

Web Development Essentials

Version 1.0
Français

030

Table of Contents

THÈME 031: DÉVELOPPEMENT DE LOGICIELS ET TECHNOLOGIES WEB	1
031.1 Bases du développement logiciel	2
031.1 Leçon 1	3
Introduction	3
Code source	3
Langages de programmation	6
Exercices guidés	12
Exercices d'exploration	13
Résumé	14
Réponses aux exercices guidés	15
Réponses aux exercices d'exploration	16
031.2 Architecture des applications Web	17
031.2 Leçon 1	19
Introduction	19
Clients et serveurs	19
Côté client	20
Variétés de clients Web	21
Langages d'un client Web	22
Côté serveur	24
Traitement des chemins à partir de requêtes	25
Systèmes de gestion des bases de données	25
Maintenance du contenu	26
Exercices guidés	27
Exercices d'exploration	28
Résumé	29
Réponses aux exercices guidés	30
Réponses aux exercices d'exploration	31
031.3 Bases du HTTP	32
031.3 Leçon 1	34
Introduction	34
Requête du client	35
En-tête de la réponse	38
Contenu statique et dynamique	40
Mise en cache	41
Sessions HTTP	42
Exercices guidés	44
Exercices d'exