contexte, on risque de produire des solutions de design qui répondent à des situations mobiles présumées au lieu de se focaliser sur la véritable richesse de l'usage mobile du Web tel qu'il est aujourd'hui.*
> Luke Wroblewski, http://bkaprt.com/rrd/2-03/

Nous devons cesser de supposer que le facteur de forme de l'appareil est un indicateur des fonctionnalités de son navigateur ou des conditions du réseau auquel il est connecté. En réalité, ces caractéristiques se chevauchent couramment entre différentes catégories d'appareils courants.

> *La présence d'un écran tactile n'a aucun lien avec la taille du viewport. Il est peut-être vrai que les appareils tactiles les plus répandus sont actuellement des téléphones et des tablettes, mais on trouve également des moniteurs tactiles de 27 pouces et même plus.*
> Trent Walton, « Type & Touch » (http://bkaprt.com/rrd/2-04/)

Les catégories mobile et bureau, si elles ont pu être utiles par le passé, ont perdu tout leur sens aujourd'hui. On utilise parfois « mobile » pour décrire le contexte physique itinérant d'un appareil, mais les gens utilisent couramment leurs smartphones et leurs tablettes à la maison, depuis leur canapé. Nous

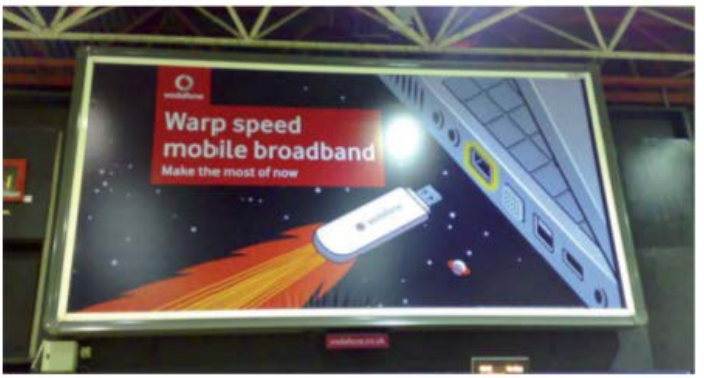


FIG 2.6 : Wi-fi non garanti : publicité pour une clé USB qui permet d'accéder au Web par le biais du réseau cellulaire à l'aide d'une carte SIM. Photographie de Frankie Roberto (http://bkaprt.com/rrd/2-05/).

considérons peut-être que le Web mobile présente des contraintes de vitesse de connexion, mais des appareils de toutes sortes sont aussi susceptibles d'être connectés à un wi-fi haut débit qu'à une antenne-relais à forte latence (FIG 2.6). On peut partir du principe que les mobiles sont des appareils présentant des caractéristiques particulières, telles qu'un écran plus petit qui supporte les interactions tactiles, ou des contraintes telles que de piètres capacités de rendu, mais chaque jour sortent de nouveaux appareils qui échappent à toutes les classifications que nous essayons de leur imposer.

En essayant de catégoriser les appareils et les navigateurs par leur seul facteur de forme, nous en oublions les paramètres véritablement importants pour concevoir pour le Web : les fonctionnalités (comme les propriétés CSS et les API JavaScript) et les contraintes (comme la taille du viewport, l'imprévisibilité de la connexion ou l'utilisation hors ligne). En concevant pour ces fonctionnalités et ces contraintes, nous nous apercevrons que des modèles que nous pensions distincts sont en fait partagés par plusieurs types d'appareils, et nous pourrons construire de façon modulaire afin de créer des expériences uniques qui s'adaptent à chaque appareil.

Media queries responsables

Le principe le plus mémorable du workflow de responsive design original d'Ethan Marcotte est peut-être l'utilisation des media queries de CSS3, ces déclarations conditionnelles que nous utilisons pour fournir des styles dans certains contextes et pas dans d'autres. L'article initial de Marcotte utilisait les media queries dans une approche *desktop first*, ce qui signifie qu'il concevait d'abord la mise en page la plus grande et utilisait les media queries pour l'adapter progressivement aux plus petits écrans.

Renverser la vapeur

Vers la fin de son livre *Responsive web design*, Marcotte fait remarquer qu'en inversant notre approche des media queries pour suivre une philosophie *mobile first*, c'est-à-dire axée en premier lieu sur les petits écrans, nous pourrions offrir une expérience plus responsable et durable à nos utilisateurs. Pour paraphraser Luke Wroblewski, un workflow *mobile first* nous aide à prioriser le contenu, comme il n'y a pas assez de place sur un petit écran pour le contenu superflu. Une approche *mobile first* va également de pair avec le concept d'amélioration progressive, qui consiste à partir du strict minimum et à ajouter des couches de mise en page plus complexes en fonction de l'espace disponible.

> *L'absence de prise de charge des media queries est en fait la première media query.*
> Bryan Rieger, http://bkaprt.com/rrd/2-06/

Une feuille de style adaptative *mobile first* commence par les styles qui sont partagés par toutes les expériences, formant la fondation de la mise en page pour les plus petits écrans. Ces styles sont suivis d'une série de media queries utilisant essentiellement `min-width` afin de redimensionner cette mise en page pour les viewports plus grands et les résolutions plus élevées. À un niveau global, la CSS ressemble à ceci :

```
/* styles pour les petits viewports ici */
.logo {
```

```
  width: 50%;
  float: left;
}
.nav {
  width: 50%;
  float: right;
}

@media (min-width: 50em) {
  /* styles pour les viewports de largeur égale ou
  supérieure à 50em */
}

@media (min-width: 65em) {
  /* styles pour les viewports de 65em et plus */
}
```

Et max-width dans tout ça ?

Même si vous suivez une approche *mobile first*, les requêtes `max-width` peuvent être utiles. Par exemple, si une variante du design doit se produire uniquement dans une certaine plage de largeur, `max-width` peut vous venir en aide. Vous pouvez combiner `min-width` et `max-width` pour isoler des styles CSS et éviter qu'ils ne soient hérités par les points de rupture supérieurs, de sorte que votre CSS soit plus simple et plus compacte :

```
@media (min-width: 50em) {
  .header {
    position: static;
  }
}

@media (min-width: 54em) and (max-width: 65em) {
  .header {
    position: relative;
  }
}
```

```
@media (min-width: 65em) {
  /* L'en-tête est statique placé ici */
}
```

Et pourquoi tous ces em, au fait ?

Vous avez peut-être remarqué qu'en plus d'inverser la direction du design adaptatif, les points de rupture ci-dessus utilisent le cadratin (em) plutôt que des pixels. Le cadratin est une unité flexible qui est dimensionnée en fonction du contenant d'un élément dans la mise en page. En utilisant le cadratin, nous pouvons concevoir des points de rupture adaptatifs de manière proportionnelle à notre contenu fluide et redimensionnable qui est généralement lui aussi conçu avec des unités redimensionnables telles que `em` et `%`.

Il est simple de convertir en cadratins vos points de rupture en pixels : divisez la valeur en pixels par 16, la taille par défaut équivalant à `1em` dans la plupart des navigateurs Web :

```
@media (min-width: 800px){
...
}
@media (min-width: 50em){ /* 800px / 16px */
...
}
```

Si les points de rupture en cadratins ne sont pas votre tasse de thé, les pixels peuvent être tout à fait convenables - je préfère simplement utiliser des unités proportionnelles dans mes mises en page. Le plus important, c'est d'éviter de baser vos points de rupture sur la largeur des appareils et de vous focaliser plutôt sur des points de rupture qui sont adaptés au contenu de votre site. Pour plus d'informations sur les em dans les media queries, consultez l'article de Lyza Gardner intitulé « The EMs have it: Proportional Media Queries FTW! » (http://bkaprt.com/rrd/2-07/).

Réserver CSS3 aux navigateurs compatibles

Tous les navigateurs mobiles ne prennent pas en charge le CSS sur lequel nous comptons pour rendre le design de notre site, comme les *floats*, les positions ou les animations. Si vos styles pour une expérience petit écran sont relativement complexes, vous pouvez envisager de réserver leur application aux navigateurs plus récents qui prennent en charge les media queries. Pour cela, un moyen fiable consiste à intégrer vos styles *mobile first* dans une media query `only all`. Bien qu'elle puisse sembler un peu curieuse à première vue, la requête `only all` s'applique à tous les navigateurs qui prennent en charge les media queries CSS3. Si `all` est un type de média CSS qui s'adresse à tous les navigateurs prenant en charge CSS 1.0, le préfixe `only` n'est compris que par les navigateurs qui prennent en charge les media queries CSS3 – ce qui signifie que les styles définis dans ce bloc seront uniquement reconnus par les navigateurs modernes. Voici à quoi ressemblera notre feuille de style *mobile first* réservée aux navigateurs qui prennent en charge les media queries :

```
@media only all {
  /* styles pour les petits viewports qualifiés ici */
}

@media (min-width: 50em) {
  /* styles pour les viewports de 50em et plus */
}

@media (min-width: 65em) {
  /* styles pour les viewports de 65em et plus */
}
```

Appliquer un minimum de style aux navigateurs basiques

Pour préserver une certaine expérience de marque dans les navigateurs qui ne prennent pas en charge les media queries, je trouve utile de déterminer quels sont les styles les plus sûrs de votre premier point de rupture CSS et de les placer avant la media query `only all` afin qu'ils s'appliquent partout.

FIG 2.7 : Interface basique du site web du Boston Globe sur un vieux modèle de BlackBerry

Les styles sûrs - comme `font-weight`, `margin`, `padding`, `border`, `line-height`, `text-align` et d'autres - peuvent être envoyés à n'importe quel navigateur sans poser de problèmes (**FIG 2.7**).

```
/* styles pour les petits viewports ici */
body {
  font-family: sans-serif;
  margin: 0;
}
a {
  font-color: #a00;
}
section {
  margin: 1em;
  border-bottom: 1px solid #aaa;
}
```

```
@media only all {
  /* styles pour les petits viewports qualifiés ici */
}
/* suite... */
```

Un petit rappel rapide (et responsable) : si vous avez l'intention de fournir des styles aux navigateurs basiques, assurez-vous de les tester !

Blinder le viewport

Traditionnellement (si tant est que l'on puisse utiliser ce mot), dans les mises en page responsive, nous utilisons une balise `meta` pour spécifier la largeur que le navigateur doit utiliser pour rendre une page lors de son premier chargement, telle que la classique déclaration `width=device-width` :

```
<meta name="viewport" content="width=device-width; »
  initial-scale=1">
```

Cette approche nous a bien servi jusqu'à présent, mais elle n'est pas particulièrement durable : pour commencer, le W3C ne l'a jamais standardisée ; par ailleurs, les éléments `meta` ne sont pas l'endroit le plus évident pour définir un style visuel. Par chance, le W3C a standardisé une approche pour spécifier les informations de style du viewport avec `width` et `scale`, et elle est gérée par CSS au lieu d'HTML. Pour nous assurer que nos réglages du viewport continuent à fonctionner dans les futurs navigateurs, nous devons inclure ces règles dans notre CSS :

```
@-webkit-viewport{width:device-width}
@-moz-viewport{width:device-width}
@-ms-viewport{width:device-width}
@-o-viewport{width:device-width}
@viewport{width:device-width}
```

Pour les navigateurs qui ne supportent pas la règle `@viewport`, nous devons continuer d'inclure l'élément `meta viewport`. Trent Walton a écrit un billet pratique à ce sujet qui comprend des astuces pour faire en sorte que nos sites responsive

fonctionnent correctement avec le mode « snap » d'IE10 sous Windows 8 (http://bkaprt.com/rrd/2-08/). (Spoiler qui n'en est pas un : ça demande un peu plus de code que ça.)

Autres media queries

Déterminer la largeur et la hauteur du viewport avec `min-width` et `max-width` suffit à produire une mise en page utilisable, mais il existe de nombreuses autres conditions que nous pouvons tester pour ajouter des améliorations de manière contextuelle. Par exemple, pour fournir des images de plus haute résolution aux écrans HD, nous pouvons utiliser une media query `min-resolution` de `144dpi` (deux fois la résolution standard de `72dpi`). Pour certains navigateurs existants qui n'ont pas encore fait la transition vers la syntaxe standard, nous pouvons également inclure une propriété alternative comportant le préfixe WebKit (`-webkit-min-device-pixel-ratio`) dans notre requête :

```
@media (-webkit-min-device-pixel-ratio: 1.5),
       (min-resolution: 144dpi) {
  /* Styles pour les écrans HD ici */
}
```

Dans un futur proche, les media queries supporteront de nouvelles fonctionnalités intéressantes, qui permettront par exemple de déterminer si les mécanismes de saisie tactile ou les interactions de survol sont supportés à l'aide des règles `@media (pointer:fine) {...}` et `@media (hover) {...}`, de détecter la prise en charge de JavaScript à l'aide de `@media (script) {...}` et même de détecter la luminosité ambiante avec `luminosity`. Pour suivre l'avancement de leur implémentation, gardez un œil sur le site Can I use... (http://bkaprt.com/rrd/2-09/), et pour quelques articles décrivant les bons et les mauvais côtés des media queries de niveau 4, consultez l'excellent article de Stu Cox à ce sujet (http://bkaprt.com/rrd/2-10/).

DÉTECTER DES FONCTIONNALITÉS AVEC JAVASCRIPT

À mesure que de nouvelles fonctionnalités font leur arrivée dans les navigateurs, nous avons souvent besoin de qualifier leur usage de manière plus modulaire. La détection de fonctionnalités JavaScript fait partie depuis longtemps des affres du développement web, en raison des différences entre les fonctionnalités propriétaires des premiers navigateurs. À l'époque, et dans une moindre mesure aujourd'hui, pour que notre code fonctionne dans plusieurs navigateurs, il était nécessaire de vérifier si toutes les fonctionnalités, même les plus courantes, étaient définies avant de les utiliser. Par exemple si nous voulions détecter un événement comme `click`, nous aurions d'abord dû vérifier quelle était l'API d'événements supportée par le navigateur :

```
// si la détection d'événements standard est supportée
if( document.addEventListener ){
  document.addEventListener( "click", myCallback, »
    false );
}
// sinon, essayer la méthode attachEvent d'Internet
// Explorer
else if( document.attachEvent ){
  document.attachEvent( "onclick", myCallback );
}
```

Détecter des fonctions JavaScript

Heureusement, ces dernières années, le mouvement pour la normalisation du Web a poussé la plupart des navigateurs à prendre en charge des API courantes pour certaines fonctions comme la gestion d'événements, réduisant considérablement le nombre d'exceptions à appliquer pour chaque navigateur et rendant notre code plus viable à long terme.

Aujourd'hui, il est devenu plus courant d'utiliser la détection de fonctions JavaScript pour déterminer si une fonction est supportée, avant d'utiliser cette fonction pour ajouter des améliorations par-dessus une expérience HTML déjà fonctionnelle.

Par exemple, la fonction JavaScript suivante détecte le support de l'élément HTML `canvas` (une sorte de plan de travail offrant une API pour dessiner des graphiques avec JavaScript) :

```
function canvasSupported() {
  var elem = document.createElement('canvas');
  return !!(elem.getContext && elem.getContext('2d'));
}
```

Ce code peut être utilisé avant de charger et d'exécuter une pile de code dépendant de `canvas` :

```
if( canvasSupported() ){
// utilisez l'API canvas ici !
}
```

Détecter des fonctionnalités CSS

La détection de fonctions JavaScript est une pratique qui n'a rien de nouveau, mais l'utilisation de JavaScript pour détecter la prise de charge de fonctionnalités CSS a débuté relativement récemment. J'ai utilisé cette approche pour la première fois dans les exemples de mon article « Test-Driven Progressive Enhancement » paru sur *A List Apart* en 2008, qui suggérait de réaliser une série de tests diagnostiques sur un navigateur avant d'appliquer des améliorations en CSS et JavaScript sur la page (FIG 2.8).

À cette époque, les nouveaux navigateurs incluaient de nouvelles fonctionnalités CSS révolutionnaires comme `float` et `position`, mais les navigateurs qui ne les prenaient pas en charge étaient encore largement utilisés. Il était donc difficile d'appliquer du CSS moderne sur un site sans dégrader l'expérience pour les utilisateurs utilisant des navigateurs plus anciens.

Dans l'article, je donnais en exemple le test suivant permettant de vérifier si un navigateur supporte correctement le modèle des boîtes standard de CSS, qui incorpore `padding`, `width` et `border` dans les dimensions mesurées d'un élément. À l'époque, deux variantes du modèle de boîtes étaient activement prises en charge par les navigateurs les plus répandus,

FIG 2.8 : Mon article « Test-Driven Progressive Enhancement » paru en 2008 sur *A List Apart* (http://bkaprt.com/rrd/2-11/)

et une feuille de style rédigée suivant l'un des deux modèles aurait provoqué un dysfonctionnement de la mise en page sur les navigateurs (comprendre : les anciennes versions d'Internet Explorer) qui supportaient l'autre.

```
function boxmodel(){
  var newDiv = document.createElement('div');
  document.body.appendChild(newDiv);
  newDiv.style.width = '20px';
  newDiv.style.padding = '10px';
  var divWidth = newDiv.offsetWidth;
  document.body.removeChild(newDiv);
```

```
    return divWidth === 40;
}
```

Examinons cette fonction plus en détail. La fonction JavaScript crée un nouvel élément `div`, l'appose à l'élément `body` du document, et définit les attributs `width` et `padding` du `div`. La fonction renvoie ensuite une déclaration spécifiant que la largeur rendue du `div` doit être égale à 40 pixels. Si vous savez comment fonctionne le modèle de boîte standard de CSS, vous vous souviendrez que les attributs `width` et `padding` d'un élément contribuent à sa largeur calculée à l'écran ; ainsi, cette fonction détermine si le navigateur calcule cette largeur comme prévu.

Dans l'article, j'ai regroupé ce test et d'autres visant des propriétés telles que `float` ou `position` au sein d'une suite intitulée `enhance.js`, qui peut être exécutée à titre de diagnostic général au cours du chargement de la page. Si le navigateur passe le test, le script ajoute alors une classe `enhanced` à l'élément HTML qui pourra être utilisée pour qualifier l'application de propriétés CSS avancées.

```
.enhanced .main {
  float: left;
}
```

Qualifier le CSS de cette façon semblait être un grand pas en avant, mais `enhance.js` était un peu brut de décoffrage et ne pouvait pas détecter ni appliquer des fonctions à un niveau plus modulaire. Heureusement, des développeurs beaucoup plus intelligents que moi ont repris l'idée et sont partis avec en courant.

Frameworks de détection de fonctionnalités

Pratiquement tous les frameworks JavaScript modernes utilisent des tests de fonctionnalités dans leur codebase interne, mais il en existe un en particulier dont la mission est d'offrir une approche standardisée pour exécuter des tests sur nos sites : Modernizr (http://bkaprt.com/rrd/2-12/), créé en 2009 par Paul Irish, Faruk Ateş, Alex Sexton, Ryan Seddon, et Alexander Farkas (FIG 2.9). Le workflow simple de Modernizr, consistant à ajouter des classes spécifiques à l'élément `html` pour signifier

FIG 2.9 : Le framework de test de fonctionnalités Modernizr

qu'une fonction telle que le multicolonne CSS est supportée (`<html class="...css-columns...">`), rend l'approche accessible aux développeurs qui ne connaissent pas toutes les subtilités de la détection JavaScript. Il est ainsi devenu le pseudo-standard pour l'application d'améliorations qualifiée.

Utiliser Modernizr

Modernizr est plutôt facile à utiliser : incluez simplement le script `modernizr.js` entre les balises `head` d'un document HTML et le script exécutera automatiquement les tests de fonctionnalités.

```
<script src="js/modernizr.js"></script>
```

Une fois les tests de Modernizr effectués, le framework conserve une propriété JavaScript du nom de ce test, stockée dans l'objet `Modernizr`, dont la valeur est `true` si le test a réussi ou `false` si le test a échoué.

```
if( Modernizr.canvas ){
  // Canvas est supporté !
}
```

Lorsqu'un test réussit, Modernizr ajoute également une classe du nom de ce test à l'élément `html`, que vous pouvez ensuite utiliser dans vos sélecteurs CSS pour qualifier l'utilisation de certaines fonctionnalités. Plus simple que de coder tous ces tests à la main, n'est-ce pas ?

Si de nombreuses fonctionnalités CSS modernes peuvent être employées sans qualification -`box-shadow`, `border-radius` ou `transition` par exemple, une utilisation trop importante de ces fonctionnalités risque de poser des problèmes d'utilisabilité dans les navigateurs qui ne les supportent pas. Par exemple, admettons que vous vouliez superposer du texte sur une image. Vous voulez que la couleur du texte soit adaptée à l'image et que l'ombre du texte fasse ressortir les caractères (FIG 2.10).

```
.img-title {
  color: #abb8c7;
  text-shadow: .1em .1em .3em rgba( 0, 0, 0, .6 );
}
```

Dans les navigateurs qui ne supportent pas la propriété `text-shadow`, le texte est pratiquement invisible (FIG 2.11) !

Pour éviter que cela ne se produise, vous pourriez choisir une présentation différente par défaut, utilisant par exemple une couleur plus contrastée, puis utiliser la détection de fonctionnalités pour la présentation idéale.

```
.img-title {
  color: #203e5b;
}
.textshadow .img-title {
  color: #abb8c7;
  text-shadow: .1em .1em .3em rgba( 0, 0, 0, .6 );
}
```

Et voilà ! Vous avez une expérience accessible pour les navigateurs modernes comme pour les navigateurs plus anciens (FIG 2.12-FIG 2.13).

FIG 2.10 : Le design souhaité

FIG 2.11 : Notre design rendu par un navigateur qui ne supporte pas la propriété `text-shadow`

FIG 2.12 : Expérience par défaut

FIG 2.13 : Expérience enrichie

Détection de la prise en charge de CSS sans JavaScript

Aussi utile la détection de fonctionnalités en JavaScript soit-elle, elle présente l'inconvénient de charger et d'exécuter du code sans aucune autre raison que de qualifier les fonctionnalités que nous voulons utiliser. Idéalement, nous devrions standardiser la détection de fonctionnalités comme les fonctionnalités elles-mêmes ; grâce au militantisme du développeur Paul Irish, le support natif d'une approche de détection de fonctionnalités CSS a été standardisé par le W3C et fait progressivement son apparition dans les navigateurs.

La fonction `@supports` (http://bkaprt.com/rrd/2-13/) emploie une syntaxe similaire à celle des media queries. En introduisant n'importe quelle propriété CSS et sa valeur (par exemple `display: flex`) dans la règle `@supports`, vous pouvez définir des blocs de style entiers à appliquer uniquement dans les navigateurs qui implémentent cette ou ces fonctionnalités CSS. Voici un exemple :

```
@supports ( display: flex ) {
  #content {
    display: flex;
  }
  ...plus de styles flexbox ici
}
```

La règle `@supports` est plutôt pratique : elle délègue le travail de détection de fonctionnalités au navigateur et nous évite d'avoir à écrire des tests personnalisés (et souvent lents et peu fiables) pour produire les mêmes résultats. Moins de travail pour les développeurs, de meilleures performances pour les utilisateurs ! En plus de la syntaxe `@supports` en CSS, vous pouvez y associer une API JavaScript appelée `CSS.supports`. En voici un exemple qualifiant l'usage de la propriété `transition` :

```
if( CSS.supports( "(transition: none)" ) ){
  // Les transitions CSS sont prises en charge !
  // Vous pourriez ajouter quelques détecteurs
  // d'événements de transition ici...
}
```

Support de @supports

Comme de nombreuses fonctionnalités CSS, la règle `@supports` se « dégradera » élégamment, ce qui signifie que vous pouvez l'inclure dans une feuille de style en toute sécurité. Les navigateurs qui ne comprennent pas la règle `@supports` l'ignoreront ainsi que les styles qu'elle qualifie.

On ne peut pas en dire autant de la méthode JavaScript qui accompagne `@supports` : ironiquement, avant d'utiliser l'API JavaScript `CSS.supports`, vous devez vous assurer que le navigateur prend en charge `CSS.supports` ! Si vous développez des sites web depuis un certain temps, vous êtes probablement habitué à ce genre de subterfuge. Ce qui plus surprenant, c'est qu'il existe déjà deux versions de `CSS.supports` dans la nature, car certaines versions du navigateur Opera utilisent une implémentation non standard (`windows.supportsCSS`). Alors voici un bout de code qui essaie d'attribuer une variable `cssSupports` à l'une ou l'autre API, le cas échéant :

```
var cssSupports = window.CSS && window.CSS.supports || "
window.supportsCSS;
```

Une fois cette normalisation en place, vous pouvez qualifier l'usage de votre `CSS.supports` comme ceci :

```
if( cssSupports && cssSupports( "(transition: none)" "
) ){
// Les transitions CSS sont supportées !
}
```

Je vais me faire l'avocat du diable pendant un instant : le problème potentiel d'une méthode de détection de fonctionnalités native comme `@supports`, c'est qu'elle part du principe que les navigateurs sont honnêtes quant à la conformité aux normes de leur propre implémentation. Par exemple, le navigateur d'Android 2 supporte `history.pushState` – servant à changer l'URL affichée dans le navigateur pour refléter les mises à jour apportées à la page depuis le dernier chargement –, mais il n'actualise pas l'adresse de la page à moins que vous ne la rafraîchissiez, ce qui rend l'implémentation complètement inutile. Du point

de vue du développeur web, toute implémentation différant de la spécification du W3C risque de rendre la fonction inutilisable, alors comment décider si une fonction particulière est supportée ou non ? La spécification suggère de définir le support comme l'implémentation d'une propriété particulière et de ses valeurs « avec un degré de prise en charge utilisable », ce qui est bien sûr complètement subjectif (http://bkaprt.com/rrd/2-14/). Vu la propension des créateurs de navigateurs à modifier leur user-agent pour améliorer leurs chances face à la concurrence, il n'est pas exclu qu'ils soient là aussi délibérément malhonnêtes. Nous verrons dans quelle mesure cette fonction de détection continuera de fonctionner à l'avenir.

Ce qui nous amène tout droit à la section suivante.

Détection du user-agent : la meilleure solution en dernier recours

Parfois, il est impossible de répondre à la question de la prise en charge d'une fonction particulière par un simple oui ou non.

Le support inégal des navigateurs s'avère particulièrement problématique lorsqu'il s'agit des « indétectables » : des fonctionnalités difficiles à détecter sur différents navigateurs (http://bkaprt.com/rrd/2-15/). Malheureusement, certains de ces indétectables sont susceptibles de détériorer l'utilisabilité ou l'accessibilité du contenu lorsqu'ils ne sont pas pris en charge, ou pire, lorsqu'ils le sont partiellement. Par exemple, Windows Phone 7 (utilisant Internet Explorer 9) supporte `@font-face` pour importer des polices personnalisées, mais seulement avec les polices installées sur l'appareil - ce qui rend la fonction parfaitement inutile.

De nombreuses fonctionnalités sont partiellement ou mal supportées dans certains navigateurs, ce qui présente un défi laborieux pour le responsive design : nous n'avons aucun moyen de savoir si ces fonctionnalités marchent correctement sans les tester nous-mêmes dans le navigateur en question.

Dans les situations où le support d'une technologie dont vous avez besoin est inégal et indétectable, et que le manque de support (ou le support partiel) risque de produire un effet indésirable, il peut être judicieux d'employer une méthode de détection de navigateur (plutôt que de fonction) en dernier recours.

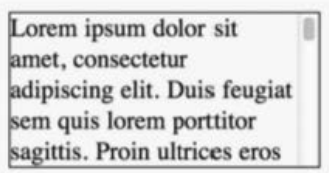


FIG 2.14 : Exemple de la propriété CSS overflow

Il convient de noter que la détection de user-agent présente de sérieux inconvénients et n'est pas viable à long terme. Évitez-la dans la mesure du possible. Cela dit, elle est parfois nécessaire. L'approche responsable consiste à épuiser tous les moyens potentiels de détection de fonctionnalités avant de recourir au user-agent. Voici quelques exemples incorporant cette solution de secours.

Débordements de contenu avec overflow

La propriété CSS `overflow` nous permet de contrôler ce qu'il se passe lorsque le contenu dépasse les limites d'un élément. Elle peut prendre les valeurs `visible` (affiche visuellement le contenu débordant), `hidden` (masque le contenu) et `scroll` ou `auto` (permet à l'utilisateur de faire défiler le contenu). Par exemple, le code CSS suivant, lorsqu'il est appliqué à un élément avec la classe `.my-scrolling-region` :

```
.my-scrolling-region {
  border: 1px solid #000;
  height: 200px;
  width: 300px;
  overflow: auto;
}
```

... produira la FIG 2.14 dans le navigateur, si le contenu dépasse la hauteur de l'élément.

Malheureusement, aussi simple que cela puisse paraître, un support partiel d'`overflow` prévaut encore sur le Web. Par exemple, de nombreux navigateurs mobiles traitent `overflow: auto` de la même façon qu'`overflow: hidden`, coupant le contenu

sans offrir à l'utilisateur un moyen d'y accéder. De plus, certaines versions anciennes d'iOS nécessitent d'utiliser deux doigts pour faire défiler une région `overflow` (ce que peu d'utilisateurs penseront vraisemblablement à essayer).

À cause de ces problèmes de prise en charge, il est risqué d'utiliser `overflow` sans qualification, mais pour compliquer encore la chose, la prise en charge d'`overflow` est quasiment impossible à détecter ! Un test de support de la propriété `overflow` réussira même si la propriété n'est pas correctement supportée, et le test du support de la propriété `overflow: auto` requiert spécifiquement une interaction de l'utilisateur pour être vérifié (autrement dit, nous ne pouvons pas être sûrs que le scrolling fonctionne tant que l'utilisateur ne l'a pas essayé). Pour ces raisons, `overflow` est un bon candidat pour un peu de détection de user-agent (comme solution de secours). Overthrow (http://bkaprt.com/rrd/2-16/) est un script qui nous permet d'utiliser `overflow` en toute sécurité ; lorsque le script est exécuté, il prend les mesures suivantes :

Il exécute tout d'abord un test de fonctionnalités pour détecter dans quelle mesure `overflow` est supporté. Ce test permet de détecter avec fiabilité les navigateurs qui ne supportent pas `overflow` ; malheureusement, certains navigateurs plus modernes qui le supportent sont détectés comme étant non compatibles, et il faut donc employer une autre approche pour prendre en charge ces navigateurs. La solution consiste à vérifier le user-agent du navigateur pour détecter les huit navigateurs environ dont on sait qu'ils sont capables de rendre `overflow` correctement, mais qui ne passent pas le test de fonctionnalités. Le script suppose que ces navigateurs continueront à supporter la fonction dans leurs futures versions (une supposition un peu risquée). Dans les navigateurs qui réussissent le test, Overthrow ajoute la classe `overthrow-enabled` dans l'élément HTML, qui peut être alors utilisée pour qualifier `overflow` dans une feuille de style.

Je tiens à insister sur le fait que nous nous sommes efforcés d'utiliser un moyen de détection indépendant du navigateur avant d'avoir recours au user-agent. Ce point est essentiel, car nous voulons que notre code soit viable à long terme. Une fois que cette classe est en place, nous pouvons qualifier l'élément de sorte à utiliser `overflow` en toute sécurité :

```
.overthrow-enabled .my-scrolling-region {
  overflow: auto;
  -webkit-overflow-scrolling: touch;
  -ms-overflow-style: auto;
  height: 200px;
}
```

Le code CSS ci-dessus fait en sorte que les navigateurs qui supportent `overflow` obtiennent une barre de défilement avec une hauteur (`height`) spécifique, tandis que les autres voient le contenu entier sans une hauteur définie qui nécessiterait de faire défiler le contenu. Mais surtout, si le test dysfonctionne ou échoue dans un navigateur qui supporte `overflow`, le contenu sera tout de même accessible. En plus des propriétés `overflow` et `height`, j'ai ajouté des propriétés à préfixes pour activer le défilement avec inertie dans les environnements tactiles basés sur WebKit et IE10. Les FIG 2.15 et 2.16 illustrent la différence entre les environnements supportés et les environnements non supportés – tous deux sont parfaitement utilisables.

Correctif pour la propriété position

Un autre exemple de propriété indétectable dangereuse est la propriété CSS `position: fixed`. Dans de nombreux navigateurs mobiles encore très répandus (Android 2, Opera Mobile, certaines vieilles versions d'iOS), le contenu à position fixe reste là où il se trouvait lors du chargement de la page et masque alors une partie du contenu (FIG 2.17).

Pour combattre ce problème, découvrez Fixed-Fixed (http://bkaprt.com/rrd/2-17/). De même qu'Overthrow, Fixed-Fixed emploie un qualificateur de classe CSS simple que vous pouvez utiliser dans vos sélecteurs ; et comme Overthrow, il exécute un test de fonctionnalités avant de recourir à la détection du user-agent si nécessaire. Voici un exemple :

```
.fixed-supported #header {
  position: fixed;
}
```

FIG 2.15 : Site d'Overthrow dans un navigateur qui supporte overflow

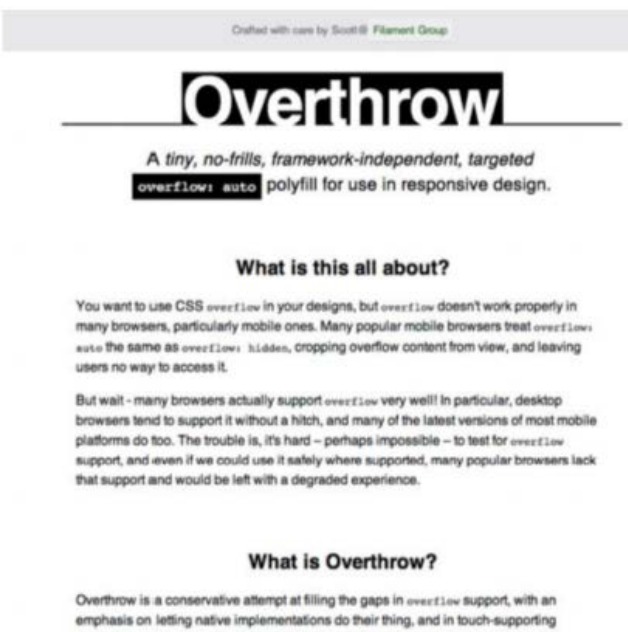


FIG 2.16 : Site d'Overthrow dans un navigateur qui ne supporte pas overflow

FIG 2.17 : Comportement attendu (gauche) et comportement bugué (droite) dans un navigateur supportant mal la propriété `position`

Aussi simple que ça ! Dans les navigateurs qualifiés, l'élément `#header` sera fixé en haut du viewport ; dans les autres, il défilera avec le reste de la page.

Palliatifs pour les fonctionnalités non supportées

Si un navigateur ne supporte pas une fonction particulière, cela veut-il dire que n'avons aucun moyen de l'utiliser dans ce navigateur ? Pas forcément. Ces dernières années, s'est répandue la pratique consistant à simuler des fonctionnalités dans les navigateurs incompatibles, connue sous le nom de *shimming* ou de *polyfilling*. D'ailleurs, le site de Modernizr fournit une liste

d'alternatives pour quasiment toutes les fonctionnalités que la bibliothèque détecte.

Les shims (ou « cales ») sont des bricolages rapides permettent d'utiliser une certaine approche alors que les polyfills sont plus approfondis. Intéressons-nous d'abord aux premiers.

Shims

Le shim le plus connu est sans doute le shim HTML5, également appelé shiv (« surin ») HTML5, peut-être en raison du mépris commun des développeurs web à l'égard des anciennes versions d'Internet Explorer (pour plus d'informations : http://bkaprt.com/rrd/2-18). En effet, les versions antérieures à la 9 ne peuvent pas appliquer de styles CSS aux éléments HTML qui n'existaient pas à la date de sortie du navigateur, ce qui signifie que des éléments HTML5 tels que `section` et `header` ne sont pas stylables dans l'un des navigateurs les plus répandus sur le Web. Par chance, un palliatif en JavaScript découvert par le développeur Sjoerd Visscher permet de faire « découvrir » par IE n'importe quel élément généré avec la méthode `document.createElement` et de styler cet élément comme tous les autres. La solution ne pourrait pas être plus simple : créez un élément d'un certain nom en utilisant `document.createElement`, et toutes les instances de cet élément qu'Internet Explorer rencontrera par la suite seront reconnues comme si elles étaient prises en charge nativement.

Remy Sharp a créé un script open source (http://bkaprt.com/rrd/2-19/), désormais maintenu par Alexander Farkas et d'autres, qui applique cette solution à tous les nouveaux éléments HTML5.

La FIG 2.18 présente un exemple de style HTML5 dans IE8 sans le shim.

```
<!DOCTYPE HTML>
<html>
<head>
  <style>
    header {
      font-size: 22px;
      color: green;
```

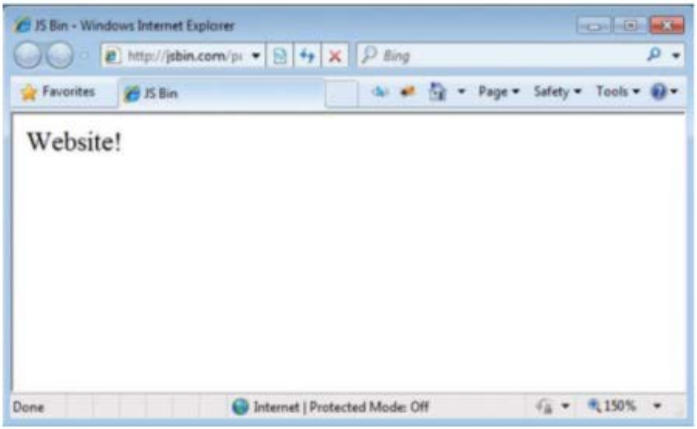


FIG 2.18 : Élément header HTML5 non reconnu et non stylé

```
    }
  </style>
</head>
<body>
  <header>Website!</header>
</body>
</html>
```

La FIGURE 2.19 illustre le rendu avec le shim.

```
<!DOCTYPE HTML>
<html>
<head>
  <!--[if lt IE 9]>
  <script src="html5shiv.js"></script>
  <![endif]-->
  <style>
    header {
      font-size: 22px;
      color: green;
```

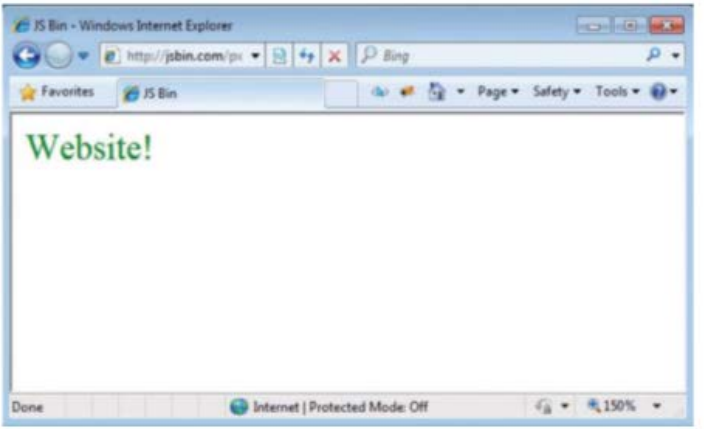


FIG 2.19 : Élément header HTML5 stylé à l'aide d'un shim

```
    }
  </style>
</head>
<body>
  <header>Website!</header>
</body>
</html>
```

En matière de développement responsable, il y a un inconvénient mineur mais non négligeable à utiliser des shims pour prendre en charge HTML5 : si le code JavaScript ne parvient pas à se charger pas dans un navigateur IE plus ancien, les éléments HTML5 ne recevront aucun style CSS. Cela peut ne pas présenter de problème majeur si le seul style que nous appliquons est un peu de couleur, comme dans l'exemple ci-dessus, mais si vous avez une mise en page en colonnes qui dépend des styles appliqués aux éléments HTML5, les éléments de la page entreront en collision dans IE, ce qui risque de poser des problèmes d'utilisabilité. Pour remédier à cela, une solution courante consiste à encadrer les éléments HTML5 par des

balises `div` avec une classe portant le nom de cet élément (`<div class="article"><article></article></div>`), puis à styler cet élément `div` à la place. Cela fait un peu gonfler le code, mais cela permet aux navigateurs modernes de bénéficier d'un code HTML5 sémantique sans qu'il y ait besoin d'un palliatif en JavaScript pour styler la page.

Polyfills de responsive design

Le terme polyfill a été inventé par Remy Sharp pour désigner une approche que Paul Irish décrit ainsi : « un polyfill est un shim qui imite une future API pour offrir des fonctionnalités de secours aux navigateurs plus anciens » (http://bkaprt.com/rrd/2-20/). Un polyfill se donne la peine de reproduire une API standardisée avec JavaScript, et est généralement plus qu'un simple palliatif.

Un shim ou un polyfill responsable doit toujours essayer de déterminer si une fonction est supportée nativement avant de reproduire son API. Pour des raisons de performance, une implémentation native sera toujours préférable, alors demandez-vous si le polyfill est absolument nécessaire. Neuf fois sur dix, il est plus responsable de fournir une expérience moins riche aux navigateurs non compatibles que de forcer des mises à jour *ad hoc* pour les fonctionnalités qu'ils ne supportent pas. La décision d'utiliser un polyfill doit être basée sur trois critères principaux : dans quelle mesure la fonction améliore-t-elle l'expérience utilisateur de votre audience, quel impact le polyfill aura-t-il sur les performances de la page et quel effort sera nécessaire pour le supprimer de votre codebase le jour venu.

En matière de responsive design, quelques polyfills me semblent particulièrement utiles.

MatchMedia : media queries en JavaScript

Si les media queries sont généralement utilisées pour appliquer du CSS, il est parfois utile de savoir si une media query s'applique également à la logique JavaScript. Il peut par exemple s'agir de télécharger des images supplémentaires de taille appropriée pour une galerie d'images. `MatchMedia` nous permet d'évaluer les media queries en JavaScript.

FIG 2.20 : Le projet `matchMedia.js` par Scott Jehl, Paul Irish et Nicholas Zakas (http://bkaprt.com/rrd/2-21/)

Pour l'utiliser, il suffit d'introduire un type de média ou une media query dans la fonction `window.matchMedia`, et celle-ci renverra un objet avec une propriété `matches` dont la valeur sera `true` ou `false` selon que le média s'applique ou non à cet instant :

```
if( window.matchMedia( "(min-width: 45em)" ).matches ){
// Le viewport fait au moins 45em de large !
}
```

Bon, j'ai oublié une petite précision : la fonction `matchMedia` n'est pas prise en charge par tous les navigateurs qui supportent les media queries CSS3. Alors avant de l'utiliser, nous devons vérifier si c'est le cas et utiliser un polyfill pour la faire fonctionner dans le cas contraire. Si vous êtes intéressé par cette deuxième option, j'ai écrit un polyfill pour `matchMedia` il y a quelques années, et Paul Irish a eu la gentillesse de créer un dépôt GitHub où nous continuons de mettre à jour le script (FIG 2.20).

Pour utiliser le polyfill, il suffit de référencer le fichier `matchMedia.js` dans votre page, et vous pourrez alors utiliser la fonction `window.matchMedia` dans tous les navigateurs, même ceux qui ne supportent pas les media queries CSS ! Mais attention : vous devrez tout de même utiliser un navigateur qui supporte les media queries pour que la valeur de la media query corresponde (bien que certains types de médias comme `screen` fonctionnent sur pratiquement tout appareil doté d'un écran).

Une fois le polyfill en place, vous pouvez utiliser la fonction `matchMedia` pour tester si les media queries CSS3 sont nativement supportées, ce qui peut être utile si vous voulez évaluer l'ajout d'un script avancé qui ne doit s'appliquer que dans les navigateurs modernes. Comme avec le CSS lui-même, la media query `only all` peut nous apporter cette information.

```
if( window.matchMedia( "only all" ).matches ){
// Les media queries sont supportées nativement !
}
```

Une autre fonction potentiellement utile de l'API `matchMedia` est sa capacité à accepter des détecteurs (*listeners*) qui nous permettent de détecter les changements apportés à l'état d'une requête `matchMedia` particulière après l'avoir vérifié une première fois. Pour nous assurer qu'elle fonctionne le plus largement possible, le polyfill `matchMedia.js` comporte une extension « listener » qui prend en charge cette partie de l'API. Il est plutôt simple d'ajouter un détecteur `matchMedia` : appelez une fonction `matchMedia` comme dans l'exemple ci-dessus et affectez-lui une méthode `addListener`, comme ceci :

```
window.matchMedia( "(min-width: 45em)" ).addListener( »
  callback );
```

Dans ce cas, `callback` est une fonction que vous pouvez définir et qui s'exécute chaque fois que la media query bascule entre les états `true` et `false`. Le premier argument introduit dans la fonction `callback` contient une référence à l'objet `matchMedia`, permettant un accès facile à sa propriété `matches` lorsque

le détecteur s'exécute. Voici un exemple d'utilisation de cette fonction :

```
window.matchMedia( "(min-width: 45em)" )
  .addListener( function( mm ){
    if( mm.matches ){
      // Le viewport fait au moins 45em de large !
    }
    else {
      // Le viewport fait moins de 45em de large !
    }
} );
```

Allô IE, ici media queries

Comme nous l'avons vu auparavant, Internet Explorer 8 et les versions antérieures ne prennent pas en charge les media queries CSS. Ainsi, un design *mobile first* rendra une mise en page conçue pour les petits écrans sur les ordinateurs de bureau – utilisable, mais pas formatée de manière idéale pour les grands écrans (FIG 2.21).

Cet inconvénient risquerait de compromettre notre approche responsive, mais fort heureusement, nous disposons de quelques alternatives fiables.

Tout d'abord, nous avons un petit polyfill, `respond.js` (http://bkaprt.com/rrd/2-22/), que j'ai développé au cours du projet de design du *Boston Globe* pour faire en sorte que les anciennes versions d'IE rendent les mises en page adaptatives comme si elles interprétaient correctement les media queries CSS3. `respond.js` fonctionne en lisant chaque feuille de style référencée dans le document afin de trouver toutes les media queries qu'elles contiennent. Le script analyse les valeurs de ces media queries pour trouver une largeur minimale ou maximale pouvant être comparée aux dimensions du viewport. Lorsqu'il trouve une media query correspondante, il injecte les styles contenus dans cette requête dans un bloc de style sur la page, permettant ainsi d'appliquer les styles dans les navigateurs qui ne comprennent pas les media queries, et le script exécute cette logique à chaque fois que le navigateur est redimensionné (et

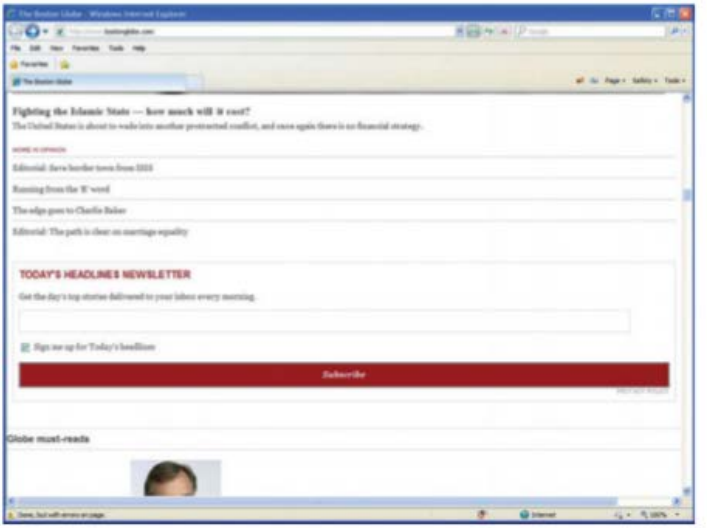


FIG 2.21 : Exemple de la page d'accueil du Boston Globe dans IE8

lorsque l'orientation de l'appareil change). Afin de rester léger et rapide, l'usage de `respond.js` est délibérément restreint aux media queries `min-width` et `max-width`, qui devraient être suffisantes pour offrir une mise en page raisonnablement adaptative aux utilisateurs d'anciennes versions d'IE.

Pour utiliser `respond.js`, référencez le script dans votre page n'importe où après vos références CSS. Je recommande d'utiliser également un commentaire conditionnel IE (une syntaxe de commentaire spéciale que les anciennes versions d'IE ignorent) autour de la balise `script` afin que le fichier soit uniquement téléchargé par les versions d'Internet Explorer qui en ont besoin. Ce commentaire conditionnel déclare : « Si le navigateur est une version d'Internet Explorer antérieure à la version 9, traiter le contenu de ce commentaire comme le reste des balises HTML sur la page. »

FIG 2.22 : Le site Web du Boston Globe visualisé sous IE8 en employant `respond.js` pour prendre en charge les media queries

```
<!--[if lt IE 9]><script src="respond.js">
  </script><![endif]-->
```

Grâce à ce script, la page d'accueil du *Boston Globe* devient plus utilisable dans les anciennes versions d'IE (FIG 2.22).

Éviter le polyfill avec du CSS statique

Une autre approche responsable permettant de combler les lacunes d'IE consiste à lui fournir des règles CSS supplémentaires qui le forcent à rendre les styles des points de rupture les plus larges d'un design responsive. Vous pouvez faire cela manuellement ou avec l'aide d'un préprocesseur CSS tel que Sass. Pour plus d'informations sur cette approche, consultez l'article écrit en 2013 par Jeremy Keith intitulé « Dealing with IE » (http://bkaprt.com/rrd/2-23/).

Cette approche permet de fournir une mise en page fluide mais non adaptative aux utilisateurs d'anciennes versions d'IE, ce qui peut convenir si elle se redimensionne correctement. Cependant, selon la taille de l'écran de votre utilisateur et les particularités de votre mise en page, l'expérience peut ne pas être idéale.

Ne rien faire du tout

La troisième option consiste à ne rien faire du tout et à fournir le site responsive tel quel aux anciennes versions d'IE. La mise en page reste alors dans son état par défaut sans media queries. Selon la mise en page, cela peut être tout à fait convenable, particulièrement si vous définissez une valeur `max-width` raisonnable pour contrôler la longueur des lignes.

TESTING RESPONSABLE

Pour s'assurer qu'un site fonctionne correctement avec des écrans de différentes tailles, des mécanismes de saisie et des navigateurs divers, rien ne vaut des tests sur de vrais appareils. Pour vous faire une bonne idée des appareils nécessaires dans un laboratoire de test personnel, lisez l'excellent billet de Brad Frost intitulé « Test on Real Mobile Devices without Breaking the Bank » (http://bkaprt.com/rrd/2-24/).

Une collection d'appareils peut coûter cher, alors pour tester une palette d'appareils pertinents, le développeur moyen pourra recourir à l'un des laboratoires communautaires proches de chez lui, qui sont heureusement de plus en plus répandus (FIG 2.23). Pour plus d'informations sur les laboratoires d'appareils près de chez vous, visitez le site Open Device Lab (http://bkaprt.com/rrd/2-25/).

Tester sur de vrais appareils est idéal, mais nous n'avons pas forcément accès à ne serait-ce qu'une fraction des appareils à prendre en charge. Dans ce cas, l'émulation peut être une excellente solution. Les appareils émulés présentent certains inconvénients, comme des performances trompeuses (car le navigateur fonctionne sur un matériel différent de celui qu'il utilise normalement), des taux de rafraîchissement d'écran lents qui rendent les animations difficiles à tester, des vitesses

FIG 2.23 : Un groupe d'amis réuni autour d'une collection d'appareils et d'ordinateurs de test. Photographie de Luke Wroblewski (http://bkaprt.com/rrd/2-26/).

de connexion souvent plus rapides que l'appareil lui-même et un manque de feedback physique qui nous permettrait de nous faire une idée précise du comportement d'un site sur un appareil particulier. Mais malgré ces inconvénients, les émulateurs sont un moyen très fiable de diagnostiquer les problèmes avec la mise en page CSS et JavaScript.

Ces temps-ci, je fais la plupart de mes tests de navigateur émulés sur BrowserStack (http://bkaprt.com/rrd/2-27/), qui offre des tests de navigateur en temps réel sur des plateformes telles qu'iOS, Android et Opera Mobile, ainsi que divers navigateurs de bureau Windows et Mac (FIG 2.24). BrowserStack offre même un moyen de tester facilement des sites locaux sur votre machine, afin que vous n'ayez pas besoin de télécharger quoi que ce soit pour tester une page.

Par ailleurs, je passe la plus grande partie de mon temps de développement dans un navigateur doté d'outils de développement puissants, comme Google Chrome ou Firefox, car leurs outils d'inspection de code apportent des informations précieuses sur la façon dont les différents composants d'un site

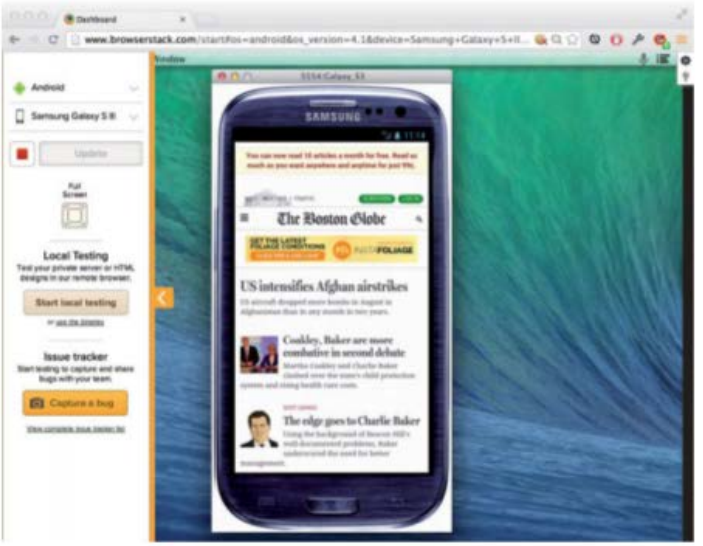


FIG 2.24 : Le service de testing BrowserStack

travaillent de concert, et me permettent même de tester des fonctionnalités qui ne sont pas activées dans le navigateur par défaut, comme les événements tactiles. Une fois que j'ai établi qu'une fonctionnalité marche, j'élargis mes tests à d'autres appareils physiques et émulés pour en vérifier l'utilisabilité et les performances, processus que je répète à de multiples reprises au fil du cycle de développement.

Avec le nombre croissant d'appareils capables d'accéder au Web, les tests de navigateurs sont devenus une activité pleine de nuances, demandant aux développeurs de prendre des décisions subjectives sur des variations mineures de l'expérience que chaque appareil reçoit. Lorsque je teste un site sur un appareil particulier, je me pose une série de questions sur son design et sa fonctionnalité :

- Le site se charge-t-il et se présente-t-il en un temps raisonnable ?
- Les fonctionnalités et le contenu principaux sont-ils utilisables et accessibles ?
- Le degré d'amélioration de la mise en page semble-t-il adapté à l'appareil ?
- Le texte est-il facile à parcourir ? La longueur des lignes est-elle propice à une bonne lisibilité ?
- Le site est-il contrôlable et navigable à l'aide des mécanismes de saisie courants de l'appareil (tactile, souris, clavier, etc.) ?
- Les régions actionnables de la page sont-elles faciles à toucher sans toucher les éléments adjacents ?
- La mise en page résiste-t-elle aux changements d'orientation, au redimensionnement du viewport et de la taille des polices ?
- Si l'appareil est équipé d'une technologie d'accessibilité (telle que VoiceOver), le contenu est-il lu d'une manière sensée ?
- La page est-elle facilement à faire défiler ? Les animations sont-elles fluides ?

Plus nous avons d'appareils à tester, meilleures seront nos chances d'atteindre nos utilisateurs où qu'ils se trouvent.

AU PROCHAIN ÉPISODE…

Dans ce chapitre, nous avons couvert les complexités d'un code durable et interopérable. Nous pouvons maintenant aborder le quatrième principe fondamental d'un responsive design responsable : les performances. Comme les performances sont un sujet important – peut-être celui qui mérite le plus d'attention pour développer des sites web responsive aujourd'hui, j'y consacrerai deux chapitres.

3 OPTIMISER LES RESSOURCES

Je veux que vous vous demandiez, quand vous concevez quelque chose, quand vous prototypez des interactions, est-ce que je pense à mon temps ou à celui de l'utilisateur ?
Paul Ford, « 10 Timeframes », http://bkaprt.com/rrd/3-01/

NOUS NE FAISONS PAS DU BON TRAVAIL

Sur les réseaux mobiles modernes, il est encore courant de devoir attendre dix secondes pour qu'une page se charge, et ce n'est qu'une fraction du temps que cela prend dans des pays où les réseaux sont plus vétustes. Pourquoi une telle lenteur ? C'est essentiellement de notre faute : nos sites sont trop lourds, et ils sont souvent assemblés et transférés d'une façon qui ne tire pas pleinement parti du fonctionnement des navigateurs. D'après l'HTTP Archive (http://bkaprt.com/rrd/3-02/), le site web moyen pèse 1,7 Mo. (Ce chiffre a probablement augmenté depuis, vous ferez peut-être bien de le vérifier.) Pour corser le

tout, la plupart des sites examinés par l'HTTP Archive ne sont même pas des sites responsive et se focalisent sur un cas d'utilisation spécifique : le classique ordinateur de bureau avec un grand écran.

C'est une très mauvaise nouvelle pour les designers responsive (et espérons-le, responsables) qui veulent prendre en charge de nombreux types d'appareils avec le même code source. En vérité, une bonne partie des critiques formulées à l'encontre du responsive design est due au poids colossal des sites responsive que l'on trouve dans la nature, comme le site assurément magnifique réalisé pour le masque Airbrake MX d'Oakley (http://bkaprt.com/rrd/3-03/), qui ne pesait pas moins de 80 Mo lors de son lancement (même s'il a été lourdement optimisé par la suite pour être beaucoup plus responsable), ou la page d'accueil de Disney, qui inclut de nombreux médias et sert un site responsive de 5 Mo à tous les appareils.

Pourquoi certains sites responsive sont-ils si lourds ? Essayer de prendre en charge tous les navigateurs et les appareils dans un seul code source peut clairement alourdir la taille des fichiers – si nous ne prenons pas de mesures pour l'éviter. La nature même du responsive design implique de fournir un code prêt à répondre aux différentes conditions qui sont susceptibles de se présenter, et il n'est pas toujours évident de fournir le bon code au bon moment étant donnée notre palette d'outils actuelle.

Pas de panique !

Un responsive design responsable est possible même pour les sites les plus complexes et les plus riches en contenu, mais il ne se fait pas tout seul. Pour transmettre des sites responsive rapidement, il est nécessaire de porter délibérément attention à nos systèmes de transmission, car notre manière de servir et d'appliquer nos ressources a un énorme impact sur la performance perçue et les temps de chargement effectifs de nos pages. En fait, notre façon de transmettre le code est plus importante que le poids de notre code.

Il est difficile de transmettre des sites de manière responsable, alors ce chapitre abordera en profondeur et de manière concrète l'optimisation de ressources adaptatives pour leur transmission sur un réseau. Nous allons toutefois commencer

par étudier l'anatomie du processus de chargement et d'amélioration pour voir comment le code côté client est demandé par le navigateur, chargé et rendu, et où les problèmes de performance et d'utilisabilité ont tendance à survenir.

Prêt ? Faisons un rapide récapitulatif du processus de chargement de page.

BALADE SUR LE CHEMIN CRITIQUE

En comprenant comment les navigateurs demandent et chargent le contenu d'une page, nous serons bien plus aptes à prendre des décisions responsables dans notre façon de transmettre le code et à réduire les temps de chargement pour nos utilisateurs. Si vous releviez tous les événements qui se produisent entre le moment où une page est demandée et celui où cette page est utilisable, vous obtiendriez ce que la communauté du développement web appelle le chemin critique. Il est de notre devoir de développeurs web de raccourcir ce chemin autant que possible.

Anatomie simplifiée d'une requête

Pour démarrer notre dissection du protocole HTTP, intéressons-nous à la base de tout ce qui se passe sur le Web : l'échange de données entre un navigateur et un serveur web. Entre le moment où notre utilisateur appuie sur Entrée et le moment où le site commence à se charger, une requête initiale est envoyée par le navigateur à un serveur DNS (Domain Name Service) local, qui contacte lui-même le serveur hôte et traduit l'URL en une adresse IP (FIG 3.1).

C'est ainsi que l'on peut résumer le processus pour les appareils qui accèdent au Web via wi-fi (ou un bon vieux câble Ethernet). Un appareil connecté à un réseau mobile comprend une étape supplémentaire : le navigateur envoie d'abord une requête à une antenne-relais locale, qui transmet la requête au serveur DNS afin de démarrer la boucle navigateur-serveur. Même avec une bonne vitesse de connexion, comme sur le réseau 3G, cette connexion radio prend une éternité en termes informatiques. Par conséquent, établir une connexion mobile

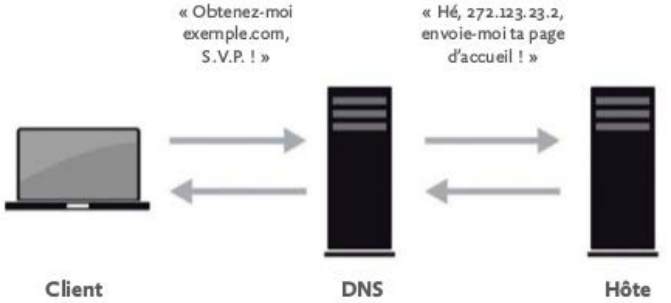


FIG 3.1 : Les bases d'une connexion Web

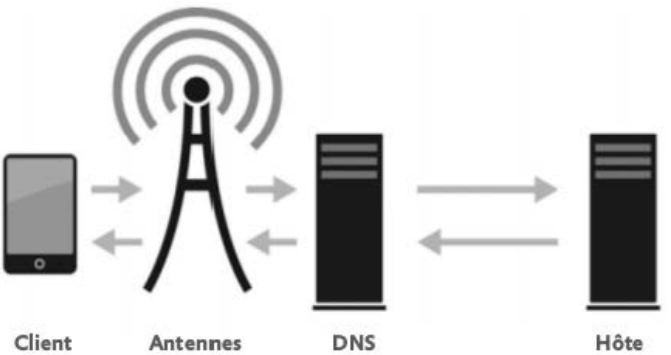


FIG 3.2 : Les connexions mobiles passent d'abord par l'antenne-relais, ce qui prend deux secondes en moyenne sur le réseau 3G (http://bkaprt.com/rrd/3-04/).

à un serveur distant peut prendre au moins deux secondes de plus que sur le wi-fi (FIG 3.2).

Vous vous dites peut-être que deux secondes, ce n'est pas si long, mais songez au fait que les utilisateurs sont capables de remarquer un retard de 300 ms, et d'en être affectés. À cause de ce retard crucial de deux secondes, le Web mobile est intrinsèquement plus lent que le wi-fi.

Par chance, les connexions LTE et 4G allègent considérablement cette peine, et leur couverture mondiale s'améliore lentement. Nous ne pouvons toutefois pas compter sur le fait qu'une connexion soit rapide, alors mieux vaut partir du principe qu'elle ne l'est pas. Dans tous les cas, une fois la connexion au serveur établie, les fichiers peuvent être transmis sans être retardés par la connexion à l'antenne-relais.

REQUÊTES, REQUÊTES, REQUÊTES !

Admettons que notre navigateur demande un fichier HTML. À mesure que le navigateur reçoit les informations contenues dans ce fichier HTML depuis le serveur, il les analyse de manière procédurale, cherche les références à des ressources externes qui doivent également être téléchargées, et convertit le code HTML en une arborescence d'éléments HTML connue sous le nom de DOM, pour Document Object Model. Une fois cette structure DOM établie, nous pouvons parcourir et manipuler les éléments du document avec JavaScript et les styler visuellement avec CSS comme nous le souhaitons.

Les subtilités du parsing HTML (et ses variations selon les navigateurs) peuvent faire l'objet d'un livre à elles seules. Je serai donc bref : le plus important est d'avoir une idée de l'ordre fondamental des opérations lorsqu'un navigateur interprète et rend du code HTML.

- CSS, par exemple, fonctionne mieux lorsque tous les styles utilisés dans la mise en page initiale de la page sont chargés et parsés avant que le document HTML ne soit visuellement rendu sur un écran.
- À l'inverse, les fonctions JavaScript sont plus souvent appliquées aux éléments de la page une fois que ceux-ci ont été chargés et rendus.

Mais JavaScript comme CSS produisent des cahots sur notre chemin critique, empêchant notre page de s'afficher pendant qu'ils se chargent et s'exécutent. Intéressons-nous un instant à l'ordre de ces opérations.

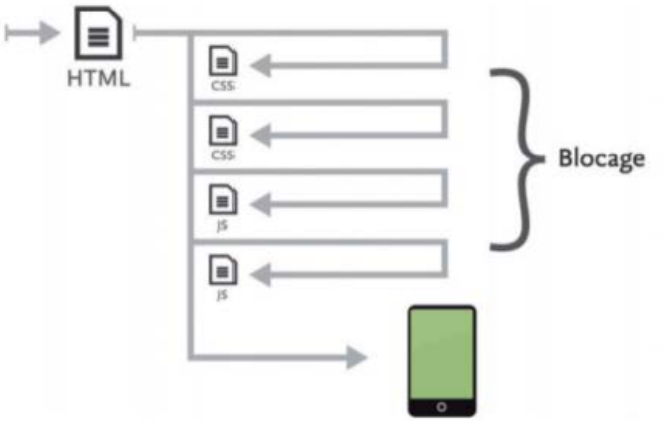


FIG 3.3 : Blocage des requêtes CSS et JavaScript au cours du chargement de la page

Rendu et blocage

Le document HTML le plus rapide à charger est un document sans référence à des fichiers externes, mais il n'est pas des plus courants. Un document HTML typique incorpore une pléiade de ressources externes telles que des feuilles de style, du code JavaScript, des polices de caractères et des images.

Vous verrez souvent CSS et JavaScript entre les balises `head` du document HTML, respectivement sous la forme d'éléments `link` et `script`. Par défaut, les navigateurs ne rendent pas le contenu d'une page tant que ces ressources n'ont pas été chargées et parsées, comportement que l'on appelle blocage (FIG 3.3). Les images, elles, sont des ressources non bloquantes, et le navigateur n'attendra pas qu'une image soit chargée pour rendre une page.

Malgré son nom, le blocage du rendu pour CSS permet de charger l'interface utilisateur de manière cohérente. Si vous chargez une page avant que son code CSS ne soit disponible, vous verrez une page non stylée par défaut ; lorsque le CSS aura fini de se charger et que le navigateur l'appliquera, le contenu de la page se réorganisera conformément à la mise en page nouvellement stylée. Ce processus en deux étapes s'appelle FOUC,

pour *flash of unstyled content* (flash de contenu sans style), et il peut être extrêmement déroutant pour les utilisateurs. Il peut donc être souhaitable de bloquer le rendu de la page tant que le CSS n'est pas chargé, du moment que celui-ci se charge en peu de temps - ce qui n'est pas toujours chose aisée.

Pour ce qui est de JavaScript, le blocage du rendu a presque systématiquement un impact négatif sur l'expérience utilisateur, notamment à cause d'une méthode JavaScript encore en utilisation appelée `document.write`, utilisée pour injecter directement du code HTML à un point d'une page. On considère généralement que c'est une mauvaise pratique maintenant que des méthodes plus découplées et plus performantes sont disponibles dans JS, mais `document.write` est encore utilisé, notamment par les scripts publicitaires. `document.write` pose un problème majeur : s'il s'exécute une fois qu'une page a fini de se charger, il écrase le contenu entier du document. Pas génial. Malheureusement, les navigateurs n'ont aucun moyen de savoir si le script qu'il demande contient une référence à `document.write`, aussi ont-ils tendance à privilégier la sécurité et à supposer que c'est le cas. Si le blocage du rendu permet d'éviter la suppression potentielle de tout le contenu, il oblige également les utilisateurs à attendre le chargement des scripts avant de pouvoir accéder à la page, même si les scripts en question n'auraient pas posé problème. L'une des solutions à ce problème consiste à éviter d'utiliser `document.write`. Au chapitre suivant, nous étudierons divers moyens de charger des scripts en contournant le comportement de blocage par défaut afin d'améliorer la performance perçue.

FAMILIARISEZ-VOUS AVEC VOS OUTILS DE DÉVELOPPEMENT

Nos navigateurs intègrent d'excellents outils pour nous aider à inspecter, tester et analyser nos pages et voir ce qu'il se passe véritablement sous le capot. Il peut être utile de se familiariser avec ces outils dans plusieurs navigateurs, mais je parlerai ici de mes préférés, les outils de développement de Chrome. En ce qui concerne les performances de chargement de page, deux onglets sont particulièrement utiles : Network et Timeline.

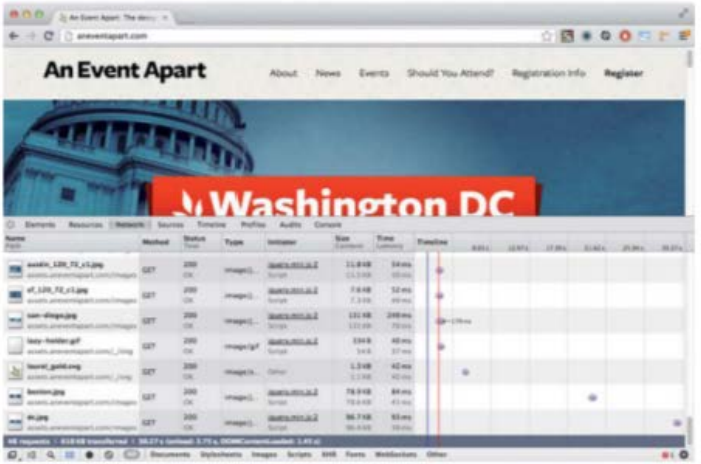


FIG 3.4 : L'onglet Network dans Chrome

L'onglet Network contient le détail de toutes les ressources que le navigateur doit obtenir pour rendre la page (FIG 3.4). Il comprend entre autres des colonnes pour le type de fichier, le statut de la mise en cache, la taille du fichier et la durée de la requête ; en bas du panneau, vous trouverez un décompte total. Mon ami Mat Marquis aime l'appeler le « panneau du jugement dernier », et je suis assez d'accord : c'est le meilleur moyen d'évaluer la transmission d'un site web en détail.

L'onglet Timeline détaille l'ordre dans lequel les ressources sont chargées et rendues, et présente ces événements dans un tableau pratique sous forme de cascade que vous pouvez parcourir de haut en bas et de gauche à droite le long d'un axe temporel (FIG 3.5). Avec l'onglet Timeline, on peut enregistrer le processus de chargement de la page ou une séquence d'interactions, afin de déterminer le délai du rendu de certaines parties de la page, les requêtes HTTP qui retardent ce rendu, et les améliorations qui entraînent un ajustement de la position des éléments de la page (*reflow*) ou un nouveau rendu d'un élément à la même place (*repaint*). À l'aide de cet outil, je découvre

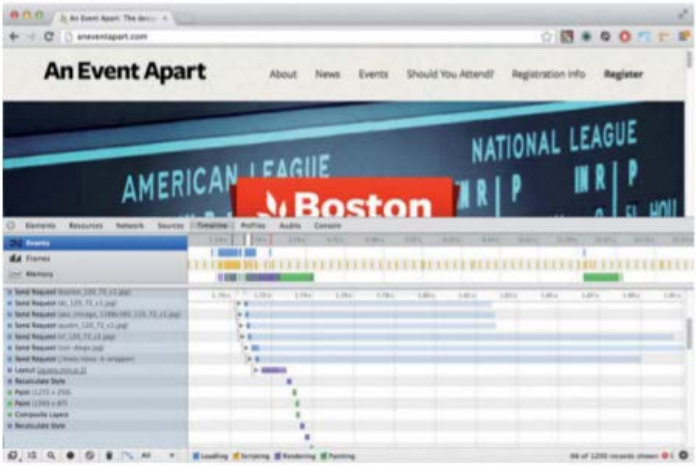


FIG 3.5 : L'onglet Timeline des outils de développement de Chrome

souvent des modifications à apporter à la concaténation et au chargement de mes fichiers, puis je peux enregistrer le processus à nouveau pour vérifier si mes changements ont amélioré les performances.

Les navigateurs chargent et lisent les ressources de différentes manières ; je vous recommande donc de vous familiariser avec les outils de développement de plusieurs navigateurs. IE, Firefox, Opera, Safari ainsi que des navigateurs mobiles tels que Chrome pour Android et iOS Safari offrent tous des capacités de débogage qui sont simples à utiliser et extrêmement utiles pour dénicher les bugs. Pour plus d'aide concernant les outils de développement du navigateur de votre choix, consultez Secrets of the Browser Developer Tools (http://bkaprt.com/rrd/3-05/).

Performance perçue : l'indicateur le plus critique

Pour évaluer les performances d'un site web, il est important de penser à la fois au temps de chargement et au poids effectifs, mais également à la perception du chargement de la page. Après tout, une page est souvent utilisable bien avant que toutes les

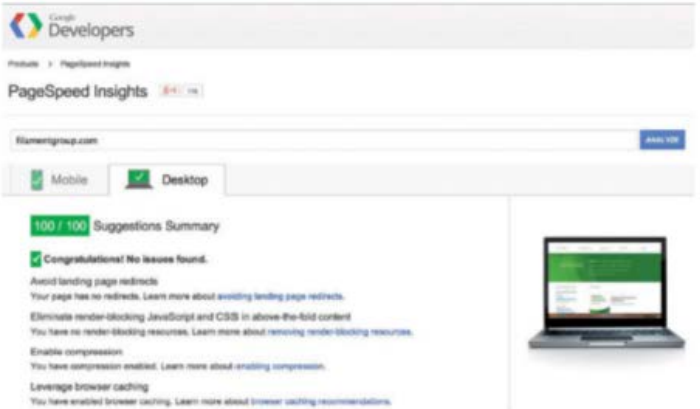


FIG 3.6 : Résultats du test PageSpeed Insights pour le site du Filament Group

ressources n'aient été téléchargées ; ce temps de chargement perçu est souvent plus important pour nous que le temps de chargement total de la page (une page peut prendre dix secondes à se charger complètement sur le réseau 3G, mais il se peut que l'utilisateur puisse interagir avec la page après seulement quelques secondes, voire moins). S'il existe des moyens d'améliorer la performance perçue sans vraiment améliorer le temps de chargement de la page (par exemple en affichant une icône de chargement pendant que le reste du contenu finit de se charger), il existe un consensus général quant à ce qui constitue un temps de chargement « suffisamment rapide ». Le chargement en une seconde est devenu de facto l'objectif standard, et il existe d'excellentes ressources qui expliquent les optimisations à apporter pour y parvenir.

L'une de ces ressources est PageSpeed Insights de Google (http://bkaprt.com/rrd/3-06/). PageSpeed Insights propose une application web et des extensions de navigateur pour analyser vos sites et recommander des améliorations. Voici une capture d'écran du test réalisé sur le site hautement optimisé du Filament Group (FIG 3.6). Et oui, la vitesse de chargement de votre site est un sujet de vantardise parfaitement acceptable !

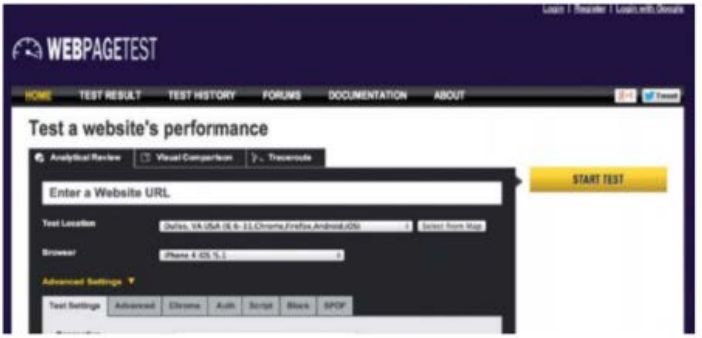


FIG 3.7 : WebPagetest est une ressource fantastique pour évaluer la performance perçue.

Pour tester la performance perçue de votre site, je vous recommande chaudement les outils de WebPagetest (http://bkaprt.com/rrd/3-07/), un projet développé par Patrick Meenan de Google (FIG 3.7). Pour utiliser WebPagetest, saisissez une URL et remplissez les champs du formulaire selon les résultats que vous voulez obtenir - vous pouvez même choisir différents emplacements de test dans le monde, ce qui peut être particulièrement révélateur. Une fois le test exécuté, vous obtiendrez des résultats détaillés concernant les données de performance de la page.

De tous les indicateurs de WebPagetest, le Speed Index est peut-être le plus pertinent pour analyser la performance perçue. La formule du Speed Index prend en compte des facteurs tels que la taille du viewport et le moment où la page commence à être rendue (qui se traduit en un score représentant le temps qu'une page prend avant d'être utilisable) ; plus le score est faible, meilleures sont les performances. D'après Google, le Speed Index moyen pour un site du top 300 000 d'Alexa est de 4 493, alors que le score moyen d'un site du premier décile est de 1 388. Pour le développeur Paul Irish de Google, 1 000 est un bon score à viser (http://bkaprt.com/rrd/3-08/).

Il n'est pas facile d'obtenir un Speed Index faible, et même lorsque nous avons de bonnes bases en place, il suffit d'un changement de cadre – par exemple l'ajout d'une publicité tierce

ou d'un diaporama - pour dégrader la performance perçue d'un site bien optimisé. Pour combattre ces situations, il est judicieux d'établir un budget de performance aussi tôt que possible.

ÉTABLIR UN BUDGET DE PERFORMANCE

L'idée d'un budget de performance est relativement nouvelle, et il semble que la communauté web s'interroge encore sur sa définition et son application. Cela dit, l'idée de base est solide : un budget de performance est un nombre, ou un ensemble de nombres servant de ligne directrice pour déterminer si vous pouvez vous permettre d'ajouter un certain code dans votre codebase, ou si les performances d'un site existant doivent être améliorées. Ces nombres peuvent représenter le poids de transfert de la page (« la page ne doit pas dépasser X kilooctets et ne pas émettre plus de Y requêtes ») ou le temps de chargement perçu (« la page doit être utilisable en X secondes au maximum »), quoique je préfère vérifier les deux. Je trouve également utile de communiquer votre enthousiasme pour ces chiffres à votre client afin qu'ils soient respectés tout au long du processus de développement. Les chiffres ne sont pas seulement le problème de l'équipe technique ; tout le monde doit y prêter attention.

À ce propos, je voudrais signaler que les performances ne sont pas seulement un problème d'ordre technique, mais également culturel au sein d'une organisation. De bonnes performances font un bon design, et les performances doivent être une priorité dès le départ plutôt que d'être laissée aux développeurs pour qu'ils les optimisent après coup. Les décisions prises au début d'un projet auront un impact considérable sur les contraintes que nous devrons affronter lorsque nous passerons au code, et les développeurs ont tout intérêt à s'impliquer dès le début du processus de planification du site pour rappeler aux membres de l'équipe à quel point leurs stratégies de contenu et de design sont susceptibles d'affecter les performances du site. Comme le fait observer Tim Kadlec dans son billet « Holistic Performance » : « La performance n'est pas seulement une question de développement, c'est un composant fondamental de l'expérience utilisateur » (http://bkaprt.com/rrd/3-09/).

Il peut être difficile de déterminer quel doit être le budget de performance d'un site, et celui-ci varie selon les projets. Si vous partez de zéro, une analyse des performances des sites de vos concurrents peut vous donner une bonne idée de ce à quoi vous vous confrontez et des indicateurs que vous voulez battre. Concentrez-vous sur les améliorations que vous ajoutez sur votre site et essayez de limiter au maximum les ressources supplémentaires pour rester dans les temps.

Dernièrement, j'ai pris plaisir à utiliser l'outil Grunt-Perf-Budget de Tim Kadlec (http://bkaprt.com/rrd/3-10/) pour garder un œil sur notre budget de performance à mesure que nous développons notre codebase. L'outil de Kadlec est un programme en ligne de commande que vous pouvez automatiser pour qu'il s'exécute à chaque fois que vous apportez des changements à votre site. Par défaut, l'outil teste vos pages sur des serveurs distants de WebPagetest et vous avertit si vous dépassez le budget que vous vous êtes fixé. Généralement, j'essaie de m'en tenir à un indice de 1 000 (une seconde) pour le rendu initial. Voici à quoi ressemble le rapport généré par l'outil lorsque je le lance :

```
$ grunt perfbudget
Running "perfbudget:dev" (perfbudget) task
Running test...
Test ID ADKLKJCLKD.... obtained....
Test Pending...
Test Started...
>> ------------------------------------------------
>> Test for http://client-website.com/      FAILED
>> ------------------------------------------------
>> render: 594 [PASS]. Budget is 1000
>> SpeedIndex: 1049 [FAIL]. Budget is 1000
>> Summary: http://www.webpagetest.org/result/140712_
EJ_....
```

MOINS DE REQUÊTES

Si je n'avais qu'un seul conseil à vous donner en ce qui concerne les requêtes, ce serait de réduire le nombre de requêtes bloquantes dans votre document. Chaque requête HTTP qui bloque le rendu est une barrière entre nos utilisateurs et le contenu qu'ils sont venus chercher. Si une requête bloquante ne parvient pas à se charger, l'utilisateur ne pourra pas accéder à votre site tant qu'elle n'aura pas expiré, ce qui peut prendre jusqu'à trente secondes dans les navigateurs les plus courants aujourd'hui. C'est un long moment à passer devant une page blanche, à supposer que vous décidiez d'attendre.

Outre la réduction des requêtes bloquantes, il y a bien d'autres choses à faire pour optimiser les fichiers que nous transférons afin qu'ils se chargent plus vite.

PRÉPARER LES FICHIERS POUR LEUR TRANSFERT

Lorsque vous préparez les ressources à transférer, il est important de réduire à la fois le nombre total de fichiers que vous envoyez par le réseau ainsi que leur taille.

Optimiser les fichiers image

Pour que les images que nous transférons soient aussi légères que possible, il est essentiel d'optimiser leur compression. Pour compresser une image, vous pouvez par exemple vous servir de la fonction Enregistrer pour le Web de Photoshop et régler les paramètres d'exportation, mais il existe d'autres outils conçus purement pour l'optimisation d'images. Pour moi, le plus pratique est ImageOptim (http://bkaprt.com/rrd/3-11/), qui offre une simple interface « glisser-déposer » permettant d'optimiser des images par lots (FIG 3.8). Déplacez les images dans la fenêtre et elles seront remplacées par des versions optimisées.

Si vous voulez automatiser cette optimisation, essayez l'un des puissants outils de compression d'image en ligne de commande qui existent. OptiPNG (http://bkaprt.com/rrd/3-12/) et jpegtran (http://bkaprt.com/rrd/3-13/) sont conçus pour optimiser respectivement les images PNG et JPEG, et peuvent être facilement intégrés dans un workflow de compilation automatisé à

FIG 3.8 : L'interface à « glisser-déposer » d'ImageOptim

l'aide d'un outil tel que grunt-contrib-imagemin (http://bkaprt.com/rrd/3-14/).

Lorsqu'il s'agit d'optimiser des images, les graphiques plus simples ont tendance à être mieux compressés que les images comportant de nombreuses couleurs et des dégradés. Certains designers sont même allés jusqu'à développer des solutions visuelles créatives qu'ils n'auraient peut-être pas explorées autrement pour répondre à ces contraintes de taille. Par exemple, le site de la conférence dConstruct 2012, conçu par l'équipe exceptionnelle de Clearleft, contient des images bichromiques conçues pour réduire le poids des fichiers tout en offrant un design unique et attrayant (FIG 3.9). En dépit de son esthétique riche, la page d'accueil entière ne pèse que 230 ko !

Concaténer les fichiers texte

Réduire le nombre de fichiers chargés n'implique pas forcément de les supprimer ; la pratique consistant à combiner automatiquement les fichiers, également appelée concaténation, est courante avec les fichiers CSS et JavaScript. Après tout, il est préférable de réduire le nombre de requêtes bloquantes. Vous pouvez concaténer les fichiers à la main ou automatiser le processus ; si vous travaillez sur un site relativement complexe, je vous conseille de laisser les outils faire le travail à votre place.

Un rapide exemple consiste à exécuter la commande `$ cat` dans une fenêtre de terminal, qui accepte un nombre illimité de noms de fichier que l'on fait suivre d'un caractère `>` afin de

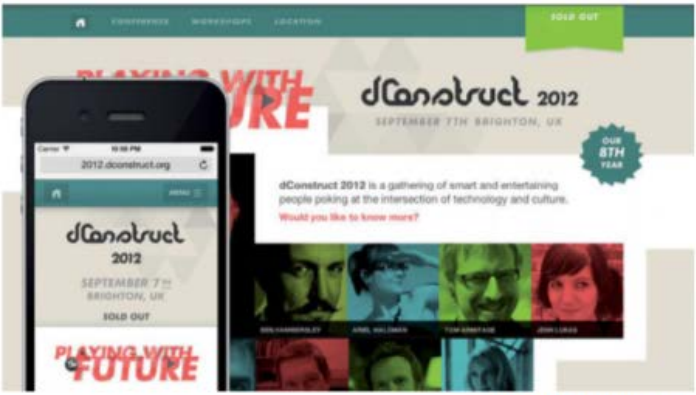


FIG 3.9 : Le site Web responsable de la conférence dConstruct (http://bkaprt.com/rrd/3-15/)

spécifier un nouveau fichier combinant le contenu de tous les fichiers énumérés :

```
$ cat foo.js bar.js > foobar.js
```

Minifier les fichiers texte

Une fois que nous avons réduit le nombre de fichiers qui se chargeront sur tous les appareils, il faut que ces fichiers concaténés soient aussi petits que possible. Il existe pour cela plusieurs méthodes. La première est la minification, pratique consistant à supprimer automatiquement toutes les portions du fichier qui ne sont pas nécessaires au navigateur. Dans les fichiers HTML, il s'agit généralement des espaces et des sauts de ligne entre les éléments. Dans les fichiers CSS et JavaScript, il s'agit essentiellement des espaces, des commentaires et des sauts de ligne, mais la minification de JavaScript va souvent plus loin, en renommant par exemple les variables pour utiliser moins de caractères (comme le nom des variables n'a pas besoin d'avoir un sens pour l'ordinateur) (FIG 3.10).

Souvent le code source d'un fichier minifié tient sur une seule ligne de texte, sans retour à la ligne.

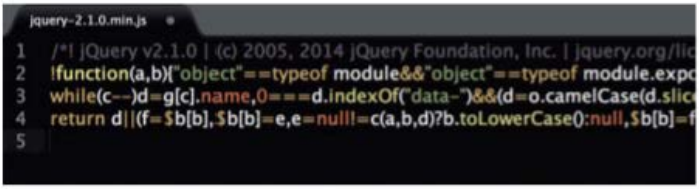


FIG 3.10 : Une copie minifiée de jQuery visualisée dans l'éditeur Sublime Text 2

Sur la page de téléchargement de jQuery, les fichiers minifiés sont souvent appelés « versions de production » en raison de leur manque de lisibilité pour les humains et de l'absence de numéros de ligne qui peuvent être utiles pour le débogage. (Pas de sauts de ligne, donc pas de numéros de ligne !)

Compresser les fichiers texte

Une fois nos fichiers texte concaténés et minifiés, nous allons les compresser avant de les envoyer sur le Web. Un protocole de compression courant, Gzip, permet de réduire la taille des fichiers texte pour les transférer du serveur au navigateur. Pour en savoir plus sur le fonctionnement de Gzip et de son algorithme Deflate, lisez l'explication d'Antaeus Feldspar (http://bkaprt.com/rrd/3-16/).

À chaque requête, tous les navigateurs modernes sont configurés pour informer le serveur qu'ils sont capables de décompresser des fichiers Gzip, et Gzip est facile à configurer sur la plupart des serveurs web. Par exemple, pour activer Gzip pour tous les fichiers HTML, CSS et JavaScript sur un serveur Apache, ajoutez un fichier `.htaccess` à la racine publique de votre site web en y insérant les instructions suivantes :

```
<IfModule mod_deflate.c>
AddOutputFilterByType DEFLATE text/html text/css text/
javascript
</IfModule>
```

Pour vérifier si Gzip fonctionne, chargez votre site et ouvrez l'onglet Réseau des outils de développement de votre navigateur.

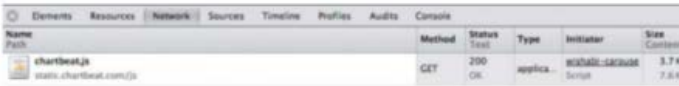


FIG 3.11 : Exemple d'une requête JavaScript compressée à l'aide de Gzip dans les outils de développement de Chrome

Si vous voyez deux tailles différentes indiquées pour un fichier donné, c'est que cela fonctionne. Par exemple, dans l'onglet Network de Chrome, vous pouvez vérifier que la colonne Size comporte un fichier de petite taille et un fichier de grande taille (3,7 Ko et 7,6 Ko, comme illustré à la FIG 3.11). Il s'agit respectivement de la taille de transfert du fichier (compressé avec Gzip) et de la taille réelle du fichier (après sa décompression par le navigateur).

Le cache est roi

Notre discussion sur la préparation des ressources pour leur transfert ne serait pas complète si l'on oubliait de mentionner la mise en cache du navigateur - processus consistant à stocker une copie locale et statique d'un fichier. La mise en cache est un sujet complexe, mais une compréhension de son fonctionnement de base et des divers caches auxquels nous avons accès dans les navigateurs modernes vous aidera considérablement à accélérer vos sites.

Optimiser pour une mise en cache classique

Le premier cache à prendre en compte est le cache par défaut du navigateur. Le rôle du cache par défaut est de stocker automatiquement tous les fichiers téléchargés par le navigateur de manière à éviter d'envoyer une nouvelle requête réseau et à utiliser la copie locale la prochaine fois que ces fichiers seront sollicités. Sur un site web donné, nous pouvons nous permettre de mettre en cache la plupart des ressources pour une brève période, et certaines ressources peuvent être mises en cache sur une plus longue durée. Tout contenu hautement dynamique fait exception à cette règle, tel que le texte d'une discussion en temps réel.

Lorsque vous transmettez des fichiers sur le Web, vous pouvez configurer la mise en cache de chaque fichier. Vous devez pour cela paramétrer des en-têtes de réponse, qui sont simplement des métadonnées que le serveur inclut avec chaque réponse. Les en-têtes de réponse peuvent être configurés simplement sur n'importe quel serveur web tel qu'Apache. Par exemple, en indiquant une date lointaine pour l'en-tête Expires, un mois ou un an à compter du transfert du fichier, le navigateur comprend qu'il doit le conserver jusqu'à cette date. Soyez tout de même prudents, car si vous pouvez avoir intérêt à mettre en cache les fichiers qui ne risquent pas de changer de sitôt (comme les fichiers CSS, JavaScript, les images et les polices de caractères), vous devrez par exemple éviter de mettre en cache un fichier HTML contenant une liste d'articles récents. Pour d'excellentes informations sur la configuration de vos fichiers pour une mise en cache optimale, consultez les recommandations d'HTML5 Boilerplate (http://bkaprt.com/rrd/3-17/), ainsi que la suite d'outils Make the Web Faster de Google (http://bkaprt.com/rrd/3-18/).

Pensez aux caches hors-ligne HTML5

Outre la mise en cache ordinaire, la plupart des navigateurs modernes offrent des caches accessibles même si l'appareil n'est pas connecté. Ces caches sont très utiles, car il est courant pour les utilisateurs, y compris dans les régions développées, de subir des pertes de connectivité temporaires (par exemple lorsque vous passez sous un tunnel ou sortez de la zone de couverture du réseau mobile). Le cache d'application HTML5 est un moyen extrêmement simple de préparer un site pour qu'il puisse être utilisé hors connexion. Pour utiliser le cache d'application, ajoutez un fichier portant un nom comme `exemple.appcache` dans le répertoire de votre site web et référencez-le dans votre ou vos fichiers HTML comme ceci :

```
<html manifest="exemple.appcache">
```

Le contenu de ce fichier `exemple.appcache` dira au navigateur quelles ressources mettre en cache pour un usage hors ligne et quelles sont celles qui doivent toujours être rapatriées

par le réseau. Votre fichier `exemple.appcache` pourrait ressembler à cela :

```
CACHE MANIFEST
index.html
styles.css
logo.jpg
scripts.js
```

Ces instructions demandent au navigateur de faire en sorte que les fichiers `index.html`, `styles.css`, `logo.jpg` et `scripts.js` soient disponibles si vous essayez de les charger hors connexion. Bien sûr, vous pouvez également utiliser le cache d'application pour répondre à des scénarios plus compliqués et plus nuancés. Avec le cache d'application et d'autres fonctionnalités associées du navigateur telles que le stockage local et l'API Service Worker (http://bkaprt.com/rrd/3-19/), il est possible de définir quelles fonctionnalités d'un site doivent fonctionner hors connexion et lesquelles nécessitent une connexion au Web (par exemple effectuer un paiement par carte de crédit).

Un accès hors connexion peut être utile pour nos utilisateurs dans les moments les plus critiques : nous devrions toujours envisager de l'activer, surtout sur les sites simples pour lesquels cela demande peu d'efforts. Pour plus d'informations sur le cache d'application, visitez le site HTML5 Rocks (http://bkaprt.com/rrd/3-20/).

Automatisation

Vous pourriez gérer manuellement les techniques évoquées précédemment, mais je ne vous le conseille pas. Ces derniers temps, les outils permettant d'automatiser ces tâches se sont considérablement améliorés ; si vous ne les utilisez pas encore, vous ne savez pas ce que vous perdez. Découvrons deux outils qui valent la peine qu'on s'y intéresse.

CodeKit

CodeKit est une application de bureau pour Mac qui propose d'exécuter diverses tâches courantes liées à la compilation de sites web, comme l'optimisation des images, la concaténation

FIG 3.12 : CodeKit avec un certain nombre de tâches configurées (http://bkaprt.com/rrd/3-21/)

et la minification des fichiers, l'exécution de préprocesseurs tels que Sass et bien plus encore (**FIG 3.12**).

Grunt

Pour ceux qui n'ont pas peur de se plonger un peu dans la ligne de commande, Grunt est un gestionnaire de tâches programmé en JavaScript que vous pouvez configurer pour exécuter autant de tâches de compilation que vous le souhaitez, de la concaténation et de la minification des fichiers CSS et JavaScript à la copie et à la manipulation du système de fichiers, en passant même par la génération d'icônes (**FIG 3.13**).

FIG 3.13 : Découvrez toutes les tâches officiellement prises en charge sur le site Web de Grunt et sur GitHub (http://bkaprt.com/rrd/3-22/).

4 TRANSMISSION RESPONSABLE

MAINTENANT QUE NOUS avons préparé nos fichiers pour le Web, voyons comment les transmettre de manière responsable : HTML, CSS, images, polices de caractère et JavaScript.

TRANSMETTRE DU HTML

Nous avons précédemment vu que le site web moyen pesait aujourd'hui environ 1,7 Mo. Le code HTML ne constitue qu'une petite partie de ce poids, environ 55 Ko, mais sa taille ne suffit pas à décrire son impact sur la performance perçue et le temps de chargement total. Comme pour la plupart des technologies côté client, chaque ligne de HTML peut faire référence à des ressources externes qui doivent être téléchargées via le réseau (comme les images et la vidéo), et chacune de ces ressources pèse un certain poids et a un certain impact sur le temps de chargement.

Contenu *mobile first*

Dans le billet de Luke Wroblewski paru en 2009 et intitulé « Mobile First » (qui a précédé le livre éponyme paru chez A Book Apart), celui-ci fait remarquer qu'en concevant pour des appareils dont la taille d'écran est fortement contraignante, vous êtes obligé de vous focaliser sur les données et les actions les plus importantes : « Il n'y a tout simplement pas assez de place sur un écran de 320 par 480 pixels pour des éléments superflus. Vous devez établir un ordre de priorité. » (http://bkaprt.com/rrd/4-01/)

Idéalement, nous ne devrions transmettre que le contenu et les fonctionnalités que nos utilisateurs désirent, quel que soit l'appareil qu'ils utilisent. En pratique, cela implique de trier le contenu de nos pages ou de nos écrans. Relevez le contenu qui n'est pas essentiel pour répondre au but premier de chaque page, comme les accroches menant à des articles externes, l'intégration des réseaux sociaux, les commentaires et les publicités. Ce contenu auxiliaire peut ne pas être nécessaire pour le rendu initial de la page ; en le transmettant d'avance, vous risquez d'allonger le temps de chargement de la page avant qu'elle ne soit utilisable, particulièrement sur les connexions lentes.

Il est utile d'identifier quelles portions du contenu sont absolument nécessaires et lesquelles peuvent être chargées plus tard, une fois que l'essentiel a été transmis. C'est ce que l'on appelle un chargement différé ou paresseux.

Chargement de contenu différé pour améliorer la performance perçue

Si un morceau de contenu supplémentaire est déjà accessible à un endroit qui lui est consacré ailleurs sur le site, ce contenu est un bon candidat pour un chargement différé. En d'autres termes, si le contenu est accessible en un ou deux clics, considérez que sa présence sur d'autres pages est un luxe pour l'utilisateur, un petit bonus qui n'a rien d'essentiel (contrairement à tout ce que le service marketing pourra vous dire).

En configurant nos pages pour transférer le contenu critique en priorité, le chargement initial de la page sera plus rapide. Une fois la page affichée, nous pouvons utiliser JavaScript pour charger le contenu complémentaire.

Mettre en place un chargement conditionnel

Deux des meilleurs articles ayant proposé l'idée du chargement différé dans le responsive design sont intitulés « Conditional Loading for Responsive Designs » (http://bkaprt.com/rrd/4-02/) et « Clean Conditional Loading » (http://bkaprt.com/rrd/4-03/), tous deux écrits par le génie des standards du Web, Jeremy Keith. Ces articles offrent des modèles en JavaScript permettant de charger un certain fragment de HTML dans une page existante en fonction de la taille du viewport.

À l'époque où Keith écrivait ces articles, nous avons publié notre propre approche appelée Ajax-Include (http://bkaprt.com/rrd/4-04/). Le modèle Ajax-Include peut être utilisé pour charger du contenu de manière différée dans certains environnements, vous permettant ainsi de transmettre une version rationalisée du contenu - un lien vers une section du site, par exemple - et de remplacer ce contenu par un fragment du HTML de cette section une fois la page chargée.

Hypothétiquement, nous pourrions choisir d'appliquer le modèle Ajax-Include aux blocs de contenu de la page d'accueil du *Boston Globe* illustrés à la FIG 4.1. Chaque bloc comprend un lien vers une section principale du site (sport, informations locales, chroniques) ainsi qu'un aperçu du contenu de la page d'accueil de cette section : des liens vers des articles, des images, des vidéos, etc. Le balisage de base des liens au début de chaque bloc de contenu ressemblerait à quelque chose comme ça :

```
<a href="/sports">Sports</a>
```

Pour inclure le contenu de façon dynamique à la suite de chacun de ces liens, nous pouvons modifier cette balise pour la préparer à accueillir le modèle Ajax-Include. Pour cela, nous devons incorporer un ou deux attributs HTML5 `data`, qui sont de nouveaux attributs personnalisables que nous pouvons utiliser sur n'importe quel élément HTML pour stocker des données. Ces attributs n'ont aucun effet à eux seuls, mais ils offrent un moyen pratique de définir des informations de configuration pour nos scripts (simplifiant ainsi la maintenance). D'un point de vue syntaxique, les attributs `data` sont des attributs ouverts

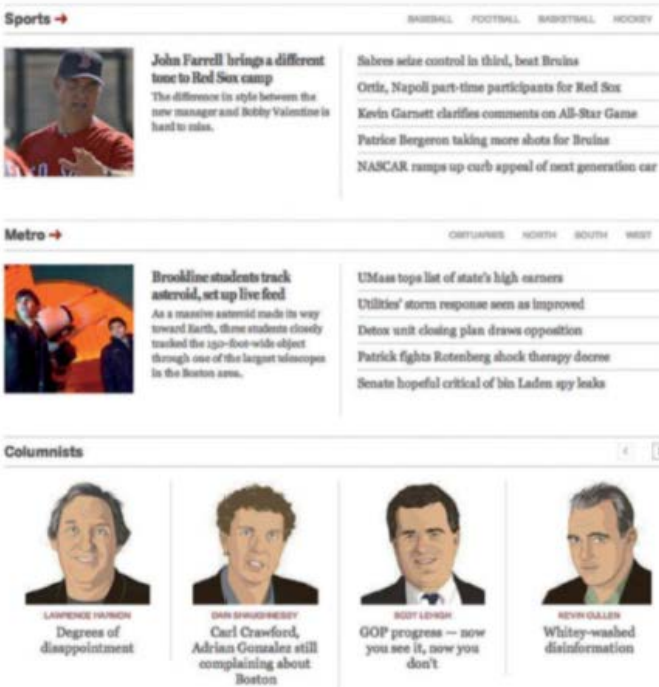


FIG 4.1 : Blocs de section sur la page d'accueil du Boston Globe

qui démarrent par `data-` et se terminent par le terme de notre choix (par exemple `data-foo`).

Nous avons conçu le script Ajax-Include de sorte qu'il recherche des attributs `data` spécifiques qui lui indiquent de rapatrier et d'incorporer du contenu : `data-append`, `data-replace`, `data-after` et `data-before`. Ces noms d'attributs demandent au script de greffer le contenu à un endroit particulier une fois qu'il a été rapatrié :

- `data-append` ajoute le contenu à la fin de l'élément référençant.
- `data-replace` remplace l'élément référençant par le contenu injecté.
- `data-before` et `data-after` injectent le contenu avant ou après l'élément référençant.

En ajoutant l'un de ces attributs HTML5 personnalisés, `data-after`, dans le lien de notre section Sports, nous pouvons référencer une URL qui contient le contenu de ce bloc et demander à JavaScript de télécharger ce contenu et de l'insérer sur la page.

```
<a href="/sports" data-after="/sports/ »
  homepage-well/">Sports</a>
```

Nous devons ensuite inclure le JavaScript. Si vous consultez l'article du Filament Group sur Ajax-Include, vous trouverez un lien pour télécharger le fichier source du script hébergé sur GitHub (http://bkaprt.com/rrd/4-05/). Ajax-Include dépend également de jQuery (ou d'un framework offrant une syntaxe similaire), alors vous devez référencer à la fois jQuery et Ajax-Include pour que cela fonctionne.

```
<!-- références à jQuery et Ajax-Include -->
<script src="jquery.js"></script>
<script src="ajaxinclude.js"></script>
```

Après avoir référencé les fichiers JavaScript nécessaires, nous ajoutons une ligne de JavaScript (dans la syntaxe de jQuery) afin de demander à Ajax-Include d'exécuter sa logique sur les éléments dont nous avons besoin. Par exemple, cette ligne de jQuery demande au navigateur de trouver tous les éléments de la page comportant un attribut `data-after` et de leur appliquer le plug-in `ajaxInclude`.

```
$( "[data-after]" ).ajaxInclude();
```

FIG 4.2 : Contenu initial et contenu après l'exécution d'Ajax-Include

Pour configurer tous les usages d'Ajax-Include dont je pourrais avoir besoin sur un site avec une seule commande, j'ajoute généralement les autres sélecteurs disponibles dans la commande :

```
$( "[data-after],[data-before],[data-replace], »
  [data-append]" ).ajaxInclude();
```

Pour en revenir à notre exemple de blocs de contenu, la commande ci-dessus produit un effet « avant-après » sur la page (FIG 4.2).

Les avantages de cette approche sont persuasifs. Il est vrai qu'elle introduit des requêtes HTTP supplémentaires, mais ces requêtes sont effectuées une fois la page initiale rendue et utilisable, de sorte que les utilisateurs peuvent interagir avec certaines parties de la page un peu plus tôt. Le script peut également être optimisé pour inclure plusieurs morceaux de contenu dans une seule requête si besoin est, voire rapatrier le contenu sous forme de données structurées si votre API vous permet de telles subtilités. Mais le plus intéressant, c'est peut-être encore de pouvoir inclure du contenu - ou non - sur la base de diverses conditions, puisqu'il est déjà à portée de clic pour tous les utilisateurs.

Charger pour certains points de rupture

Ajax-Include n'est pas nécessairement été conçu pour servir du contenu différent à divers appareils. Cela dit, vous pouvez également lui indiquer de rapatrier un certain contenu uniquement lorsqu'une media query s'applique. Pour cela, spécifiez un attribut `data-media` avec une valeur de media query, et ce contenu sera uniquement récupéré si la condition de la requête devient valide (au moment du chargement ou par la suite). Voici un exemple d'Ajax-Include qui s'applique lorsque la largeur du viewport dépasse `35em`. Les viewports plus petits recevront un lien vers la page Sports, où ils trouveront de toute façon le même contenu.

```
<a href="/sports" data-after="/sports/homepage-well/" »
  data-media="(min-width: 35em)">Sports</a>
```

Ordre des sources et responsive design

Un autre problème que pose HTML pour construire des sites responsive complexes, c'est qu'il est souvent difficile de parvenir à une mise en page particulière à cause de l'ordre des éléments dans le fichier HTML, également appelé ordre des sources. Jusqu'à maintenant, les possibilités de mise en page en CSS avec des outils traditionnels tels que `float` et `clear` étaient restreintes à cause de l'ordre des sources HTML ; ce n'est que récemment que des outils comme CSS Flexbox sont apparus pour nous donner plus de contrôle. Néanmoins, étant donné que ces nouvelles approches ne fonctionnent que dans les navigateurs les plus récents, il peut être judicieux d'avoir un plan B - idéalement, un plan qui n'implique pas d'utiliser la détection d'appareils pour servir du code différent à différents appareils, ni de répéter le même balisage à plusieurs emplacements pour afficher ou masquer du contenu.

AppendAround démêle votre mise en page

Si nous avons le balisage qui convient, mais qu'il n'est pas au bon emplacement dans notre document pour obtenir une mise en page particulière, nous pouvons utiliser JavaScript pour le déplacer ailleurs dans le code HTML. AppendAround

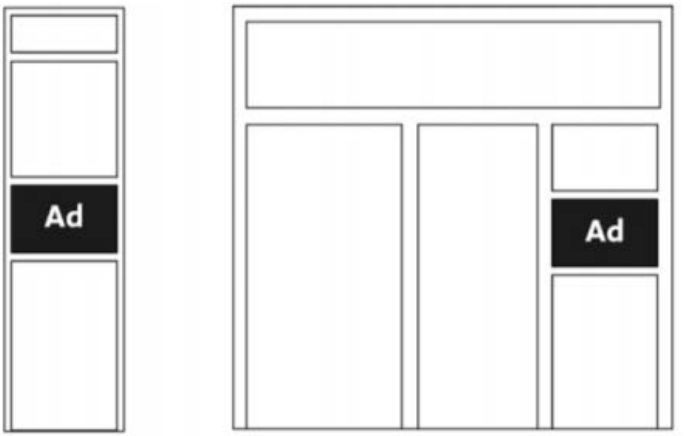


FIG 4.3 : Exemple d'utilisation d'AppendAround pour déplacer une unité publicitaire dans le DOM

(http://bkaprt.com/rrd/4-06/) est une solution que nous avons développée et utilisée pour le site du Boston Globe. Voyons plutôt un exemple : le wireframe suivant illustre une publicité qui doit s'afficher à différents emplacements de la page selon la taille du viewport (FIG 4.3).

À cause des contraintes dues à l'ordre des sources, il n'est pas toujours possible de placer le contenu à ces deux endroits avec du CSS traditionnel : sur les petits écrans, la publicité doit être placée plus haut dans l'ordre linéaire des sources afin que les utilisateurs la voient dès qu'ils font défiler la page, tandis sur un écran plus grand, la publicité doit être placée à mi-chemin dans la colonne de droite, bien plus tard dans l'ordre des sources et à la suite d'un bloc de texte dont la hauteur est susceptible de varier.

Avec la technique AppendAround, nous pouvons automatiquement déplacer la publicité d'un endroit à un autre dans le DOM en fonction du point de rupture CSS actif. La FIG 4.4 montre comment cela fonctionne avec un bloc de contenu basique.

Le code HTML pour AppendAround est plutôt simple. Si vous souhaitez qu'un bloc de contenu itinérant apparaisse à un

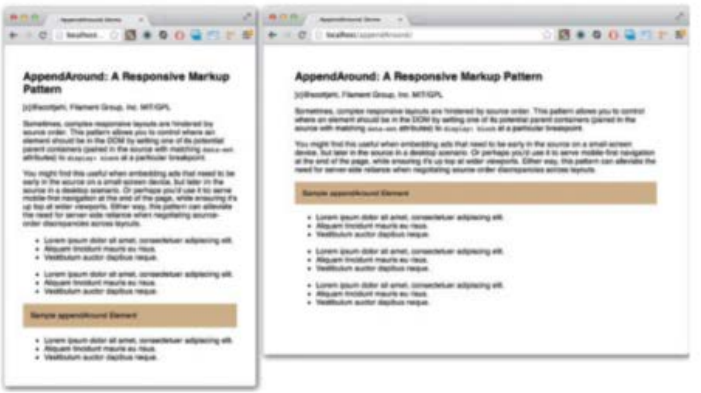


FIG 4.4 : Exemple de déplacement d'un bloc de contenu avec AppendAround (http://bkaprt.com/rrd/4-07/)

certain endroit dans le document, créez un conteneur `div` vide avec un attribut `data-set` prenant le nom de tous les parents potentiels de ce bloc de contenu, et appliquez ce même attribut au parent initial du contenu itinérant. Dans l'exemple ci-dessous, les éléments parents potentiels possèdent l'attribut `data-set="rover-parent"`.

```
<!-- conteneur initial pour appendAround -->
<div class="rover-parent-a" data-set="rover-parent">
  <p class="rover">Exemple de contenu appendAround</p>
</div>

<ul>
  <li>Lorem ipsum dolor sit amet.</li>
  <li>Vestibulum auctor dapibus neque.</li>
</ul>

<!-- conteneur potentiel pour appendAround -->
<div class="rover-parent-b" data-set="rover-parent">
</div>
```

Nous devons ensuite inclure la bibliothèque JavaScript d'AppendAround, puis localiser notre élément sur la page et lui appliquer la méthode `appendAround()`.

```
<script src="jquery.js"></script>
<script src="appendAround.js"></script>
<script>
  /* Appel de la fonction appendAround */
  $( ".rover" ).appendAround();
</script>
```

C'est là que les choses deviennent intéressantes. Tout ce qu'il nous reste à faire, c'est de rendre l'un des conteneurs potentiels visibles pour un point de rupture donné en CSS et l'élément `.rover` sera greffé à cet élément. Lorsque la page se chargera, et chaque fois que le viewport sera redimensionné, le script s'exécutera pour vérifier si un élément AppendAround est masqué. Dans ce cas, il essaiera de trouver un parent potentiel visible et y greffera le contenu. Voici le code CSS :

```
.rover-parent-a {
  display: block;
}
.rover-parent-b {
  display: none;
}

@media ( min-width: 30em ){
  .rover-parent-a { display: none; }
  .rover-parent-b { display: block; }
}
```

Et voilà ! Grâce à cette technique, nous pouvons déplacer un bloc de contenu dans le DOM simplement en CSS. Une mise en garde, toutefois : évitez d'utiliser AppendAround sur des blocs de contenu critiques de grande taille, car il peut parfois réorganiser la page autour du nouvel élément greffé (un effet que nous voulons minimiser). Idéalement, nous devons nous efforcer d'épuiser toutes les options avec CSS seul avant de recourir

à des bricolages comme celui-ci. Mais il est tout de même utile d'avoir cette possibilité au cas où.

Maintenant que nous avons évoqué quelques approches pour réduire la taille de transfert de notre code HTML, passons aux ressources référencées. Et pour commencer : CSS.

TRANSMETTRE DU CSS

Parmi les ressources front-end, ce sont les requêtes CSS qui ont le plus gros impact sur la durée du rendu d'une page (http://bkaprt.com/rrd/4-08/). Malheureusement, notre CSS ne fera que s'alourdir à mesure que le nombre d'écrans et de conditions auxquels nous devrons répondre s'accroîtra.

Si le CSS offre de nombreux moyens de qualifier l'application de styles particuliers (media queries, classes conditionnelles, règles `@supports`), il manque pour le moment des mécanismes permettant de qualifier sa transmission à des environnements spécifiques. Néanmoins, nous pouvons prendre des mesures pour réduire autant que possible le CSS superflu et préparer la transmission de notre CSS de manière à améliorer la performance perçue.

Tout est dans l'en-tête

Comme nous l'avons appris au chapitre précédent, tous les styles requis pour la mise en page initiale doivent être référencés dans la balise `head` de la page ; autrement, nous risquons d'obtenir un flash de contenu sans style au cours du chargement de la page. Il existe plusieurs moyens de référencer des styles externes dans la balise `head`.

Approche A : une grosse feuille de style contenant toutes les media queries

L'approche la plus courante pour transmettre du CSS responsive consiste à concaténer tout le CSS dans un seul fichier et à qualifier les styles à l'aide de media queries qui s'appliquent sous diverses conditions. Le code HTML ressemble alors à cela :

```
<head>
...
<link href="all.css" rel="stylesheet">
...
</head>
```

Le CSS de cette feuille de style est des plus classiques :

```
/* d'abord, quelques styles généraux pour tous les
contextes */
body {
  background: #eee;
  font-family: sans-serif;
}
/* ensuite, des styles qualifiés pour des médias
particuliers */
@media (min-width: 35em){
  ...styles pour les viewports de 35em (~560px) de
largeur et plus
}
@media (min-width: 55em){
  ...styles pour les viewports de 55em (~880px) de
largeur et plus
}
```

Tout d'abord, les avantages. En combinant tout le CSS dans un seul fichier, une seule requête HTTP bloquante sera nécessaire pour récupérer le fichier, et la réduction du nombre de requêtes bloquantes est l'un des meilleurs moyens d'accélérer le transfert de la page et de réduire les points de dysfonctionnement potentiels. Par ailleurs, en ayant à disposition tous les styles potentiellement applicables, le navigateur peut appliquer les styles dès que les conditions changent, par exemple l'orientation de l'appareil ou la taille de la fenêtre.

L'inconvénient, c'est que cette approche risque d'accroître inutilement le temps de chargement de la page et de consommer plus de données que nécessaire en exigeant des utilisateurs qu'ils téléchargent des styles qui ne s'appliqueront peut-être jamais dans leur navigateur ou sur leur appareil. Selon le poids

global et l'impact sur la performance perçue du code généré avec cette approche, il peut être judicieux d'envisager une solution alternative.

Tout bien considéré, la syntaxe redondante de CSS lui permet de se compresser particulièrement bien avec Gzip, ce qui aide à réduire la charge superflue apportée par les styles inapplicables.

Approche B : fichiers séparés pour chaque média

Une seconde méthode pour charger du CSS responsive consiste à placer les styles de chaque média dans des fichiers séparés et à transférer ces fichiers de façon indépendante. Pour spécifier dans quelles conditions chaque feuille de style doit s'appliquer, nous ajoutons un attribut `media` avec la valeur de la media query souhaitée dans chaque élément `link`. Ces attributs `media` fonctionnent comme les media queries inline : vous pouvez donc retirer celles-ci de vos fichiers CSS et les styles s'appliqueront tout de même comme prévu.

```
<head>
...
<link href="shared.css" rel="stylesheet">
<link href="medium.css" media="(min-width: 35em)" »
  rel="stylesheet">
<link href="large.css" media="(min-width: 55em)" »
  rel="stylesheet">
</head>
```

Les avantages et les inconvénients de cette approche dépendent du navigateur. Je commencerai par l'inconvénient : si vous espérez que tous les navigateurs ignoreront les feuilles de style qui ne correspondent pas à leurs conditions, vous risquez d'être déçu. À l'heure actuelle, les navigateurs les plus courants téléchargeront toutes les feuilles de style référencées dans un document HTML, quel que soit l'attribut `media` (http://bkaprt.com/rrd/4-09/).

Si vous espériez gagner quelques octets avec cette approche, c'est raté. D'autre part, nous avons un autre problème : nous avons ajouté deux requêtes HTTP bloquantes pour charger les

mêmes styles que précédemment. Et pour couronner le tout, chaque fichier doit être compressé séparément pour être transféré, ce qui signifie que la taille cumulée des fichiers CSS sera vraisemblablement plus élevée.

Toutefois, il y a bien un point positif. Plusieurs navigateurs modernes, tels que Safari (Mac et iOS), Opera et Chrome, évalueront l'attribut `media` d'un élément `link` pour vérifier si ses conditions s'appliquent à l'environnement de navigation actuel, puis se serviront des ces informations pour élever ou abaisser la priorité de la requête de cette feuille de style. Les requêtes de faible priorité ne bloqueront pas le rendu de la page, et même si toutes les feuilles de style seront au final téléchargées, la page sera rendue dès que toutes les feuilles de style applicables auront fini de se charger, permettant aux autres feuilles de style d'arriver à leur rythme. Les FIG 4.5 et 4.6 illustrent la différence entre l'approche traditionnelle et une approche plus moderne permettant de charger les feuilles de style applicables en priorité.

Ce comportement émergent des navigateurs peut mériter d'être utilisé en fonction de plusieurs facteurs. Si de grosses portions de votre CSS ciblent des environnements ou des points de rupture particuliers, cette approche peut permettre à votre site de se charger plus vite dans certains navigateurs que si vous ne regroupiez tous les styles dans un seul fichier CSS. Cependant, étant donné que de nombreux navigateurs répandus ne sont pas encore capables de considérer les requêtes de feuilles de style inutiles comme moins prioritaires, il peut être préférable de s'en tenir à l'approche A. La seule façon de s'en assurer, c'est de faire des tests dans de véritables navigateurs et de comparer les résultats.

Approche C : inclure tout le CSS dans le code HTML

Une troisième approche consiste à placer le CSS directement dans le document HTML lui-même :

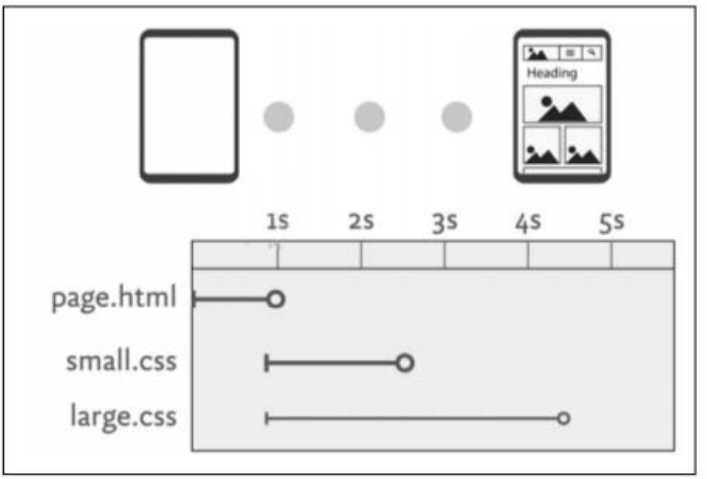


FIG 4.5 : Chronologie simplifiée d'une requête pour un navigateur qui n'attribue pas une priorité moindre au CSS inapplicable

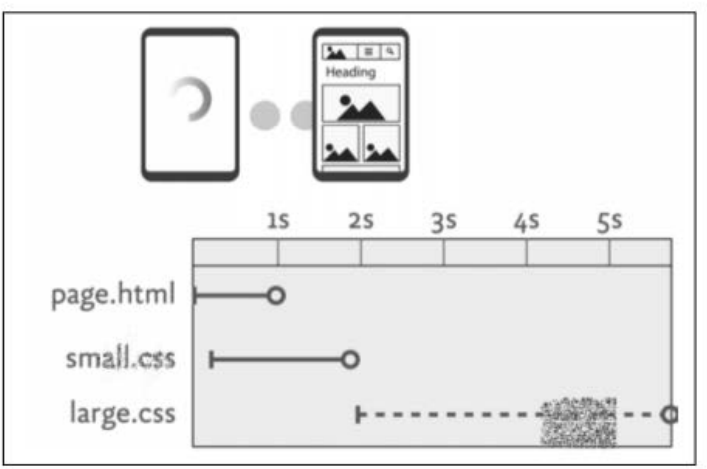


FIG 4.6 : Chronologie simplifiée d'une requête pour un navigateur qui attribue une priorité moindre au CSS inapplicable

```
<head>
  ...
  <style>
    /* d'abord, quelques styles généraux pour tous les
    contextes */
    body {
      background: #eee;
      font-family: sans-serif;
    }
    /* ensuite, des styles qualifiés pour des médias
    particuliers */
    ...
  </style>
</head>
```

Les avantages et les inconvénients de cette approche sont plutôt évidents. Comme pour l'approche A, le fait d'inclure tout le CSS à un seul endroit permet de mieux le compresser pour le transfert. De plus, l'approche C nous permet de charger tout notre CSS sans avoir à émettre une requête HTTP supplémentaire, ce qui permettra probablement de charger la page un peu plus vite lors de la première visite. L'inconvénient, c'est que l'inclusion des styles dans le code HTML empêche le navigateur de mettre en cache ces styles en tant que ressource séparée pour le chargement d'autres pages, de sorte que les mêmes styles seront chargés à chaque fois qu'une nouvelle page sera consultée.

Je recommande de réserver cette stratégie aux sites web ne comportant qu'une seule page, à ceux dans lesquels le CSS de chaque page est différent (ce qui est inhabituel), et à ceux dont le CSS complet est minime (disons moins de 8 Ko).

Quelle est la meilleure approche ?

C'est compliqué.

Pour les sites responsive dont la majorité des règles CSS sont partagées par les différents navigateurs et appareils, l'approche A est généralement la manière la plus responsable de transmettre le CSS. Cependant, à mesure que nous continuerons à ajouter du CSS pour des types d'appareils extrêmement différents et que les navigateurs s'amélioreront, nous nous apercevrons

probablement que l'approche B est une meilleure option. Malheureusement, les approches A et B nécessitent toutes deux des requêtes externes bloquantes, ce que l'approche C permet d'éviter – au prix de la mise en cache.

Il doit bien y avoir un meilleur moyen ! Une solution hybride combinant les approches B et C serait optimale...

La meilleure approche est hybride

Une tactique de plus en plus répandue pour améliorer la performance perçue consiste à optimiser la page pour le tout premier aller-retour entre le navigateur et le serveur, qui transporte environ 14 Ko de données. Si vous parvenez à faire tenir tout le code HTML, CSS et JavaScript nécessaire pour rendre le contenu situé au-dessus de la ligne de flottaison (une mesure imprécise désignant la portion supérieure de la page) au cours de ce premier aller-retour, vous êtes pratiquement sûr d'atteindre ce temps de chargement perçu d'une seconde auquel nous aspirons tous. Pour atteindre cet objectif, l'outil PageSpeed Insights de Google (http://bkaprt.com/rrd/4-10/) recommande de n'inclure que le CSS strictement nécessaire pour rendre le contenu dans la vue initiale dans le code HTML, et de charger le reste d'une manière non bloquante. Bien sûr, la ligne de flottaison varie d'un écran à l'autre, et il peut être difficile de déterminer avec certitude quelles parties de notre CSS sont critiques et quelles parties ne le sont pas. Une approche possible consiste à organiser nos feuilles de style dans un certain ordre – du haut vers le bas, de l'extérieur vers l'intérieur – en fonction de l'endroit où les composants sont placés dans la page. Nous pourrions commencer par inclure l'essentiel de la mise en page pour la portion supérieure de la page dans le code HTML, et déterminer un point à partir duquel nous référençons le reste des ressources de manière externe et non bloquante.

Admettons que nous ayons une mise en page dans laquelle, pour tous les points de rupture, le sommet de la page comprend un en-tête, une barre de navigation et un extrait du contenu, avec d'autres composants tels que du contenu secondaire et un pied de page en dessous. Dans ce cas, le code CSS que nous inclurons dans la balise head de notre page pourra ressembler à ça :

```
<head>
  ...
  <style>
    /* Les styles CSS critiques pour ce template sont
    placés ici... */
  </style>
  ...
</head>
```

Une fois ce code en place, nous n'avons plus de requêtes CSS bloquantes dans le head de la page et nous pouvons obtenir une bonne partie du sommet de la page lors de ce premier aller-retour de 14 ko.

Il peut être difficile de gérer manuellement les fichiers CSS ainsi, et c'est pourquoi je vous conseille d'utiliser un outil pour bien faire le travail. Dans son article intitulé « Detecting Critical Above-the-fold CSS », Paul Kinlan offre un bookmarklet que vous pouvez exécuter sur n'importe quelle page pour extraire ses styles critiques (http://bkaprt.com/rrd/4-11/). La logique de Kinlan est simple : le CSS critique est l'ensemble des règles CSS nécessaires pour rendre la portion supérieure d'une page dans une taille de viewport donnée. Pour les sites responsive, j'ai l'habitude d'exécuter ce bookmarklet sur un viewport large, disons 1 200 pixels par 900, afin de capturer les styles nécessaires pour rendre les nombreux points de rupture d'une mise en page responsive.

Les bookmarklets sont utiles, mais pour un codebase de plus grande envergure, vous aurez besoin de plus d'automatisme. À cette fin, mon collègue Jeff Lembeck et moi-même avons développé un outil appelé Grunt-CriticalCSS (http://bkaprt.com/rrd/4-12/) qui extrait automatiquement le CSS critique de chaque template et le copie dans un fichier qui peut être inclus dans le code HTML. Lorsqu'il est correctement configuré, Grunt-CriticalCSS s'exécute en arrière-plan de manière invisible à chaque fois que vous changez un fichier CSS, de sorte que vos fichiers CSS critiques sont toujours à jour.

Quel que soit l'outil que vous utilisiez pour générer votre CSS critique, une fois celui-ci généré, vous devrez l'inclure directement dans l'élément head de la page. Quant au CSS complet

du site, vous devrez le charger de manière non bloquante aussi rapidement que possible.

Pour cela, vous pouvez utiliser une fonction JavaScript appelée `loadCSS` (http://bkaprt.com/rrd/4-13/), qui charge les fichiers CSS de manière asynchrone de façon à ce qu'ils ne bloquent pas le rendu de la page. Dans la continuité de notre objectif consistant à éliminer les requêtes bloquantes, `loadCSS` est assez petit pour être inclus dans l'élément `head` du HTML. Par ailleurs, je vous recommande de placer le script qui contiendra `loadCSS` après votre élément `style`, car cet ordre permet à la fonction JavaScript d'insérer le CSS complet du site après le CSS inline, évitant ainsi les conflits de spécificité potentiels. L'approche globale devrait ressembler à cela :

```
<style>
  /* les styles CSS critiques pour ce template sont
  placés ici */
</style>
<script>
  // d'abord, inclure la fonction loadCSS
  function loadCSS( href ){ ... }
  // puis référencer une feuille de style pour la charger
  loadCSS( "full.css" );
</script>
```

Je recommande d'ajouter un lien vers le CSS complet du site après cette dernière balise `style` au cas où JavaScript ne serait pas disponible. Voilà à quoi cela ressemble :

```
<noscript><link href="full.css" rel="stylesheet">
</noscript>
```

Voilà un moyen d'optimiser le chargement de votre CSS qui est certes un peu compliqué, mais qui en vaut la peine. Nous revisiterons cette approche à la fin du chapitre où nous la combinerons à d'autres pour optimiser complètement une page.

Quelle que soit l'approche que vous choisirez pour transmettre votre CSS, efforcez-vous de le rédiger de manière aussi concise que possible et de tirer parti de la cascade pour réduire

les répétitions. Minifiez (supprimez les espaces blancs et les commentaires à l'aide d'un outil tel que Grunt-CSS) systématiquement chaque fichier CSS et activez la compression Gzip pour le transfert de tous vos fichiers CSS externes.

TRANSMETTRE DES IMAGES

En matière de taille de fichier, les images sont les pires contrevenants. Sur ce site web moyen de 1,7 Mo, les images représentent jusqu'à 61 % du poids. Et le problème ne cesse d'empirer à mesure que les tailles d'appareil et les résolutions se diversifient.

Heureusement, contrairement à CSS et JavaScript, les images sont téléchargées de façon asynchrone par défaut par tous les navigateurs, c'est-à-dire sans bloquer le rendu de la page. Mais si les requêtes d'image inachevées ne bloquent pas le rendu de la page, elles causent tout de même de sérieux problèmes de performance. La plupart de ces problèmes sont tout simplement dus à la taille des images, qui se chargent lentement et peuvent venir à bout des forfaits de données les plus généreux.

Pour commencer notre exploration du chargement d'image responsive et responsable, commençons par voir la différence entre les images d'arrière-plan et les images de premier plan.

Images d'arrière-plan

Même lorsqu'elles sont incluses à l'arrière-plan via CSS, les images créent des requêtes HTTP supplémentaires. Par exemple, la règle suivante demande au navigateur de télécharger l'image `foo.jpg` et de la rendre à l'arrière-plan de tous les éléments comportant la classe `foo`.

```
.foo {
  background: url(foo.jpg);
}
```

C'est plutôt clair. Les choses se corsent un peu lorsque nous voulons charger différentes images d'arrière-plan à l'aide de media queries. Vu le peu d'options exposées dans la section CSS, vous serez peut-être surpris d'apprendre qu'il est en fait

plutôt simple de charger des images d'arrière-plan de manière responsable avec CSS et les media queries. Une étude de Tim Kadlec a démontré que la plupart des navigateurs activement utilisés, lorsqu'on leur présente deux règles `background-image` appliquées au même élément, iront chercher la dernière image référencée (http://bkaprt.com/rrd/4-14/). Cela marche également à l'intérieur des media queries.

Dans l'exemple suivant, les navigateurs dont la largeur du viewport est supérieure ou égale à `30em` téléchargeront et afficheront l'image `foo-large.jpg`, tandis que les navigateurs plus petits iront chercher `foo.jpg`.

```
.foo {
  background: url(foo.jpg);
}

@media (min-width: 30em){
  .foo {
  background: url(foo-large.jpg);
}
```

Améliorer les images d'arrière-plan pour les écrans HD

Vous pouvez utiliser cette approche pour toutes les conditions prises en charge par les media queries, et vous pouvez donc facilement l'utiliser pour « améliorer » les images à l'attention des écrans HD. La media query `min-resolution` nous permet de cibler les appareils dont la résolution en dpi est égale ou supérieure à une certaine valeur (des préfixes vendeurs tels que `-webkit-min-device-pixel-ratio` permettent là encore d'étendre la prise en charge à différents navigateurs). Dans cet exemple, j'ai utilisé `144dpi` car il s'agit du double de la résolution standard de 72 points par pouce, une bonne base de départ pour les écrans HD (qui dépassent aujourd'hui bien souvent les 144 dpi).

```
.foo {
  background: url(foo.jpg);
}
```

```
@media (min-resolution: 144dpi){
  .foo {
    background: url(foo-large.jpg);
    background-size: 50px 50px;
  }
}
```

Notez que la propriété `background-size` est incluse avec l'image plus large pour spécifier que l'image devra être rendue à une taille qui peut différer de ses dimensions inhérentes. Dans ce cas, nous voulons qu'une image plus large tienne dans le même espace physique que l'image de résolution normale, afin d'offrir une plus grande densité de pixels et une image plus riche. Vous constaterez peut-être que la version deux fois plus grande de certaines images est déjà suffisamment légère et responsable et qu'aucune négociation n'est nécessaire (c'est parfois le cas des illustrations comportant peu de couleurs et de dégradés, par exemple). Dans ce cas, il peut convenir d'envoyer l'image de grande taille à tous les appareils en spécifiant un attribut `background-size`.

URI de données

Une autre option est le schéma d'URI `data`, qui incorpore les données d'une image (ou de tout autre fichier) directement dans une chaîne de caractères que vous pouvez utiliser à la place d'une référence externe à ce fichier, ce qui vous évite d'émettre une requête supplémentaire au serveur pour obtenir cette ressource. Voici l'exemple d'une image de flèche encodée dans une URI de données :

```

YAAADd/14OAAAABHNCSVQICAgIfAhkiAAAAAlwSFlzAAALEwAACxMBAJ
qcGAAAAUFJREFUKJGV0C9MAnEUB/Dve3eMIptszuL43XE/1GgjOCduZp
3RrDSjVYPFzWR0jtnMMKPB2ZwUk8WheIdAcWM08QbvZzhRQAi+9rbP3p
+vlclk4tOJRHEmmWy22u0AE4pNGJ4TaLNrzI1W6fxkyHwrIiGDbJApaO
We5pCzRyEBgFZqGQYlMM8CgDFy3SPaDoKgPQQBwHVdx+rJFZiXAAAiT0
bsjWqjWgEA7kPf94N4Z2oFglJ0FC+SLWVPqfWhiYN3a8c5Aujgd63ZGw
ehlErGiFv93gBNHkXenDcfA+77vQAPVq+bHYLacdbIljKIF75VMdHSWK
```

FIG 4.7 : Vous voyez cette flèche ? Cette image a été créée à partir d'une URI de données.

```
3U642fvLRK5wVyxoAdRYTj6pt/GA2NnrG8lHtCjP0oFQktsnafa6+Xg9
tIp9wLMHYAQIy8M7D1Uqvd/YmC2BRE5BMwj0KUHYf+VV8xa3TEn/anuA
AAAABJRU5ErkJggg==
```

Si cela vous paraît illisible, c'est le cas - pour nous autres humains, en tout cas. Mais attendez ! Copiez ce texte dans la barre d'adresse d'un navigateur, et vous verrez quelque chose qui ressemble à la FIG 4.7.

La syntaxe d'une URI de données peut être décrite simplement : elle commence toujours par `data:`, qui indique au navigateur que l'URL elle-même contient les données d'un fichier, puis deux ou trois informations sur le type de fichier (dans ce cas, un fichier de type `image/png` encodé en base64) séparées par des points-virgules, puis une virgule, et enfin les données brutes de ce fichier :

```
data:[<type-MIME>][;charset=<encodage>]
[;base64],<données>
```

Cette approche s'avère particulièrement utile lorsqu'on l'applique dans notre codebase au lieu de référencer des images externes. Par exemple, voici comment on référencerait cette URI comme image d'arrière-plan CSS (données tronquées pour votre santé mentale) :

```
.menu {
  background: url( "..." );
}
```

Vous pouvez également inclure la source brute de fichiers texte dans une URI de données. Voici un fichier SVG :

FIG 4.8 : L'outil de conversion de fichiers en URI de données créé par Boaz Sender (http://bkaprt.com/rrd/4-15/)

```
.header {
  background: url("data:image/svg+xml, »
    <svg viewBox='0 0 40 40' height='25' width='25' »
    xmlns='http://www.w3.org/2000/svg'> »
    <path fill='rgb(91, 183, 91)' d='M2.379,14.729L5 »
    .208,11.899L12.958,19.648L25.877,6.733L28.707, »
    9.561L12.958,25.308Z'/></svg>");
}
```

Il existe de nombreux outils permettant de convertir des fichiers en URI de données, mais le plus simple que j'ai trouvé est une petite application web créée par Boaz Sender de Boucoup, une agence de conseil web de Boston, qui fonctionne avec un simple glisser-déposer (FIG 4.8).

Contrairement aux références ordinaires à des ressources externes, les URI de données accélèrent les performances car il n'est pas nécessaire d'envoyer une requête supplémentaire pour télécharger la ressource. Mais en incluant les données d'un fichier dans le code HTML, il n'est plus possible de télécharger le fichier uniquement lorsqu'il est requis (par exemple, une image d'arrière-plan externe qui s'applique à une media query spécifique). De plus, l'abus d'URI de données peut causer des problèmes sur certains appareils mobiles (http://bkaprt.com/rrd/4-16/). Pour ces raisons, réservez les URI de données aux ressources universelles qui s'appliquent à tous les appareils et à tous les points de rupture.

Dernière remarque : les URI de données ne sont pas limitées aux images d'arrière-plan ; vous pouvez également les utiliser pour les images de premier plan.

Images de premier plan responsive et responsables

Les images de premier plan comprennent toutes les images référencées dans le code HTML qui sont destinées à faire partie du contenu : les images qui apportent du sens à une page web, comme une photo associée à un article, et non les images servant de décoration visuelle comme les icônes ou les textures d'arrière-plan.

Comme vous vous en souviendrez sans doute, l'un des principes fondamentaux du responsive design consiste à ajouter la règle CSS suivante pour s'assurer que tous les éléments `img` d'une page occupent 100 % de la largeur de leur conteneur sans dépasser les dimensions propres de l'image :

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

Comme les images ne peuvent pas être agrandies sans perdre en qualité, les web designers incluent bien souvent les images dans leur taille d'affichage maximale et chargent le navigateur de les réduire pour les viewports plus petits. Malheureusement, cette pratique consistant à fournir des images de grande taille à tout le monde n'est pas très responsable - les utilisateurs finissent souvent par charger beaucoup plus de données que de nécessaire pour les besoins de leur appareil.

Les problèmes que nous rencontrons pour servir des images de premier plan de manière responsable sur divers appareils dérivent en grande partie de l'incapacité d'HTML (jusqu'à récemment) à servir différentes versions d'une image selon la taille requise par l'appareil. Par chance, nous avons aujourd'hui les outils qui nous permettent d'y répondre.

Images compressives

Si vous n'avez besoin de rien de plus qu'une seule image redimensionnable, vous pouvez envisager une technique intéressante développée par Daan Jobsis et que j'ai surnommée images « compressives » (http://bkaprt.com/rrd/4-17/) (FIG 4.9). Cette

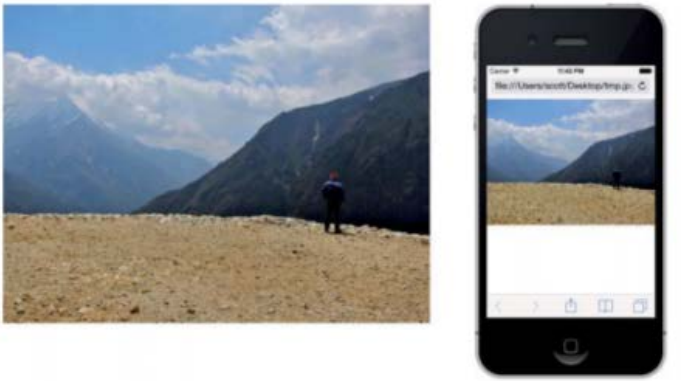


FIG 4.9 : La même image en qualité « compressive » (à gauche) et telle qu'elle s'affiche une fois redimensionnée par le navigateur (à droite)

approche consiste à sauvegarder des images JPEG deux fois plus grandes que les dimensions d'affichage prévues avec des paramètres de qualité médiocre, puis à laisser le navigateur redimensionner les images pour éliminer les artefacts. Aussi incroyable que cela puisse paraître, cette technique réduit la taille des fichiers de moitié tout en améliorant la netteté sur les écrans HD. Il faut le voir pour le croire, alors n'hésitez pas à consulter le lien.

Les images compressives présentent également des inconvénients : le redimensionnement d'images de grande taille utilise pas mal de puissance de traitement et de mémoire, et cette approche se prête mal à une utilisation à grande échelle. Elle est également limitée dans la mesure où la taille des fichiers image doit augmenter pour s'adapter aux résolutions plus élevées, aux dépens des écrans de plus faible résolution qui n'ont pas besoin de ces données supplémentaires. Les images compressives sont adaptées aux cas les plus simples. Abordons maintenant une solution d'image responsive plus complète.

FIG 4.10 : Le site web Responsive Images Community Group du W3C (http://bkaprt.com/rrd/4-18/)

Images responsive avec HTML

En 2012, le W3C a créé le Responsive Images Community Group (RICG), présidé par Mat Marquis, pour définir les cas d'utilisation auxquels une solution d'image responsive idéale devrait répondre et pour recommander de nouvelles fonctionnalités HTML permettant d'implémenter ces images dans les navigateurs (FIG 4.10).

Le groupe a proposé l'élément `picture` et ses attributs associés tels que `srcset`, `sizes`, `media` et `type`, qui, par bonheur, sont devenus de véritables normes du W3C et sont pris en charge par les navigateurs modernes à l'heure où j'écris ce livre. Ces nouvelles fonctionnalités présentent d'énormes avantages : voyons comment nous pouvons les utiliser aujourd'hui.

L'élément picture

D'après la spécification, « l'élément `picture` est un conteneur d'image dont le contenu source est déterminé par une ou plusieurs media queries CSS » (http://bkaprt.com/rrd/4-19/). `picture` est un nouvel élément HTML, ou plutôt une série d'éléments,

comprenant son lot d'attributs propres. L'utilisation des media queries avec l'élément `picture` permet de servir facilement des variantes d'une image correspondant à des points de rupture visuels dans une mise en page CSS. C'est particulièrement utile si vous avez des images de premier plan à redimensionner de concert avec d'autres éléments d'une mise en page.

Un élément `picture` contient une série d'éléments `source` suivis d'un élément `img`. Si cela vous semble familier, c'est parce que `picture` utilise une syntaxe similaire aux éléments HTML existants `video` et `audio`, qui comportent un élément `container` et plusieurs éléments `source` qui référencent des sources possibles pour l'élément parent à utiliser. Dans le cas de `picture`, cependant, les éléments `source` servent de contrôleurs pour l'URL qui sera affichée par l'élément `img` associé. Voici un exemple d'élément `picture` avec plusieurs images source potentielles :

```
<picture>
  <source media="(min-width: 45em)" srcset="large.jpg">
  <source media="(min-width: 18em)" srcset="med.jpg">
  <img srcset="small.jpg" alt="...">
</picture>
```

Les éléments `source` sont listés du plus grand au plus petit, avec des attributs `media` spécifiant la taille minimale du viewport à laquelle l'image doit être appliquée. Cet ordre de parsing des éléments `source` peut sembler contre-intuitif si vous avez pris l'habitude de rédiger les media queries pour petits écrans en premier en CSS, mais il est conçu pour correspondre à l'ordre de sélection des sources dans les éléments HTML `video` et `audio`. Le navigateur passera en revue les éléments `source` dans l'ordre où ils apparaissent et s'arrêtera lorsqu'il rencontrera une source avec un attribut `media` correspondant, puis attribuera l'URL spécifiée dans l'attribut `srcset` de l'élément `source` à l'élément `img`. Si aucun élément `source` ne correspond, les attributs de l'élément `img` lui-même (tels que `srcset`) seront utilisés pour déterminer la source.

Qu'est-ce que c'est, srcset ?

Vous vous dites peut-être « `srcset` ressemble fortement à `src` ; quelle est la différence ? » Comme son nom l'indique, `srcset` est un nouvel attribut conçu pour contenir une ou plusieurs URL source pour une image - mais ce n'est pas tout. `srcset` présente un énorme avantage : il demande au navigateur de décider quelle est la ressource la plus appropriée selon les critères qu'il trouve pertinents, comme la taille du viewport, la résolution de l'écran et même la vitesse du réseau ou d'autres conditions environnementales, telles que la quantité de données restantes sur le forfait mobile de l'utilisateur. En d'autres termes, si nous pouvons déclarer plusieurs images potentielles avec `srcset`, le navigateur est autorisé à les traiter comme des suggestions. Cette caractéristique unique de `srcset` est extrêmement pratique, car si vous voulez que vos images s'adaptent à diverses résolutions et tailles d'écran, il devient rapidement lourd de les décrire avec quelque chose de plus prescriptif, comme des media queries.

Les attributs `srcset` peuvent être utilisés soit dans des éléments `source` au sein d'un élément `picture`, soit dans les éléments `img` eux-mêmes - même les éléments `img` qui ne sont pas encadrés par des balises `picture`. À moins que vous n'essayiez d'associer les sources d'une image aux points de rupture des media queries dans une mise en page, vous n'aurez probablement pas besoin d'un élément `picture` du tout. Les valeurs de `srcset` sont séparées par des virgules ; vous pouvez associer une description des dimensions de l'image à chaque valeur (à l'aide d'unités `w` et `h` représentant les dimensions en pixels de l'image source) afin d'aider le navigateur à déterminer quelle image est la mieux adaptée à la taille du viewport et à la résolution de l'écran. Par exemple, voici un attribut `srcset` avec deux sources d'image potentielles :

```
<img srcset="imgs/small.png 400w, »
  imgs/medium.png 800w" alt="...">
```

Cette balise `img` comporte deux URL source, `small.png` (`400px` de large) et `medium.png` (`800px` de large). Il est bien sûr possible d'utiliser `srcset` avec une seule URL d'image et sans

aucune information supplémentaire, comme je l'ai démontré dans mon premier exemple. Mais il y a un avantage à utiliser `srcset` pour spécifier plusieurs images potentielles dans un élément `picture` : offrir des images adaptées aux différents points de rupture du design tout en laissant le navigateur choisir la résolution la plus adaptée à la qualité de l'écran. C'est de la direction artistique avec prise en charge de la HD. Prenez par exemple le balisage suivant ; la première URL listée pour chaque source s'affichera sur un écran de définition standard, alors que la seconde URL (en gras) ne s'appliquera qu'aux écrans haute résolution.

```
<picture>
  <source media="(min-width: 45em)" srcset="large.jpg
    45em, large-2x.jpg 90em"> »
  <source media="(min-width: 18em)" srcset="med.jpg »
    18em, med-2x.jpg 36em"> »
  <img srcset="small.jpg 8em, small-2x.jpg 16em" »
    alt="...">
</picture>
```

C'est peut-être un bon moment pour mentionner les solutions de secours, car l'attribut `srcset` n'est pas encore pris en charge nativement par de nombreux navigateurs. Une première solution consisterait à permettre à l'élément `img` de se replier sur le texte fourni dans l'attribut `alt`, mais la plupart des navigateurs existant aujourd'hui n'obtiendraient pas une image. Si nous voulons que l'image fonctionne plus largement, nous devons soit utiliser un polyfill en JavaScript pour l'attribut `srcset`, soit ajouter un bon vieil attribut `src` dans notre balise `img`. Mais s'il est simple à utiliser et pris en charge par tous les navigateurs, l'attribut `src` a un coût : la plupart des navigateurs iront télécharger l'image listée dans cet attribut même s'ils ne l'utilisent pas au final, un vrai gâchis de bande passante. Pour le moment, cela ne nous laisse que l'option du polyfill, que j'aborderai dans un instant.

Utiliser l'attribut sizes

Si votre réaction en voyant ces exemples est « super, mais j'aimerais bien quelque chose d'un peu plus compliqué », c'est votre jour de chance ! Pour mieux aider le navigateur à choisir les images source des éléments `picture` et `img`, le nouvel attribut `sizes` nous permet de suggérer la taille à laquelle une image doit être rendue pour un point de rupture donné. Comme pour `srcset`, la syntaxe de l'attribut `sizes` est également délimitée par des virgules, chaque valeur comprenant une media query optionnelle et la largeur à laquelle l'image doit être rendue par CSS lorsque cette media query est active. Voici notre exemple d'`img` précédent avec un attribut `sizes` :

```
<img
srcset="imgs/small.png 400w, imgs/medium.png 800w"
sizes="(max-width: 30em) 100%, 50%"
"alt="...">
```

Je vous ai perdu ? Ne vous inquiétez pas, il m'a fallu un peu de temps pour comprendre l'attribut `sizes` moi aussi. Traduisons cet exemple en langage courant :

- `(max-width: 30em) 100%`. Lorsque le viewport mesure `30em` de large ou moins, la largeur de l'image occupera `100%` de la largeur du viewport.
- `50%`. Autrement, si la largeur du viewport est supérieure à `30em`, la largeur de l'image sera fixée à `50%` de la largeur du viewport.

Il est important de noter que ces largeurs ne seront pas directement appliquées à l'image ; elles sont simplement des suggestions permettant au navigateur de rendre l'image aussi proche de ses dimensions voulues que possible, ce qui permet d'éviter les *reflows* à mesure que la page se dessine.

Utiliser picture avec des types d'images différents

Il y a un autre attribut de `picture` dont j'aimerais parler, et il s'agit de `type`. Dans chaque élément `source` au sein d'un élément `picture`, vous pouvez utiliser un attribut de type optionnel pour spécifier un format de fichier. Si le navigateur prend

en charge ce format de fichier, la source sera utilisée. La valeur du type doit être spécifiée dans la syntaxe définie par HTTP ; ainsi, le type d'un fichier SVG serait défini comme `"image/svg+xml"`, tandis que le type d'un format WebP (un nouveau format d'image hautement optimisé qui est de mieux en mieux pris en charge par les navigateurs) serait `"image/webp"`. Voici un exemple de `picture` avec une source offerte aux formats WebP et JPEG :

```
<picture>
  <source media="(min-width: 18em)" srcset="med.webp" »
    type="image/webp">
  <source media="(min-width: 18em)" srcset="med.jpg">
  <img srcset="small.jpg" alt="...">
</picture>
```

Dans les navigateurs qui prennent en charge le format WebP, ce balisage vous fera faire d'énormes économies de bande passante grâce à la compression exceptionnelle de ce format.

Utiliser des images responsive en HTML aujourd'hui

À l'heure où j'écris ces lignes, une poignée de navigateurs - Chrome, Opera, Firefox et quelques autres qui leur emboîtent le pas - ont l'intention de prendre en charge `picture` prochainement. C'est super, mais c'est loin de couvrir tous les navigateurs dont nous devons nous soucier pour servir des images à nos utilisateurs. Pour utiliser les fonctionnalités de `picture` aujourd'hui, il nous faut adopter une approche de transition.

Picturefill est un polyfill léger en JavaScript développé par Filament Group et approuvé par le RICG pour faire fonctionner le nouvel élément `picture` (et les attributs `img`) dans les navigateurs qui ne le prennent pas encore en charge (FIG 4.11). La FIG 4.12 illustre l'effet de son utilisation sur le site web de Microsoft.

Pour inclure Picturefill sur votre site, vous pouvez utiliser le bout de code suivant, qui inclut un « shiv » HTML5 pour l'élément `picture` avant de charger `picturefill.js` de manière non bloquante :

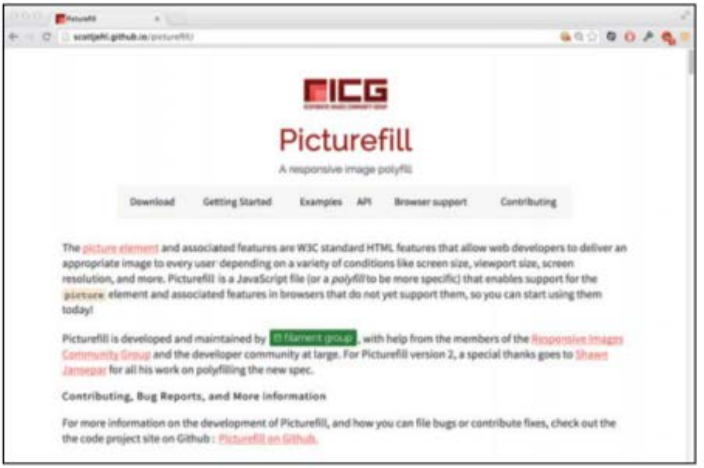


FIG 4.11 : Le projet Picturefill

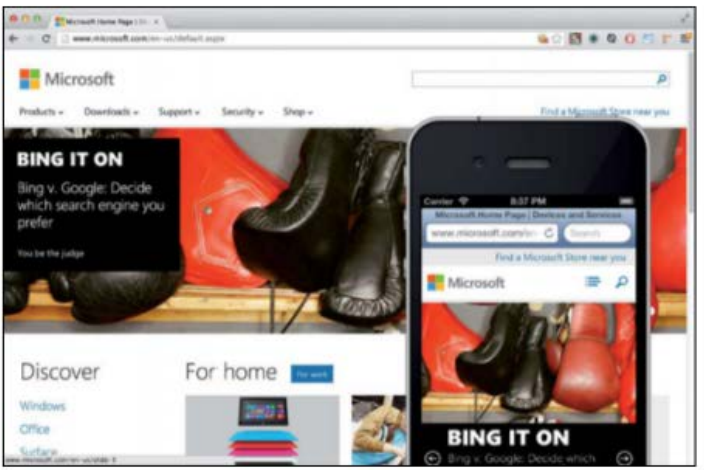
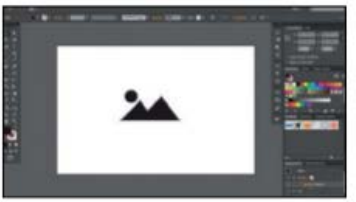


FIG 4.12 : Le site de Microsoft utilise Picturefill pour offrir des images cadrées différemment selon la taille du viewport.

FIG 4.13 : Édition d'image vectorielle



```
<script>
  // Shiv HTML5 pour l'élément picture
  document.createElement( "picture" );
</script>
<script src="picturefill.js" async></script>
```

Vous trouverez de la documentation, des exemples et des informations de compatibilité sur le site du projet Picturefill (http://bkaprt.com/rrd/4-20/). Le site contient également des informations sur la méthode employée par les deux versions de Picturefill pour offrir des images de secours lorsque JavaScript ne peut pas être exécuté, alors pensez à comparer les avantages et les inconvénients de chaque version en fonction de votre public.

ABANDONNER LE PIXEL

Les exemples précédents explorent différents moyens d'échanger et de négocier des images bitmap, car ce format présente des possibilités de redimensionnement limitées. Bien sûr, ce n'est pas le seul format d'image existant, et dans bien des cas ce n'est pas le meilleur outil pour la tâche à effectuer. En raison des différences de taille des viewports et de résolution des écrans, l'idéal serait de pouvoir redimensionner les images sans aucune perte de qualité. Par chance, la plupart des navigateurs modernes prennent en charge plusieurs formats d'image redimensionnables. Explorons maintenant quelques moyens d'implémenter des graphiques vectoriels sur le Web (FIG 4.13).

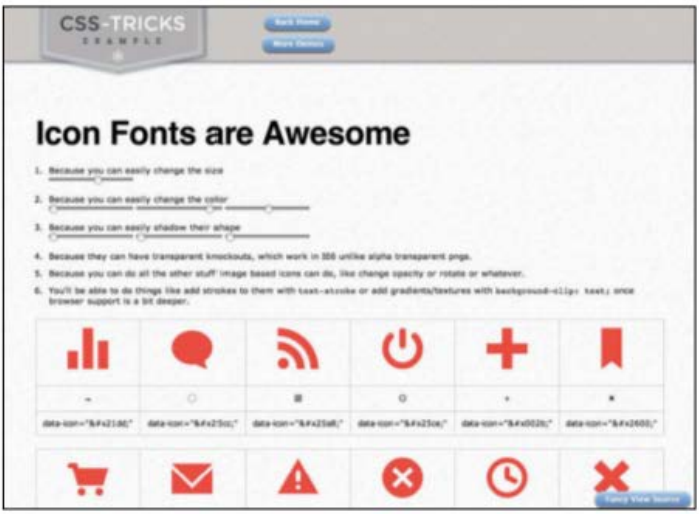


FIG 4.14 : Capture d'écran de la page de prévisualisation d'Icon Fonts de Chris Coyier (http://bkaprt.com/rrd/4-21/)

Polices d'icônes

L'une des approches remonte aux débuts de l'informatique de bureau : les polices dingbat, aussi appelées polices d'icônes (FIG 4.14). Les polices d'icônes sont de plus en plus utilisées pour présenter des images redimensionnables (particulièrement les petits éléments d'une page), et pour une bonne raison : on en trouve un très grand nombre en ligne, gratuites et payantes, et il suffit de les référencer comme polices personnalisées pour les intégrer à votre code. Du point de vue des performances, les polices d'icônes sont un excellent choix car les icônes sont toutes transmises dans un même fichier via une seule requête HTTP, voire aucune requête du tout si la police est compactée dans une URI de données.

Les polices d'icônes doivent cependant être utilisées avec précaution, car elles ont tendance à produire de piètres résultats dans les navigateurs qui ne les supportent pas. Voyons un exemple d'utilisation appropriée. Voici le code HTML :

FIG 4.15 : Glyphe en forme d'étoile tiré d'une police d'icônes avec une légende


```
<span><span class="icon-star" aria-hidden="true">
  </span>Favorite</span>
```

Voyons maintenant le code CSS. Nous allons utiliser la règle `@font-face` pour charger un fichier de police et lui attribuer la classe `font-family` « Icons ». Nous pourrons référencer cette classe `font-family` par la suite pour styler un élément en HTML :

```
@font-face {
  font-family: "Icons";
  src: url( "icons.woff" );
  font-weight: normal;
  font-style: normal;
}
.icon-star:before {
  font-family: "Icons";
  content: "★ ";
}
```

Le résultat, propre et redimensionnable, est illustré à la FIG 4.15.

Deux choses à remarquer. D'abord, nous avons utilisé un élément HTML séparé pour l'icône elle-même. C'est délibéré : nous voulons avoir la possibilité d'ajouter un attribut `aria-hidden` pour éviter que l'icône ne soit lue à voix haute par les lecteurs d'écran. (Oui, les caractères Unicode sont lus à voix haute, et l'étoile ci-dessus serait lue comme « étoile noire » par un lecteur d'écran tel que VoiceOver.)

Ensuite, nous avons utilisé le pseudo-élément `:before` pour placer le contenu de l'icône dans la page, car celui-ci nous permet de définir son contenu textuel en CSS (via la propriété `content`), ce qui est impossible avec les éléments ordinaires.

Cela nous permet de maintenir les informations de style visuel, comme le caractère ★, hors du HTML et dans le CSS, là où elles doivent se trouver.

Blinder l'approche

Comme la plupart des technologies, les polices d'icônes présentent quelques inconvénients. La règle CSS `@font-face` est supportée par de nombreux navigateurs, mais dans les environnements non compatibles, elle produit des effets inattendus. Par exemple :

- Certains navigateurs, comme le navigateur natif d'Android 2.3, affichent des carrés noirs à la place des icônes, ce qui peut provoquer des problèmes d'utilisabilité lorsqu'il n'y a pas de texte alternatif.
- Dans le navigateur à proxy populaire Opera Mini, la plupart des polices d'icônes apparaissent vides.

Pour ces raisons, nous devons inclure un test de fonctionnalité afin de blinder notre approche. Zach Leatherman a écrit un excellent article sur les différents facteurs à prendre en compte pour utiliser des polices d'icônes ; en accompagnement de son article, il a publié un script permettant d'utiliser des polices d'icônes en toute sécurité (http://bkaprt.com/rrd/4-22/). Celui-ci s'appelle A Font Garde (http://bkaprt.com/rrd/4-23/). Zach donne d'excellentes explications pour utiliser le test de fonctionnalité, et je vous recommande de lire l'article complet. Mais pour résumer, le test ajoute une classe `supports-fontface` à l'élément `html`, ce qui vous permet de qualifier vos sélecteurs comme ceci :

```
.supports-fontface .icon-star:before {
  font-family: "Icons";
  content: "★ ";
}
```

Et c'est tout !

En plus d'être redimensionnables à volonté et de donner un rendu net dans toutes les résolutions, les polices d'icônes

peuvent être stylées avec CSS de la même façon que du texte classique. Il est ainsi possible de colorer simplement une icône en lui attribuant une propriété CSS `color`, ou de lui appliquer une ombre portée avec `text-shadow`.

Du point de vue du design, le plus gros handicap des polices d'icônes est qu'elles ne permettent pas l'utilisation de plusieurs couleurs. Il n'y a rien de plus simple que de colorer une icône entière avec CSS, mais il n'y a aucun moyen de styler différemment plusieurs portions d'une icône générée ainsi. Il y a bien quelques bricolages possibles, comme empiler plusieurs caractères pour créer des icônes multicolores (http://bkaprt.com/rrd/4-24/) ou utiliser `text-shadow` pour répliquer une icône bicolore, mais on en atteint rapidement les limites.

Heureusement, si vous voulez utiliser plusieurs couleurs ou des graphiques vectoriels autres que des icônes, nous avons une autre technologie à notre disposition.

Travailler avec SVG

Le SVG, pour Scalable Vector Graphics (graphiques vectoriels adaptables), est un langage de balisage complexe et polyvalent similaire au HTML, mais conçu pour dessiner des formes. Le format SVG est pris en charge par de nombreux navigateurs depuis des années, mais en raison d'un manque de support natif sous IE8 et les versions antérieures, son usage ne s'est pas généralisé. Ces derniers temps, cependant, le SVG a connu un gros regain d'intérêt. Vu l'étendue de ses capacités, on comprend facilement pourquoi. Non seulement le SVG s'adapte à toutes les résolutions, mais ses éléments peuvent être stylés en CSS ; et comme c'est un format textuel, il se compresse très bien avec Gzip. C'est donc un format parfaitement responsable.

Voyons un exemple simple de SVG. Le petit bout de code suivant produit une étoile noire :

```
<svg>
  <polygon fill="black" points="6.504,0 8.509,4.068 »
    13,4.722 9.755,7.887 10.512,12.357 6.504,10.246 »
    2.484,12.357 3.251,7.887 0,4.722 4.492,4.068 ">
</svg>
```

FIG 4.16 : Notre exemple de code SVG produit un graphique en forme d'étoile.

Rendu dans un navigateur, ce code produit une image propre (FIG 4.16).

Comme en HTML, le document SVG commence par une balise d'ouverture (`svg` dans ce cas), qui contient un certain nombre d'éléments enfants (`line`, `circle`, `path`, `polygon`, etc.), comprenant chacun des attributs qui décrivent les propriétés visuelles de l'élément. Dans notre exemple, l'élément `polygon` est utilisé pour créer une étoile. Son attribut `fill` décrit sa couleur de remplissage (`black`), et l'attribut `points` contient une série de coordonnées délimitées par des virgules représentant les points que relient les côtés du polygone.

Au-delà de ce simple exemple, le SVG peut se faire extrêmement complexe, avec des fonctions pour les dégradés, l'ajout de liens, des modes de fusion et des filtres, et même des animations. Les éditeurs graphiques tels qu'Adobe Illustrator peuvent ouvrir, manipuler et sauvegarder des images au format SVG, permettant aux designers de travailler directement sur les fichiers qui seront transmis (FIG 4.17).

Si vous êtes un designer, vous aurez peut-être envie d'apprendre à concevoir des graphiques SVG de manière efficace à l'aide d'outils de design courants tels qu'Adobe Illustrator ou Sketch. Dans ce cas, je vous recommande fortement de lire la présentation « Leaving Pixels Behind » de Todd Parker, résumé d'un exposé qu'il a donné à la conférence Artifact en 2014 (FIG 4.18).

FIG 4.17 : Les graphiques de Paint Drop sont transmis au format SVG, ce qui leur permet d'être nets sur tous les écrans tout en étant légers à transférer.

FIG 4.18 : Première page de la présentation de Todd Parker (http://bkaprt.com/rrd/4-25/)

Il existe plusieurs moyens de servir des fichiers SVG sur le Web, en tant qu'images d'arrière-plan ou de premier plan. En voici quelques-uns.

SVG dans une balise img

Transmettre un fichier SVG via l'élément `img` est une approche pratique pour insérer des images vectorielles de premier plan, un logo par exemple. Vous pouvez référencer un fichier SVG directement dans l'attribut `src` d'un élément `img` (`<img src="star.svg" alt="...">`), mais vous ne devez pas oublier de prendre en compte les navigateurs qui ne prennent pas ce format en charge. Pour cela, incluez un élément `picture` avec

son attribut `type` et Picturefill - de sorte que les navigateurs reçoivent soit le SVG, soit un PNG en cas d'incompatibilité :

```
<picture>
  <source type="image/svg+xml" srcset="star.svg">
  <img srcset="star.png" alt="...">
</picture>
```

SVG dans votre HTML

Intégrer directement un code SVG dans un document HTML offre de nombreuses possibilités intéressantes, comme celle de réutiliser une image à travers une page, de styler différentes portions du SVG comme vous le souhaitez avec CSS, et même d'animer les traits et les formes de l'image. Pour intégrer du SVG dans un document, copiez simplement son code n'importe où sur votre page et il sera rendu par tous les navigateurs compatibles :

```
<body>
...
  <svg>
    <polygon fill="black" points="6.504,0 8.509,4.068 »
      13,4.722 9.755,7.887 10.512,12.357 6.504,10.246 »
      2.484,12.357 3.251,7.887 0,4.722 4.492,4.068">
  </svg>
...
```

Une fois le code copié, vous pouvez styler les éléments du SVG avec CSS, comme ceci :

```
svg polygon {
  fill: red;
}
```

Et ce n'est que le début ! Deux articles en particulier illustrent la puissance du code SVG intégré. Le premier est « Animated Line Drawing in SVG » de Jake Archibald, qui explique comment animer les traits d'un dessin SVG avec un peu de

FIG 4.19 : Article de Jake Archibald intitulé « Animated Line Drawing in SVG »

FIG 4.20 : Article de Chris Coyier intitulé « Icon System with SVG Sprites »

JavaScript et des transitions CSS (http://bkaprt.com/rrd/4-26/) (FIG 4.19).

Le second article est « Icon System With SVG Sprites », dans lequel Chris Coyier démontre l'utilisation des fonctions def et use de SVG, qui permettent de réutiliser des images comme

des variables à travers une page (http://bkaprt.com/rrd/4-27/) (FIG 4.20).

L'intégration de SVG directement dans le code HTML présente bien quelques inconvénients. Le premier, c'est qu'il est impossible de mettre en cache le graphique SVG comme une ressource indépendante ; le second, c'est le coût du transfert pour les navigateurs qui téléchargent le balisage SVG mais qui ne sont pas capables de le rendre. Cependant, s'il s'agit d'un petit graphique, vous pouvez éventuellement utiliser un test de fonctionnalité comme celui fourni dans Modernizr pour masquer les éléments SVG et afficher une image alternative à la place.

SVG en tant qu'objet

En servant une image SVG via l'élément `object`, on conserve les avantages d'une intégration directe dans le code HTML tout en ayant la possibilité de mettre le fichier SVG en cache pour l'utiliser ailleurs sur le site :

```
<object data="star.svg" type="image/svg+xml">
...placer le contenu alternatif ici.
</object>
```

SVG en tant qu'image d'arrière-plan

Enfin, il est également possible de référencer des fichiers SVG depuis le code CSS en tant qu'images d'arrière-plan :

```
.star {
  background: url(star.svg);
}
```

Alternativement, l'URL du fichier SVG peut être exprimée sous forme de données pures, comme nous l'avons vu précédemment dans la section sur les URI de données. Voici le code de notre image SVG intégrée en tant qu'image d'arrière-plan :

```
.star {
  background: url( "data:image/svg+xml, »
```

```
    <svg><polygon fill=\"black\" points=\"6.504,0 »
      8.509,4.068 13,4.722 9.755,7.887 10.512,12.357 »
      6.504,10.246 2.484,12.357 3.251,7.887 0,4.722 »
      4.492,4.068 \"/></svg>" );
}
```

Encore une requête HTTP de moins ! Mais faites attention à ne pas placer trop de ces URI dans un fichier CSS qui bloque le rendu : la taille du fichier résultant risque d'allonger la durée de chargement de la page. (L'astuce consiste à éviter que le CSS ne bloque le rendu, ce que nous allons voir par la suite.)

Comme nous pouvons intégrer des images d'arrière-plan SVG de cette façon, il est possible de créer toute une feuille de style contenant uniquement des images d'arrière-plan SVG, approche qui présente des avantages similaires à la technique ancestrale des sprites CSS, qui combinait plusieurs images en une seule, mais cette fois avec des graphiques vectoriels. Ce concept nous a amenés à créer Grunticon (http://bkaprt.com/rrd/4-28/), un outil permettant de générer des feuilles de sprites SVG automatiquement à partir d'un dossier de fichiers SVG source.

Automatiser les SVG avec Grunticon

Bâti sur le gestionnaire de tâches Grunt, Grunticon permet de gérer et de produire facilement des icônes et des images d'arrière-plan nettes et redimensionnables sur tous les appareils. Il prend un dossier de fichiers SVG et les convertit en fichiers CSS qui définissent des noms de classe pour chaque icône. Le CSS est exporté en trois fichiers qui contiennent les images dans trois formats différents : des URI de données SVG, des URI de données PNG et des images PNG référencées extérieurement, qui sont automatiquement créées et placées dans un dossier. Par ailleurs, Grunticon génère un morceau de JavaScript et de CSS que vous pouvez déposer sur votre site afin de charger le CSS approprié de manière asynchrone selon les capacités du navigateur, ainsi qu'un fichier HTML d'aperçu avec ce script de chargement en place (FIG 4.21).

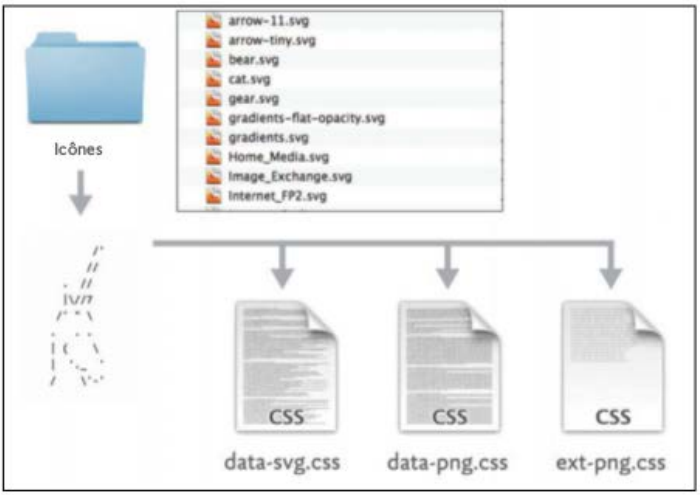


FIG 4.21 : Schéma du workflow de Grunticon

Grunticon est l'un des nombreux outils qui vous permettront d'intégrer facilement des SVG dans votre workflow de production web, et il ne cesse d'être mis à jour et amélioré.

Application Web Grumpicon

Si l'interface en ligne de commande de Grunticon offre d'excellentes possibilités d'automatisation pour le workflow d'une équipe, son installation et sa configuration peuvent s'avérer difficiles si vous n'avez pas l'habitude d'utiliser le terminal. Nous avons pour cela créé une application web appelée Grumpicon (http://bkaprt.com/rrd/4-29/) qui, en plus d'être agrémentée d'une licorne ASCII au galop, permet de convertir des fichiers SVG en ressources Grunticon tout comme l'outil en ligne de commande (**FIG 4.22**). Pour l'utiliser, rendez-vous sur le site, déposez vos fichiers SVG sur la page et téléchargez votre code prêt à l'emploi.

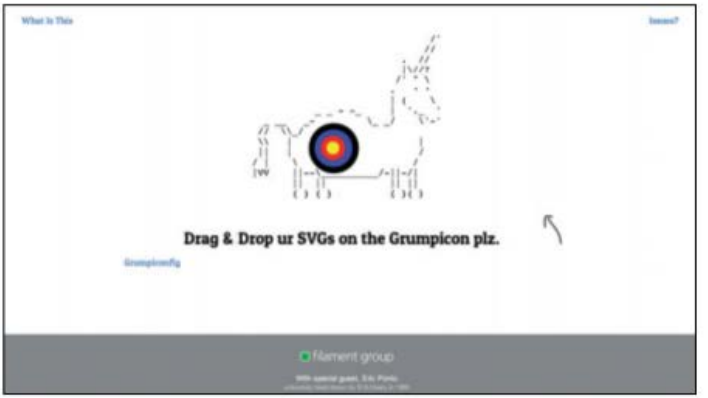


FIG 4.22 : Interface de Grumpicon

TRANSMETTRE DES POLICES DE CARACTÈRES

La prise en charge des polices web a explosé ces dernières années, et pourtant il est toujours difficile de transmettre des polices de caractères de manière responsable. Pour commencer, le comportement de chargement de polices par défaut varie selon les navigateurs. En référençant nos polices via un élément `link`, non seulement nous introduisons un point de défaillance potentiel comme avec toute autre requête CSS bloquante, mais nous devons également faire face à des problèmes tels que le potentiellement dérangeant flash de texte sans style (FOUT).

Encore un FOU

Comme nous avons déjà abordé le chargement asynchrone et le tant redouté flash de contenu sans style (FOUC), nous ne devons pas en oublier son homologue le plus récent. Un FOUT se produit lorsqu'une page HTML s'affiche avant que ses polices web personnalisées n'aient fini de se charger. Comme il se produit nativement dans certains navigateurs mais pas dans d'autres, le FOUT est un problème épineux : s'agit-il d'un bug ou d'une fonctionnalité ? (Je penche pour la deuxième option.)

FIG 4.23 : Capture d'écran du comportement de chargement des polices de caractères dans les navigateurs Firefox et Opera sur le site de STET (http://bkaprt.com/rrd/4-30/). Le FOUT est à peine remarquable ; des polices de secours sont utilisées (à gauche) avant que les polices personnalisées ne soient chargées (à droite).

Ce comportement peut être décrit ainsi : plusieurs navigateurs, notamment Firefox et Opera, n'attendent pas (ou du moins pas très longtemps) que les polices web soient chargées avant de rendre la page, utilisant alors des polices par défaut pour du texte qui serait autrement composé dans des polices personnalisées. Lorsque les polices préférées sont enfin chargées, elles sont appliquées comme indiqué par le CSS et apparaissent instantanément sur la page à la place des polices de secours. Cette substitution se produit souvent plusieurs secondes après l'affichage initial de la page. L'inconvénient, c'est que ce procédé peut être déroutant pour les utilisateurs et nécessite un nouveau rendu de la page - une nuisance pour les performances. Cela dit, la possibilité de déclarer des polices alternatives appropriées en CSS nous permet d'atténuer la douleur causée par le FOUT. Sur le site de STET, le FOUT se caractérise par un changement très subtil de mise en page une fois que les polices personnalisées ont été chargées (FIG 4.23).

Rendre un FOUT aussi subtil demande de prendre des décisions typographiques minutieuses qui ne sont certainement pas courantes sur le Web. Pour éviter tout bonnement le FOUT, certains navigateurs comme Chrome, Safari et Internet Explorer masquent le texte jusqu'à ce que les polices personnalisées aient été chargées (FIG 4.24). J'ai nommé cette approche FOIT, pour Flash of Invisible Type. Les tenants de cette approche la décrivent comme un moindre mal et avancent qu'il est moins déroutant pour les utilisateurs de ne voir aucun texte que de

FIG 4.24 : Capture d'écran du comportement de chargement des polices de caractères dans les navigateurs WebKit : le texte reste invisible jusqu'à ce que la police soit chargée.

voir du texte temporairement sans style. Le FOIT présente toutefois quelques inconvénients, et ceux-ci peuvent même être pires que le problème qu'il vise à régler. Le plus problématique, c'est le temps que le navigateur passe à attendre qu'une police se charge avant d'afficher un texte alternatif, qui peut atteindre trente secondes. Trente secondes est une éternité sur Internet, alors tant que ce comportement persistera dans les navigateurs, il vaudra mieux prendre des mesures pour l'éviter.

Pour ce que ça vaut, à l'heure où j'écris ces lignes, il semblerait que Google ait l'intention de changer ce comportement en faveur d'un *timeout* beaucoup plus court, comme l'a fait Firefox. Cependant, jusqu'à ce que tous les dérivés de WebKit soient mis à jour (allô, Android 2 ?), nous verrons des FOIT sur le Web.

Évitez le FOIT, acceptez le FOUT

Si le FOIT est le comportement par défaut de nombreux navigateurs, il se produit généralement lorsque des polices personnalisées sont chargées depuis une feuille de style référencée dans la balise `head` du code HTML d'une page. Cela s'explique par le fait que le navigateur masque uniquement le texte d'une page s'il s'attend à ce qu'une police soit chargée dans un futur proche. Dans ce contexte, nous pouvons éviter le FOIT et introduire un FOUT à la place en chargeant les fichiers CSS qui référencent nos polices de manière asynchrone via JavaScript. Charger ainsi nos polices peut présenter un compromis acceptable. La première fois que les utilisateurs visiteront une page, ils verront un bref FOUT dans tous les navigateurs, mais toutes les pages

chargées après cela ne connaîtront pas ce problème : le cache du navigateur veillera généralement à ce que toutes les polices soient immédiatement disponibles sans émettre une requête externe. Pour y parvenir, je vous recommande de convertir chacune de vos polices personnalisées en URI de données et de toutes les compiler dans un seul fichier CSS avec leurs définitions `font-face`. (Si vous servez vos polices dans différents formats, vous devrez empaqueter les polices de chaque format dans leurs propres fichiers et charger ces fichiers en fonction du support du navigateur.) L'avantage qu'il y a à transmettre des polices sous forme de données dans un seul fichier CSS, c'est que cela permet d'éliminer le temps qui s'écoule entre la définition d'une `font-face` et son chargement, minimisant ainsi le risque de FOIT. Une fois que vos polices sont combinées dans un seul fichier, vous pouvez utiliser la même fonction `loadCSS` que j'ai référencée précédemment dans la section sur le chargement de CSS :

```
<head>
  ...
  // Charger fonts.css de manière non bloquante !
  loadCSS( "fonts.css" );
  ...
</head>
```

Bien sûr, vous voudrez peut-être employer une certaine logique pour décider quel fichier inclure en fonction du format pris en charge - WOFF, TrueType, SVG, etc. Pour un programme de chargement de polices robuste, je vous recommande de vous intéresser aux nouvelles API de chargement de polices qui sortent dans les navigateurs aujourd'hui (http://bkaprt.com/rrd/4-31/).

En plus de transmettre une page utilisable aussi rapidement que possible, il existe bien d'autres facteurs à prendre en compte pour l'utilisation de polices web, mais elles dépassent le cadre de ce livre. Pour une étude plus approfondie de la typographie web, je vous renvoie humblement vers le livre de Jason Santa Maria intitulé *Typographie web*.

Si l'on revient à notre site web de 1,7 Mo, JavaScript représente la deuxième part du gâteau juste derrière les images, avec 282 Ko. Le système d'exploitation qui a amené la mission Apollo 11 jusqu'à la lune pesait 64 Ko. Qu'attendons-nous pour placer nos menus en orbite géostationnaire ?

Outre sa taille, JavaScript est un sérieux frein aux performances. Comme nous l'avons vu précédemment, JavaScript bloque le rendu de la page – par défaut, tout du moins – pendant son transfert et sa lecture, ce qui signifie que plus nous avons de scripts, plus nos utilisateurs devront attendre pour obtenir un site utilisable. Ce comportement bloquant produit des problèmes similaires à ceux que nous rencontrons lorsque nous chargeons du CSS. Cependant, JavaScript offre un certain nombre de capacités qui lui permettent d'être chargé de manière responsable plus facilement.

> *« Nous n'avons pas d'utilisateurs sans JavaScript. » Non, la vérité, c'est que tous vos utilisateurs sont sans JavaScript tant qu'ils n'ont pas téléchargé votre JS.*
> Jake Archibald, http://bkaprt.com/rrd/4-32/

Nous pouvons prendre plusieurs mesures pour obtenir d'excellentes performances réelles et perçues avec notre JavaScript, à la fois dans notre façon de le rédiger et de le transmettre. Commençons par aborder les problèmes qui concernent la taille et le transfert de notre JavaScript.

Un langage pratique mais volumineux

JavaScript a la réputation d'être une ressource volumineuse, mais ce n'est pas nécessairement dans sa nature. C'est un langage dynamique et flexible qui peut faire beaucoup avec peu de code. Le problème, c'est que JavaScript oscille depuis toujours entre plusieurs implémentations différentes et non standardisées, ce qui nous oblige à écrire le même code de multiples façons et décuple la complexité et la taille des fichiers. Les suggestions qui suivent vous aideront à combattre cette hypertrophie et à rendre votre page aussi rapide qu'elle puisse l'être.

Considérez JS comme une amélioration tertiaire

Lorsque c'est possible, il est préférable de s'appuyer sur l'HTML et le CSS natif pour faire le gros du travail et de considérer JavaScript comme un dernier recours. Car JavaScript a tendance à être notre couche d'amélioration la moins fiable : une seule erreur de syntaxe peut tout faire planter, alors qu'HTML et CSS gèrent les hics plus gracieusement. Essayez toujours de déterminer dans quelle mesure un comportement ou une présentation peut être obtenue avec HTML et CSS uniquement.

Vérifiez que vos bibliothèques sont toutes nécessaires

Les bibliothèques de gestion du DOM telles que jQuery offrent divers moyens de parcourir et de manipuler les éléments HTML via des sélecteurs CSS (entre autres). Grâce à leurs méthodes dites « écrire une fois, exécuter partout », ces bibliothèques sont devenues populaires à une époque où il était difficile de faire quoi que ce soit sur plusieurs navigateurs différents. Cependant, ces dernières années, la prise en charge de JavaScript par les navigateurs s'est considérablement améliorée et de vastes portions de ces bibliothèques ne sont plus nécessaires. La plus grosse amélioration que JavaScript a connue dans les navigateurs modernes est peut-être le support de la méthode `querySelectorAll`, qui nous permet d'accéder aux éléments du DOM à l'aide de sélecteurs CSS, comme nous le faisons avec jQuery !

```
var h3Subs = document.querySelectorAll( "h3.sub-hed" );
```

On voit également arriver dans les navigateurs de nouvelles API permettant d'ajouter et de supprimer facilement des noms de classe, de créer des boucles itératives, d'étendre des objets et bien plus encore. Grâce au support croissant de ces fonctionnalités, nous avons de moins en moins besoin de faire appel à ces grosses bibliothèques de normalisation, alors demandez-vous toujours si le coût du transfert de ces améliorations est strictement nécessaire pour les vieux navigateurs. Si vous décidez de vous passer d'une bibliothèque, faites en sorte que votre script ne se charge pas ou ne s'exécute pas dans les vieux navigateurs qui ne le comprendront pas. (Nous y viendrons bientôt.)

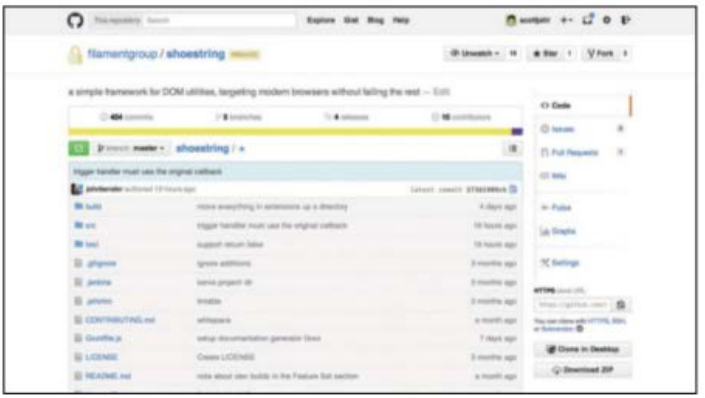


FIG 4.25 : Le projet Shoestring sur Github

Envisagez un framework DOM simple

Pour construire un site complexe, il est souvent judicieux d'utiliser une sorte de framework JavaScript ; ces frameworks offrent des fonctions courantes qui permettent d'entretenir votre code plus facilement. Cela dit, il existe de nombreux frameworks DOM de petite taille qui sont tout aussi pratiques. L'un d'entre eux, qui nous a bien servi dans nos projets (nous l'avons notamment utilisé sur des sites pour Lego), s'appelle Shoestring (http://bkaprt.com/rrd/4-33/). Shoestring a été conçu au Filament Group et il est maintenant essentiellement entretenu par mon collègue John Bender, qui est probablement un robot (il est assez *carré*). En clair, Shoestring est un framework DOM économique – il est conçu pour être rapide (FIG 4.25).

Shoestring emploie une syntaxe similaire à celle de jQuery, mais la comparaison s'arrête là. Conçu pour être extrêmement minimal, Shoestring ne contient qu'une fraction des méthodes de jQuery, et elles sont presque toutes écrites de sorte à pouvoir être exclues du fichier compilé si vous n'en avez pas besoin. Comme il ne pèse que quelques kilooctets, il offre d'excellentes performances, mais il se peut que ses capacités soient parfois trop limitées pour vos besoins. Heureusement, tout code

JavaScript que vous écrirez pour Shoestring fonctionnera également avec jQuery, aussi pourrez-vous toujours échanger les deux en un rien de temps.

Créer une compilation de jQuery personnalisée

Vous ne pouvez toujours pas vous passer de jQuery ? Vous pouvez au moins essayer de l'alléger ; jQuery propose maintenant de créer des compilations personnalisées qui excluent de nombreux modules de son code source. Selon les fonctions de jQuery dont vous avez besoin, l'essentiel peut faire à peine 12 Ko, après minification et compression. Pour apprendre à créer une compilation personnalisée, consultez ce guide (http://bkaprt.com/rrd/4-34/).

À vos marques !

Maintenant que nous avons optimisé nos dépendances JavaScript, nous pouvons apprendre à charger notre JavaScript de manière responsable. Voyons quelle est la meilleure approche pour cela.

Options pour charger du JavaScript

Souvenez-vous que tout élément `script` qui référence un fichier externe bloquera le rendu du contenu qui suit jusqu'à ce que ce fichier ait fini de se charger et de s'exécuter. Ce comportement bloquant indésirable peut être atténué ou évité selon la méthode que nous employons pour charger des scripts.

Partir du sommet

L'approche la plus simple et la plus courante pour charger du JavaScript consiste à placer un élément `script` dans la balise `head` d'un document.

```
<head>
  ...
  <script src= "myscript.js" ></script>
  ...
</head>
```

Ce comportement est clair : tout navigateur supportant JavaScript ira chercher `myscript.js` et l'exécutera dès qu'il sera transféré. Les requêtes de fichiers JavaScript référencés de cette façon sont souvent émises simultanément, mais les scripts sont exécutés dans leur ordre d'apparition dans le DOM. Cela permet de charger plusieurs scripts qui peuvent dépendre ou non les uns des autres.

```
<head>
  ...
  <script src="myjslibrary.js"></script>
  <script src="myscript.js"></script>
  ...
</head>
```

Un autre avantage de cette approche, c'est que les ressources référencées au début du code source de la page sont exposées au parser du navigateur au début du processus de chargement de la page, et sont donc téléchargées aussi tôt que possible.

Évidemment, les inconvénients ne manquent pas. Cette approche n'offre aucun moyen de qualifier les conditions sous lesquelles un script doit être téléchargé et exécuté (et dans ce cas, tous les navigateurs supportant JavaScript feront les deux), et les scripts référencés ainsi retarderont le rendu de la page jusqu'à ce qu'ils aient été chargés et exécutés. Dans quelques rares cas, il peut être désirable ou même nécessaire de bloquer le rendu jusqu'à ce qu'un script ait fini de s'exécuter. Par exemple, lorsque vous souhaitez exécuter des shims, des polyfills, des tests de fonctionnalités ou d'autres scripts qui modifient radicalement le rendu de la page, il peut être nécessaire de les référencer dans la balise `head` pour que la page se charge convenablement.

Dans ce cas, nous voulons qu'une portion de notre JavaScript apparaisse dans le `head` de la page, mais nous ne voulons néanmoins pas suspendre le chargement de la page en attendant que le JavaScript soit rapatrié. Alors examinons une autre option qui nous permet d'inclure du JavaScript dans le `head` sans requête supplémentaire.

Inclusion dans le head

Une solution au problème de latence consiste à inclure directement le code JavaScript *inline* dans le `head` de la page. Ce faisant, le JavaScript peut être exécuté dès que le code HTML est parsé, ce qui peut être mieux que d'attendre qu'un fichier externe soit rapatrié. Voici un exemple de script inline :

```
<head>
  ...
  <script>
  /* Placer le code source JavaScript ici... */
  </script>
  ...
</head>
```

Le JavaScript inline doit être utilisé avec parcimonie, car il présente aussi des inconvénients : tout script inclus directement dans la page ne peut pas être mis en cache séparément, et il sera téléchargé avec chaque nouvelle page qui l'inclut. Par exemple, les scripts inclus dans la balise `head` de la page seront téléchargés par tous les navigateurs (et exécutés par tous les navigateurs compatibles), grignotant ces 14 premiers kilooctets qui constituent notre précieux budget de rendu initial - après tout, jQuery pèse généralement plus de deux fois ce poids.

Alors à quoi peut servir cette approche ? Elle est utile pour les portions critiques et de petite taille de votre code source JavaScript mentionnées ci-dessus (les shims, polyfills, etc.), mais elle doit être réservée au JavaScript qui doit absolument se trouver dans le `head`.

Et si aucune portion de votre JavaScript ne répond à ces critères ?

Charger depuis la base

Une troisième approche pour charger du JavaScript consiste à placer les éléments `script` à la fin d'un document HTML, permettant au contenu d'être chargé et rendu aussi vite que possible et forçant les scripts à se charger et à s'exécuter une fois que le contenu lui-même a été lu et rendu. L'un des avantages

évidents de cette méthode est que les utilisateurs peuvent interagir plus rapidement avec la page.

Hélas, elle souffre également de sérieux handicaps. Tout d'abord, cette approche présente le même problème que les scripts référencés dans la balise `head` en ce sens qu'il n'y a aucun moyen de qualifier la requête ou l'exécution d'un script – il sera rapatrié par tous les navigateurs supportant JavaScript. De plus, les scripts référencés à la fin d'un document sont récupérés beaucoup plus tard, et prennent plus de temps à se charger et à s'exécuter qu'un script référencé plus haut dans la page. Cela peut être plus ou moins acceptable selon que votre JavaScript affecte ou non la présentation de la page, mais toute amélioration visuelle non chargée risque de causer un flash de contenu sans style, alors soyez bien attentif.

Retour au sommet avec les attributs defer et async

Dans les navigateurs modernes tels que IE10 (et pratiquement tous les autres navigateurs depuis des années), les attributs `async` et `defer` peuvent être ajoutés aux éléments `script` pour demander au navigateur de charger un fichier JavaScript référencé parallèlement au code HTML (`async`) et/ou d'exécuter le script une fois le chargement du HTML terminé (`defer`). Ces attributs peuvent être utilisés indépendamment ou ensemble sur un même élément `script`.

```
<script src="myScript.js" async defer></script>
```

Si un script n'a pas besoin d'être exécuté immédiatement, l'attribut `defer` peut être extrêmement bénéfique pour les performances de chargement de la page, car il libère le navigateur pour qu'il puisse se concentrer sur d'autres tâches essentielles et plus prioritaires. Vous pourrez par exemple choisir de différer les scripts qui appliquent des comportements à des composants qui seront eux-mêmes chargés de manière différée, ou qui contrôlent du contenu placé vers le bas de page, comme les commentaires d'un blog.

Cela étant, il est souvent préférable que les fichiers JavaScript s'exécutent aussi tôt que possible, si bien que `defer` n'est pas forcément idéal. Pour les scripts qui peuvent sans problème

s'exécuter dès qu'ils sont prêts, même si le document HTML n'est pas entièrement chargé, l'attribut `async` est préférable.

Ce qui est génial avec ce dernier, c'est qu'il peut s'appliquer aux éléments `script` placés dans la balise `head` (en supposant qu'ils référencent des fichiers externes) pour demander au navigateur de rapatrier le fichier référencé immédiatement, mais de commencer le rendu de la page pendant que ce fichier se charge en parallèle - le meilleur des deux mondes.

```
<head>
  ...
  <script src="myScript.js" async></script>
  ...
</head>
```

Et maintenant, les inévitables inconvénients. Tout d'abord, bien que ces attributs soient largement supportés, ils n'empêcheront pas les scripts de bloquer le rendu de la page dans les navigateurs non compatibles, tels qu'Android 2. IE9 et les versions antérieures ne prennent pas `async` en charge mais supportent `defer` (dans IE5 et versions ultérieures) ; vous pouvez donc combiner les deux avec `defer` comme option de secours.

Ensuite, s'il y a plusieurs scripts, `async` ne garantira pas qu'ils s'exécutent dans l'ordre dans lequel ils sont spécifiés dans le code source. Si `defer` est censé garantir l'ordre d'exécution, il n'y parvient toujours pas dans IE9 et les versions antérieures. Si les scripts que vous chargez ne dépendent pas les uns des autres, ça ne devrait pas poser de problème.

Enfin, comme toutes les approches précédentes, les attributs `async` et `defer` n'offrent aucun moyen de qualifier le transfert ou l'exécution d'un script. Pour cela, nous devons souvent nous appuyer sur des outils non natifs afin de développer un site multi-appareil responsable.

Le juste milieu : charger des scripts dynamiquement à l'aide d'un petit script inline

Notre dernière option pour charger du JavaScript est celle que je recommanderai le plus. Le chargement dynamique nous permet de décider de charger ou non des fichiers supplémentaires en

fonction d'un certain nombre de conditions et, le cas échéant, de transmettre ces fichiers d'une manière non bloquante. Dans un codebase conçu pour répondre à une grande diversité de conditions de réseau, de fonctionnalités d'appareil et de préférences personnelles, le chargement dynamique est l'approche la plus responsable que nous puissions employer, car elle nous permet de charger seulement le strict nécessaire et rien de plus.

Il est simple de charger du JavaScript dynamiquement : placez un bout de JavaScript inline dans la page et utilisez ce script pour greffer des éléments `script` supplémentaires, qui seront téléchargés et exécutés en parallèle. Il existe plusieurs moyens de greffer dynamiquement des éléments sur une page avec JavaScript, mais la méthode `insertBefore` est la plus sûre et la plus fiable. Voici un exemple d'utilisation d'`insertBefore` pour charger un script (« myScript.js ») dans la balise `head` d'un document HTML :

```
<head>
  ...
  <script>
    var myJS = document.createElement( "script" );
    myScript.src = "myScript.js";
    var ref = document.getElementsByTagName( »
      "script" )[ 0 ];
    ref.parentNode.insertBefore( myJS, ref );
  </script>
</head>
```

Décomposons ce qui se passe dans ce bout de code :

- Dans les deux premières lignes, nous créons un élément `script` référencé par la variable `myJS` et nous définissons son attribut `src` comme « myScript.js ».
- À la ligne suivante, nous créons une variable `ref` pour stocker une référence au premier élément `script` trouvé dans la page (qui pourrait très bien être celui qui contient l'exemple de script ci-dessus).

- Enfin, nous appelons la méthode `insertBefore` sur le parent de `ref` (dans ce cas, l'élément `head`), en spécifiant que `myJS` - qui se réfère à l'élément `script` que nous insérons - doit être inséré juste avant `ref`.

Si vous inspectez le DOM après l'exécution de ce script, vous verrez ce résultat (script nouvellement greffé et chargé en gras) :

```
<head>
  ...
  <script src="myScript.js"></script>
  <script>
    var myJS = document.createElement( "script" );
    myScript.src = "myScript.js";
    var ref = document.getElementsByTagName( »
      "script" )[ 0 ];
    ref.parentNode.insertBefore( myJS, ref );
  </script>
</head>
```

Ce modèle forme la base de bon nombre des scripts de chargement de ressources plus complets qu'on utilise aujourd'hui ; de fait, je m'en sers sur pratiquement tous les sites que je construis. Il faut toutefois garder une limitation à l'esprit : si vous avez besoin de charger plusieurs fichiers de script qui dépendent les uns des autres, cette approche peut causer des problèmes ; elle ne fait rien pour préserver l'ordre d'exécution des scripts. Cela dit, si vous combinez tous vos scripts d'amélioration en un seul fichier (ce que je conseillerai uniquement si vous pouvez en faire un fichier de taille raisonnable), ce simple script pourra répondre à vos besoins sans problème. J'ai compilé le script de cet exemple dans une fonction réutilisable appelée `loadJS` (http://bkaprt.com/rrd/4-35/). La voilà en action ; en gras, le texte demandant à `loadJS` de charger le même script que dans les exemples précédents :

```
<script>
  /* Inclure la fonction loadJS */
```

```
    function loadJS( src ){ ... }
    loadJS( "myScript.js" );
</script>
```

Avec cet outil pratique à notre disposition, nous pouvons réduire drastiquement le code inclus dans la balise `head` de notre page et améliorer l'expérience sans bloquer le chargement de la page.

Amélioration responsable

Imaginez que vous deviez développer un site qui nécessite d'ajouter des fonctionnalités spéciales dans l'interface. Ces améliorations demandent plus de JavaScript et de CSS que vous ne pourriez en caser dans la première vague de code envoyée à tous les navigateurs. Mais vous ne voulez pas surcharger tous les navigateurs avec ce code et ces requêtes supplémentaires - seulement ceux qui peuvent les utiliser.

Navigateurs à la hauteur

De même que nous pouvons qualifier les règles CSS avec `@media only all`, nous pouvons également qualifier l'application de nos améliorations en JavaScript. Parfois, ces qualifications coïncident avec des fonctionnalités qui sont nécessaires pour l'expérience améliorée d'un site web, mais elles peuvent également servir de diagnostic plus général pour la prise en charge de fonctionnalités modernes.

Dans son article « Cutting the Mustard », Tom Maslen, développeur de la BBC, décrit son approche des améliorations comme étant « une solution adaptative à deux niveaux » (http://bkaprt.com/rrd/4-36/). En fonction de leurs capacités, les navigateurs reçoivent soit une expérience simple et fonctionnelle reposant uniquement sur du HTML, soit la version améliorée. Pour tester si un navigateur est à la hauteur, la BBC a créé un outil de diagnostic pour voir si le navigateur prend en charge certaines fonctionnalités. Si le navigateur réussit le test, il obtient l'expérience améliorée. Dans son article, Maslen mentionne le diagnostic suivant à titre d'exemple :

```
if( "querySelector" in document
  && "localStorage" in window
  && "addEventListener" in window ){
  // Ce navigateur est à la hauteur !
}
```

Dans ce cas, il vérifie la présence de trois méthodes JavaScript qui doivent être définies avant de poursuivre : `querySelector`, `localStorage` et `addEventListener`, qui sont supportées dans un navigateur comme IE9, mais pas dans d'autres comme IE8. La barre à franchir peut être différente selon les besoins du site. Par exemple, le site du *Boston Globe* estime qu'un navigateur est à la hauteur s'il supporte les media queries :

```
if( window.matchMedia && window.matchMedia( »
  "only all" ) ){
  // Ce navigateur est à la hauteur !
}
```

Tester les qualificateurs nous permet de nous assurer que nous n'appliquons des scripts et des styles améliorés qu'aux navigateurs qui peuvent les comprendre. (Par ailleurs, nous pouvons toujours introduire des tests de fonctionnalités plus spécifiques pour qualifier l'utilisation de fonctionnalités qui demandent des solutions alternatives plus soignées.) En qualifiant nos améliorations, nous facilitons également les tests d'assurance qualité : lorsque vous savez qu'un navigateur particulier n'est « pas à la hauteur », il est rassurant de savoir que ses utilisateurs ne rencontreront pas de problème d'utilisabilité, puisque vous leur offrez une expérience tout aussi fonctionnelle.

Une fois que j'ai déterminé si le navigateur répondait à mes critères, j'ai l'habitude de démarrer le processus d'amélioration en appliquant une classe `enhanced` à l'élément `html`.

```
document.documentElement.className += " enhanced";
```

J'utilise parfois `.enhanced` dans les sélecteurs CSS qui appliquent des styles qui ne doivent se produire que dans les environnements qualifiés, comme ce petit bout de code qui masque

les cases à cocher dans les environnements améliorés (en supposant qu'elles seront remplacées par une icône personnalisée) :

```
.enhanced label input[type=checkbox] {
  opacity: 0;
}
```

S'il est important de qualifier l'application du code, il est également important de déterminer s'il est nécessaire de télécharger le code, car il est toujours préférable d'éviter les requêtes HTTP superflues. Voyons maintenant comment charger certaines ressources de manière qualifiée.

Chargement de ressources qualifié

Si notre but est de charger un seul script de façon dynamique, le modèle `loadJS()` présenté précédemment est tout ce dont nous avons besoin, et nous pouvons le qualifier avec n'importe quelle condition de sorte que le fichier soit uniquement téléchargé par les navigateurs capables. Par exemple, voici un bout de code qui chargera notre script dans n'importe quel navigateur supportant `querySelector` (tel qu'IE8 et les versions ultérieures) :

```
// Vérifions si le navigateur supporte querySelector
if( "querySelector" in document ){
  // Ce navigateur est à la hauteur !

  // Commençons par ajouter une classe à l'élément HTML
  document.documentElement.className += " enhanced";

  // Ensuite, chargeons nos scripts d'amélioration
  loadJS( "myScript.js" );
}
```

On chauffe ! Rapprochons cela de notre approche de chargement de CSS pour nous faire une idée plus globale.

FIG 4.26 : Le projet EnhanceJS sur Github (http://bkaprt.com/rrd/4-37)

FAIRE LA SYNTHÈSE

Nous avons abordé le chargement séparé de nos nombreuses ressources, mais pour toutes les charger de manière efficace dans le même code source, il faut du soin et de l'organisation. Pour conclure cette section, voyons comment notre HTML, notre CSS et notre JavaScript peuvent être assemblés pour transmettre notre site rapidement et de manière responsable.

La balise `head` de la page est l'endroit où nous contrôlons le processus d'amélioration de la page, alors focalisons-nous là-dessus. Dans le `head`, nous utiliserons les techniques abordées dans cette section, comme l'inclusion inline de notre CSS critique et du JavaScript qui nous permettra de charger des scripts, des styles et des polices supplémentaires de manière qualifiée.

Pour vous aider dans ce processus, j'ai mis à jour le projet EnhanceJS (évoqué au début de ce livre) pour héberger un exemple du workflow JavaScript que nous utilisons afin d'améliorer une page (FIG 4.26). Un fichier du projet, `enhance.js`, contient le code des fonctions `loadCSS()` et `loadJS()` mentionnées précédemment, ainsi que quelques fonctions d'aide pour récupérer et paramétrer des cookies, accéder aux valeurs des

éléments `meta`, et un exemple de test de fonctionnalités pour déterminer si le navigateur est à la hauteur. Contrairement à un framework JavaScript, `enhance.js` est conçu pour être un modèle standard modifiable : quand vous l'employez, supprimez tout ce dont vous n'avez pas besoin et ajoutez ce qui servira à votre projet. Les exemples suivants utilisent `enhance.js` pour les portions de JavaScript inline de notre workflow.

C'est trop méta !

Tout d'abord, nous savons que nous avons plusieurs fichiers à charger avec JavaScript ; je commence généralement par définir les URL de ces fichiers à un endroit facile à retrouver. Les éléments `meta` sont parfaits pour ça, alors j'en placerai quelques-uns au sommet de la balise `head` : un pour le fichier CSS complet de notre site, un pour nos polices personnalisées et un pour nos améliorations en JavaScript.

```
<head>
  ...
  <meta name="fullcss" content="/path/to/full.css">
  <meta name="fonts" content="/path/to/fonts.css">
  <meta name="fulljs" content="/path/to/ "
    enhancements.js">
  ...
</head>
```

Ajouter le JavaScript critique

Une fois les balises `meta` en place, nous pouvons ajouter notre script inline. Là encore, ce script inline doit uniquement comprendre le JavaScript requis pour améliorer potentiellement l'expérience. J'inclus généralement une version modifiée d'`enhance.js` contenant les fonctions et les logiques spécifiques au site dont je pourrais avoir besoin. Dans ce cas, nous voulons charger dynamiquement le fichier CSS complet de notre site sans aucune qualification (puisqu'il contient des styles qui s'appliquent à tous les environnements) et vérifier si le navigateur passe le test de fonctionnalités. Si tel est le cas, nous allons également charger nos polices personnalisées et notre JavaScript

amélioré. (Remarque : vous pouvez choisir de charger les polices personnalisées pour tout le monde, mais nous les réservons généralement aux navigateurs modernes, où elles seront le plus appréciées.)

Voici notre section `head` mise à jour avec le script inline en place (ajouts en gras).

```
<head>
  ...
  <meta name="fullcss" content="/path/to/full.css">
  <meta name="fonts" content="/path/to/fonts.css">
  <meta name="fulljs" content="/path/to/ "
    enhancements.js">
  <script>
    {% include path/to/enhance.js %}
  </script>
  ...
</head>
```

Dans ce fichier `enhance.js`, aux côtés de fonctions telles que `loadCSS()` et `loadJS()`, se trouve une autre fonction appelée `getMeta()`, qui essaie de localiser un élément `meta` à l'aide de son attribut `name`. En première étape, nous pouvons trouver l'élément `meta` référençant le fichier CSS complet de notre site avec l'appel suivant :

```
getMeta( "fullcss" );
```

Une fois que nous avons une référence à cet élément `meta`, nous pouvons obtenir sa propriété `content` pour trouver l'URL dont nous avons besoin, et charger le fichier CSS avec `loadCSS()` :

```
var cssMeta = getMeta( "fullcss" );
if( cssMeta ){
  // Chargeons le fichier CSS complet du site à partir de
  // l'attribut content de l'élément meta
  loadCSS( cssMeta.content );
}
```

Et cela suffit à charger le fichier CSS complet de notre site de manière asynchrone.

Dans la suite de ce script inline, nous voulons vérifier que le navigateur passe notre test avant d'appliquer des améliorations supplémentaires en ajoutant une classe et en chargeant notre JavaScript amélioré et nos polices personnalisées. Voici comment :

```
...
// Vérifier que le navigateur supporte querySelector
if( "querySelector" in document ){
  // Ce navigateur passe le test !

  // Commençons par ajouter une classe à l'élément HTML
  document.documentElement.className += " enhanced";

  // Ensuite, chargeons notre script d'amélioration
  var jsMeta = getMeta( "fulljs" );
  if( jsMeta ){
  loadJS( jsMeta.content );
  }

// Enfin, chargeons nos polices personnalisées
var fontsMeta = getMeta( "fonts" );
if( fontsMeta ){
  loadCSS( fontsMeta.content );
  }
}
```

Et voilà pour le JavaScript.

Ensuite, dans la balise `head` du document, ajoutons le CSS critique nécessaire pour rendre la portion supérieure de la page. J'en profite pour vous rappeler que ce CSS critique variera d'un template à l'autre, et doit être généré par un outil comme Grunt-CriticalCSS, que j'ai mentionné dans la section sur le CSS. Nous plaçons cette portion du CSS directement dans la balise `head`, comme ceci (ajouts en gras) :

```
<head>
  ...
  <meta name="fullcss" content="/path/to/full.css">
  <meta name="fonts" content="/path/to/fonts.css">
  <meta name="fulljs" content="/path/to/ "
    enhancements.js">
  <style>
    /* Les styles CSS critiques pour ce template sont
    placés ici... */
  </style>
  <script>
    {% include path/to/enhance.js %}
  </script>
  ...
</head>
```

Là encore, comme les éléments `script` et `style` ne référencent aucun code externe, leur ordre dans le code source de la page n'affecteront pas les performances de rendu. Cela étant, il vaut mieux placer le CSS critique avant le `script`, de sorte que JavaScript insère le fichier CSS complet après les styles inline et qu'aucun de ceux-ci ne soient écrasés par les styles contenus dans le fichier CSS complet.

Ensuite, j'inclus généralement une référence statique au fichier CSS complet du site pour les navigateurs qui n'ont pas JavaScript activé. On veille ainsi à ce que l'intégralité du CSS se charge, qu'on puisse ou non le charger dynamiquement avec JavaScript. Voici le dernier ajout en gras :

```
<head>
  ...
  <meta name="fullcss" content="/path/to/full.css">
  <meta name="fonts" content="/path/to/fonts.css">
  <meta name="fulljs" content="/path/to/ »
    enhancements.js">
  <script>
    {% include path/to/enhance.js %}
  </script>
  <style>
```

```
    /* Les styles CSS critiques pour ce template sont
    placés ici... */
  </style>
  <noscript>
    <link rel="stylesheet" href="/path/to/full.css">
  </noscript>
  ...
</head>
```

Pfiou ! Et voilà, c'est tout. Ensuite, dans le code source de la page, l'élément `body` doit contenir tout le contenu HTML fonctionnel nécessaire pour que cette page soit utile à tous les utilisateurs.

Optimiser les chargements suivants avec des cookies

Mais vous pouvez encore faire un petit quelque chose. Le workflow ci-dessus est fantastiquement optimisé pour un utilisateur qui visite le site pour la première fois, mais nous pouvons tirer parti du cache du navigateur pour que la page se charge encore plus vite lors des visites suivantes.

Le secteur principal d'optimisation est le CSS inline, qui doit être présent lors de la première visite seulement, le temps que le CSS complet du site soit téléchargé et mis en cache par le navigateur. Après la visite initiale, le navigateur aura déjà téléchargé et mis en cache le CSS complet ; nous pouvons donc référencer directement ce CSS dans la balise `head` de la page à la place du CSS inline que le template inclurait autrement. Pour y parvenir, il est nécessaire d'ajouter une petite logique côté serveur dans votre template ; votre page devra donc s'exécuter sur un serveur web qui prend au moins en charge les scripts simples, comme la possibilité de détecter des cookies. Si vous avez accès à un environnement de ce type, la configuration de cette optimisation est relativement simple.

Tout d'abord, vous devez configurer la balise `head` de votre document de sorte qu'elle inclue la portion de code CSS inline ou non selon l'existence d'un cookie, que nous appellerons `fullcss` pour les besoins de cet exemple. Voici à quoi votre `head` devra ressembler :

```
<head>
  ...
  <meta name="fullcss" content="/path/to/full.css">
  <meta name="fulljs" content="/path/to/ »
    enhancements.js">
  <script>
    {% include /path/to/enhance.js %}
  </script>
{% if cookie "fullcss=true" %}
  <link rel="stylesheet" href="/path/to/full.css">
{% else %}
  <style>
    /* Les styles CSS critiques pour ce template sont
    placés ici... */
  </style>
  <noscript>
    <link rel="stylesheet" href="/path/to/full.css">
  </noscript>
{% endif %}
  ...
</head>
```

Ensuite, vous devez définir, dans le code JavaScript inline, un cookie qui déclare que le fichier CSS a été téléchargé et mis en cache une fois que cela est fait. Ce cookie sera stocké dans le navigateur et accompagnera chaque requête ultérieurement envoyée au serveur, permettant ainsi à la logique ci-dessus de le détecter. Par ailleurs, pour les pages chargées par la suite, le JavaScript doit vérifier que le cookie n'a pas déjà été défini avant de charger le CSS complet du site (comme ce CSS est susceptible d'être déjà inclus dans la page).

Voici à quoi cela ressemble en utilisant la fonction `cookie` d'`enhance.js` (nouveau code en gras) :

```
var cssMeta = getMeta( "fullcss" );
if( cssMeta ){
  // Charger le fichier CSS complet du site à partir de
  // l'attribut content de l'élément meta
  loadCSS( cssMeta.content );
```

```
    // définir un cookie " fullcss " dont la valeur est true
    cookie( "fullcss", "true" );
}
```

Et c'est *vraiment* tout !

Si vous avez envie de voir cet exemple en action, rendez-vous sur le site d'EnhanceJS (http://bkaprt.com/rrd/4-37/), qui offre des fichiers de démonstration fonctionnels correspondant au workflow décrit ici.

Va et sois responsable

En utilisant le workflow de chargement simple décrit ci-dessus, nous pouvons transmettre nos ressources uniquement lorsque c'est nécessaire, mais surtout servir des pages qui sont rendues extrêmement vite pour nos utilisateurs. On n'insistera pas assez sur l'importance de qualifier les requêtes pour les améliorations de grande taille, car chaque requête est susceptible de rajouter plusieurs secondes au temps de chargement d'une page (particulièrement sur une connexion mobile).

VOUS ÊTES SERVI

Nous avons couvert beaucoup de sujets dans ce chapitre, de l'optimisation de nos ressources aux différents moyens de les charger (ou pas !) de manière responsable dans nos pages pour un rendu plus rapide. Les sites sont plus que des collections de pages ; ce sont des systèmes complexes qui peuvent facilement s'alourdir si nous ne sommes pas vigilants. Même si nous traversons encore une période de transition, nous avançons vers des expériences plus radieuses et plus légères que jamais. Voyons ce que l'avenir nous réserve.

CONCLUSION

Dans ce livre, j'ai fait part de méthodes pour construire des sites responsive qui mettent l'accent sur l'utilisabilité, l'accessibilité, la durabilité et les performances. Tous ces facteurs sont importants pour le Web d'aujourd'hui, mais préparent également nos sites pour les navigateurs de demain.

Notre public se diversifie de plus en plus, géographiquement et technologiquement parlant. À mesure que l'accès au Web s'améliore dans les régions en développement, nous avons l'occasion d'étendre notre portée, et nous avons plus que jamais besoin d'optimiser la transmission de notre contenu. Dans le même temps, ces pratiques de design responsables sont également bénéfiques pour les utilisateurs qui habitent dans les régions développées, en leur offrant des sites plus rapides et plus accessibles, même lorsque les conditions de navigation sont loin d'être idéales. Les contextes dans lesquels nos utilisateurs utilisent le Web sont très divers, mais le Web souhaité par tous prend en considération les conditions de navigation, les contraintes et les attentes de chacun.

Un futur responsive et responsable

Nous ne pouvons pas prédire ce que le futur du Web nous réserve, mais nous pouvons nous préparer à répondre aux inconnues qui nous attendent. Pour tenir la promesse d'un Web largement accessible, plaisant à utiliser et durable, le responsive design doit être intégré à d'autres bonnes pratiques. Pour aller de l'avant, nous devons accepter la diversité des appareils et des réseaux et nous focaliser sur les fonctionnalités et les contraintes plutôt que sur les navigateurs et les appareils.

Le Web a été fondé sur les prémisses d'un accès large et inclusif, et sa façon progressive et unique de mettre les technologies en application constitue l'armature qui nous permettra de concevoir des expériences toujours plus exceptionnelles sans laisser personne de côté. Pour atteindre cet objectif, nous devons réfléchir de façon créative, concevoir de manière responsable et accorder la priorité absolue à nos utilisateurs.

Puisse votre rayonnement s'étendre, vos chutes être gracieuses et vos objectifs bienveillants. Merci infiniment de m'avoir lu.

REMERCIEMENTS

Je regrette qu'il me soit impossible de remercier dans un si petit espace toutes les personnes dont le talent et la générosité ont permis à ce livre de paraître. Je peux tout au mieux citer les plus importantes.

Ma famille m'a toujours apporté un soutien sans faille : ma mère et mon père, Kristen et Adam, mes grands-parents. Merci à mes parents de m'avoir envoyé apprendre le design au Flagler College, où j'ai rencontré mon mentor Randy Taylor, un designer talentueux qui est resté un bon ami à ce jour. Randy, avec Seth Ferreira, m'a encouragé à intégrer le design web dans mon cursus et m'a présenté mes premiers clients, notamment Dorothy Hesson de la Florida School for the Deaf & the Blind. La passion et le savoir de Dorothy ont éveillé mon intérêt pour l'accessibilité.

Merci à Jon Reil, qui a pris le risque de m'engager alors que je sortais tout juste de l'école. À Jeffrey Zeldman, Eric Meyer et Jeremy Keith, que je considérais autrefois comme des héros inaccessibles et que j'ai le bonheur de compter aujourd'hui parmi mes amis. Merci à Paul Irish, Steve Souders, Ilya Grigorik et Andy Hume qui ont façonné ma compréhension des performances web. Merci à John Resig de m'avoir accueilli au sein de l'équipe de jQuery au tout début. Merci à la brillante communauté du web design qui m'en apprend un peu chaque jour : Jake Archibald, Tim Kadlec, Trent Walton, Dave Rupert, Chris Coyier, Mat Marquis, Bryan et Stephanie Rieger, Stephen Hay, Nicole Sullivan, Dan Cederholm, Brad Frost, Jason Grigsby, Josh Clark, Luke Wroblewski, Anna Debenham, James Craig, Karl Swedberg et Lyza Gardner... la liste est longue.

Un grand « high five » à mon équipe du Filament Group : Patty Toland, Todd Parker, Maggie Wachs, Zach Leatherman, Jeff Lembeck et John Bender. Chaque jour, j'ai la chance de travailler avec les gens les plus intelligents et les plus attentionnés du métier. Quasiment chaque ligne de ce livre est basée sur des recherches que nous avons effectuées ou étudiées au sein de l'entreprise. Patty et Todd dirigent une entreprise unique qui parvient à contribuer à énormément de projets open source, à produire du travail intéressant et important, et à équilibrer vie

professionnelle et vie personnelle avec générosité. Je leur suis infiniment reconnaissant de tout ce qu'ils font pour nous.

J'ai une dette énorme envers A Book Apart et cette équipe éditoriale. Tina Lee a apporté plusieurs séries de restructurations, de questions et de coupes sensées, réorganisant ma pensée chaotique de façon beaucoup plus claire. Mandy Brown, mon premier contact chez A Book Apart, m'a apporté ses précieux conseils d'éditrice. Katel LeDu, la directrice générale, a fait preuve d'un professionnalisme exceptionnel pour s'assurer que notre équipe gardait le cap et collaborait de manière efficace. Le talent de Rob Weychert et de Jason Santa Maria a permis de transformer une simple colonne de texte en ce superbe ouvrage que vous tenez entre les mains.

Un merci tout particulier à Ethan Marcotte. Non content d'avoir conçu la pratique sur laquelle ce livre est basé, Ethan est depuis longtemps un modèle pour moi, non seulement pour sa contribution essentielle à notre discipline, mais également pour la courtoisie dont il sait faire preuve. Ethan a été mon relecteur technique et a apporté de nombreuses améliorations au code contenu dans ce livre, mais il m'a également fait part de ses précieux conseils et n'a pas hésité à me dire que certains passages devaient être revus. Je suis profondément honoré qu'Ethan ait écrit l'avant-propos de ce livre, et je suis fier de le compter parmi mes amis.

Je conclurai par le plus important : merci à Stephanie, ma femme brillante et talentueuse. Ce livre a été écrit pendant des nuits et des week-ends, à des moments où Steph et moi aurions autrement pu passer du temps ensemble. Elle a non seulement sacrifié son sommeil et ses loisirs (pendant une grossesse, en plus !), mais elle m'a également fait part de ses conseils avisés pour de nombreuses sections de ce livre. J'avais écrit la moitié de ce livre quand notre fille Emory est née, et elle a eu un an quand il est paru (en anglais - *NdT*). Les livres sont apparemment le « jouet » préféré d'Emory, alors j'aime à penser qu'un jour, elle appréciera le fait que son père ait, lui aussi, fabriqué l'un de ces jouets. Steph et Emory méritent toute ma gratitude. Je vous aime toutes les deux.

Appareils, navigateurs et tests

- **BrowserStack** : si vous concevez des sites multi-appareils, vous devrez avoir accès à de multiples appareils. BrowserStack vous permet de tester votre site en direct sur un nombre croissant de systèmes d'exploitations et de navigateurs. Hautement recommandé (http://bkaprt.com/rrd/5-01/).
- **Can I use...** : une excellente ressource pour savoir quelles fonctionnalités sont prises en charge par différents navigateurs à l'heure actuelle (http://bkaprt.com/rrd/5-02/).
- **Akamai State of the Internet** : ce rapport détaille la vitesse des connexions et la couverture de l'accès à Internet dans le monde (http://bkaprt.com/rrd/5-03/).
- **StatCounter** : bien qu'il ne soit certainement pas le plus complet des outils de trafic web, StatCounter est une source fiable pour obtenir des statistiques sur les navigateurs et les systèmes d'exploitation utilisés dans le monde entier (http://bkaprt.com/rrd/5-04/).
- **« Grade Components, Not Browsers »** : ce billet plaide pour une évolution de notre façon de documenter les différences d'expérience utilisateur entre différents navigateurs (http://bkaprt.com/rrd/5-05/).

Optimisation et analyse des performances

- **WebPagetest** : un excellent service qui analyse comment votre site se charge à différents endroits dans le monde et sur différents appareils et navigateurs (http://bkaprt.com/rrd/5-06/).
- **PageSpeed Insights** : une application web rapide qui relève les améliorations à apporter pour optimiser les performances du site (http://bkaprt.com/rrd/5-07/).
- **Grunt-PerfBudget** : un outil en ligne de commande conçu par Tim Kadlec pour automatiser les tests de performance avec WebPagetest (http://bkaprt.com/rrd/5-08/).
- **« Setting a Performance Budget »** : ce billet de Tim Kadlec expose les considérations principales à prendre en compte

pour établir le budget de performance d'un site (http://bkaprt.com/rrd/5-09/).

- **« Test on Real Mobile Devices without Breaking the Bank » :** Brad Frost offre d'excellents conseils pour construire un laboratoire de test équilibré (http://bkaprt.com/rrd/5-10/).
- **Steve Souders :** Steve Souders est le gourou des performances web. Lisez-le régulièrement (http://bkaprt.com/rrd/5-11/) !
- **Building a Performance Culture :** cette excellente présentation de Lara Swanson et de Paul Lewis aborde les difficultés et les avantages qu'il y a à donner la priorité aux performances au sein d'une entreprise (http://bkaprt.com/rrd/5-12/).
- **Open Device Lab :** trouvez un laboratoire de test d'appareils près de chez vous (http://bkaprt.com/rrd/5-13/).

Pratiques de programmation et outils pensés pour l'avenir

- **Détection de fonctionnalités côté serveur :** la proposition HTTP Client-Hints, imaginée par Ilya Grigorik de Google, permettra aux navigateurs d'envoyer des informations standardisées sur leurs fonctionnalités et leurs conditions au serveur avec chaque requête. Gardez un œil sur l'avancement de la proposition (http://bkaprt.com/rrd/5-14/).
- **Chargement de CSS :** un certain nombre de solutions potentielles pour le chargement de CSS inapplicable ou de faible priorité sont en cours de discussion sur la mailing-list du W3C (http://bkaprt.com/rrd/5-15/). Des idées telles qu'un attribut `onmatch` pour les éléments `link` ont déjà été proposées (http://bkaprt.com/rrd/5-16/).
- **Test de fonctionnalités :** de nombreuses fonctionnalités ne devraient pas être utilisées sans vérifier qu'elles sont correctement supportées. Modernizr est la meilleure bibliothèque de détection de fonctionnalités disponible à l'heure actuelle (http://bkaprt.com/rrd/5-17/).
- **Images responsive :** l'article d'Opera intitulé « Responsive Images: Use Cases and Documented Code Snippets to Get You Started » est à lire absolument si vous avez l'intention d'utiliser des images responsive (http://bkaprt.com/rrd/5-18/). Consultez régulièrement le RICG (http://bkaprt.com/rrd/5-19/) et Picturefill (http://bkaprt.com/rrd/5-20/) par ailleurs.

- **Projets open source de Filament** : Filament héberge un nombre croissant de composants et d'outils responsive pleinement testés que vous pouvez utiliser gratuitement (http://bkaprt.com/rrd/5-21/).
- **SouthStreet de Filament** : cette page comprend des informations et des liens vers les projets liés au workflow d'amélioration progressive SouthStreet de Filament. (http://bkaprt.com/rrd/5-22/)

RÉFÉRENCES

Les URL abrégées sont numérotées dans l'ordre ; les URL complètes sont listées ci-dessous à titre de référence.

Introduction

0-01 http://www.wired.com/2014/01/internet-org-hackathon-low-end-rules/

0-02 http://www.moneyweb.co.za/archive/asia-mobile-internets-tomorrow/

0-03 http://appleinsider.com/articles/12/02/17/apple_sold_more_ios_devices_in_2011_than_total_macs_in_28_years

0-04 http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html

0-05 http://www.pewinternet.org/fact-sheets/mobile-technology-fact-sheet/

0-06 http://opensignal.com/reports/fragmentation-2013/

0-07 https://twitter.com/Cennydd/status/362269441645481984

0-08 http://alistapart.com/article/responsive-web-design

0-09 http://trentwalton.com/2014/03/10/device-agnostic/

0-10 https://www.flickr.com/photos/janitors/12907608763

0-11 https://www.flickr.com/photos/scottvanderchijs/5453911636

0-12 https://www.apple.com/accessibility/osx/voiceover/

0-13 https://www.thinkwithgoogle.com/research-studies/the-new-multi-screen-world-study.html

0-14 http://developer.android.com/about/dashboards/index.html

0-15 http://dev.opera.com

0-16 http://www.guypo.com/mobile/what-are-responsive-websites-made-of/

0-17 http://httparchive.org/interesting.php?a=All&l=Apr%2015%202014

0-18 http://minus.com/msM8y8nyh#1e

0-19 http://www.webperformancetoday.com/2013/03/19/new-findings-typical-leading-european-commerce-site-takes-7-04-seconds-to-load/

Chapitre 1

1-01 http://trentwalton.com/2011/05/10/fit-to-scale/

1-02 http://the-pastry-box-project.net/dan-mall/2012-september-12/

1-03 https://twitter.com/brad_frost/status/191977076000161793

1-04 http://webtypography.net/2.1.2

1-05 http://webtypography.net

1-06 http://daverupert.com/2013/04/responsive-deliverables/

1-07 http://getbootstrap.com/

1-08 http://www.lukew.com/ff/entry.asp?1569

1-09 http://demos.jquerymobile.com/1.4.2/table-reflow/

1-10 http://demos.jquerymobile.com/1.4.2/table-column-toggle/

1-11 http://bradfrost.github.io/this-is-responsive/patterns.html

1-12 http://touchlab.mit.edu/publications/2003_009.pdf

1-13 http://www.smashingmagazine.com/2012/02/21/finger-friendly-design-ideal-mobile-touchscreen-target-sizes/

1-14 http://static.lukew.com/TouchGestureCards.pdf

1-15 https://github.com/filamentgroup/tappy

1-16 https://github.com/ftlabs/fastclick/

1-17 http://www.w3.org/WAI/intro/aria

1-18 http://www.nytimes.com/2013/12/30/opinion/america-in-2013-as-told-in-charts.html

1-19 http://filamentgroup.com/lab/grade_components/

1-20 http://adactio.com/journal/6692/

Chapitre 2

2-01 http://alistapart.com/article/testing-websites-in-game-console-browsers

2-02 https://twitter.com/anna_debenham/status/246613439814971393

2-03 http://www.lukew.com/ff/entry.asp?1333

2-04 http://trentwalton.com/2013/03/19/type-touch/

2-05 https://www.flickr.com/photos/frankieroberto/2317229560/

2-06 http://www.slideshare.net/bryanrieger/rethinking-the-mobile-web-by-yiibu

2-07 http://blog.cloudfour.com/the-ems-have-it-proportional-media-queries-ftw/

2-08 http://trentwalton.com/2013/01/16/windows-phone-8-viewport-fix

2-09 http://caniuse.com

2-10 http://www.stucox.com/blog/the-good-and-bad-of-level-4-media-queries

2-11 http://alistapart.com/article/testdriven

2-12 http://modernizr.com/

2-13 http://dev.w3.org/csswg/css-conditional/#at-supports

2-14 http://dev.w3.org/csswg/css-conditional/#support-definition

2-15 https://github.com/Modernizr/Modernizr/wiki/Undetectables

2-16 http://filamentgroup.com/lab/overthrow

2-17 https://github.com/filamentgroup/fixed-fixed

2-18 https://github.com/aFarkas/html5shiv/#why-is-it-called-a-shiv

2-19 https://github.com/aFarkas/html5shiv

2-20 http://remysharp.com/2010/10/08/what-is-a-polyfill/

2-21 https://github.com/paulirish/matchMedia.js

2-22 https://github.com/scottjehl/Respond
2-23 http://adactio.com/journal/5964/
2-24 http://bradfrostweb.com/blog/mobile/test-on-real-mobile-devices-without-breaking-the-bank/
2-25 http://opendevicelab.com
2-26 https://www.flickr.com/photos/lukew/6171909286/
2-27 http://www.browserstack.com

Chapitre 3

3-01 http://contentsmagazine.com/articles/10-timeframes/
3-02 http://httparchive.org
3-03 http://moto.oakley.com
3-04 http://www.stevesouders.com/blog/2011/09/21/making-a-mobile-connection/
3-05 http://devtoolsecrets.com
3-06 https://developers.google.com/speed/pagespeed/insights
3-07 http://webpagetest.org/
3-08 http://timkadlec.com/2014/01/fast-enough/#comment-1200946500
3-09 http://calendar.perfplanet.com/2013/holistic-performance
3-10 http://timkadlec.com/2014/05/performance-budgeting-with-grunt
3-11 http://imageoptim.com
3-12 http://optipng.sourceforge.net
3-13 http://jpegclub.org/jpegtran
3-14 https://github.com/gruntjs/grunt-contrib-imagemin
3-15 http://2012.dconstruct.org
3-16 http://www.gzip.org/deflate.html
3-17 http://html5boilerplate.com
3-18 https://developers.google.com/speed/docs/best-practices/caching
3-19 https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Social_API/Service_worker_API_reference
3-20 http://www.html5rocks.com/en/tutorials/appcache/beginner
3-21 https://incident57.com/codekit/
3-22 http://gruntjs.com

Chapitre 4

4-01 http://www.lukew.com/ff/entry.asp?933
4-02 http://24ways.org/2011/conditional-loading-for-responsive-designs
4-03 http://adactio.com/journal/5042/
4-04 https://github.com/filamentgroup/Ajax-Include-Pattern

4-05 http://filamentgroup.com/lab/ajax_includes_modular_content
4-06 https://github.com/filamentgroup/AppendAround
4-07 http://filamentgroup.github.io/AppendAround/
4-08 http://httparchive.org/interesting.php#renderStart
4-09 https://github.com/scottjehl/css-inapplicable-load
4-10 https://developers.google.com/speed/pagespeed/insights/
4-11 http://paul.kinlan.me/detecting-critical-above-the-fold-css/
4-12 https://github.com/filamentgroup/grunt-criticalcss/
4-13 https://github.com/filamentgroup/loadCSS
4-14 http://timkadlec.com/2012/04/media-query-asset-downloading-results/
4-15 http://boazsender.github.io/datauri
4-16 http://www.mobify.com/blog/data-uris-are-slow-on-mobile
4-17 http://filamentgroup.com/lab/rwd_img_compression
4-18 http://responsiveimages.org/
4-19 http://www.w3.org/TR/html-picture-element/
4-20 http://scottjehl.github.io/picturefill/
4-21 http://css-tricks.com/examples/IconFont/
4-22 http://filamentgroup.com/lab/bulletproof_icon_fonts
4-23 https://github.com/filamentgroup/a-font-garde
4-24 http://css-tricks.com/stackicons-icon-fonts
4-25 https://docs.google.com/presentation/d/1CNQLbqCokrocy_fZrM5fZ-YmQ2JgEADRh3qR6RbOOGk/edit?pli=1#slide=id.p
4-26 http://jakearchibald.com/2013/animated-line-drawing-svg/
4-27 http://css-tricks.com/svg-sprites-use-better-icon-fonts/
4-28 https://github.com/filamentgroup/grunticon
4-29 http://grumpicon.com
4-30 http://stet.editorially.com
4-31 http://dev.w3.org/csswg/css-font-loading
4-32 https://twitter.com/jaffathecake/status/207096228339658752
4-33 https://github.com/filamentgroup/shoestring
4-34 https://github.com/jquery/jquery#how-to-build-your-own-jquery
4-35 https://github.com/filamentgroup/loadJS
4-36 http://responsivenews.co.uk/post/18948466399/cutting-the-mustard
4-37 https://github.com/filamentgroup/enhance/

Ressources

5-01 http://www.browserstack.com/
5-02 http://caniuse.com

5-03 http://www.akamai.com/stateoftheinternet

5-04 http://gs.statcounter.com/#all-browser_version_partially_combined-wwmonthly-201307-201407

5-05 http://filamentgroup.com/lab/grade-the-components.html

5-06 http://www.webpagetest.org

5-07 https://developers.google.com/speed/pagespeed/insights

5-08 http://timkadlec.com/2014/05/performance-budgeting-with-grunt

5-09 http://timkadlec.com/2013/01/setting-a-performance-budget

5-10 http://bradfrostweb.com/blog/mobile/test-on-real-mobile-devices-without-breaking-the-bank

5-11 http://stevesouders.com/

5-12 http://www1.practicalperformanceanalyst.com/2014/06/28/building-a-performance-culture-google-io-2014

5-13 http://opendevicelab.com

5-14 https://github.com/igrigorik/http-client-hints

5-15 http://lists.w3.org/Archives/Public/www-style

5-16 http://lists.w3.org/Archives/Public/www-style/2013Feb/0131.html

5-17 http://modernizr.com

5-18 http://dev.opera.com/articles/responsive-images

5-19 http://ricg.org

5-20 http://scottjehl.github.io/picturefill

5-21 http://filamentgroup.com/code

5-22 https://github.com/filamentgroup/Southstreet/

INDEX

À PROPOS DE A BOOK APART

Nous couvrons les sujets émergents et essentiels du design et du développement web avec style, clarté et surtout brièveté - car les professionnels du design et du développement n'ont pas de temps à perdre !

À PROPOS DE L'AUTEUR

Web designer et développeur, Scott Jehl travaille au sein de la brillante équipe du Filament Group, pour des clients tels que le Boston Globe, LEGO Systems, Inc., Global News Canada, eBay et bien d'autres. En 2010, il a coécrit le livre *Designing with Progressive Enhancement* et a participé à des conférences telles que An Event Apart, Breaking Development et Mobilism. Scott Jehl participe activement à la communauté open source sur GitHub en publiant des projets axés sur des pratiques accessibles, durables et performantes pour un développement multiplateforme. Il vit à Seagrove Beach, en Floride.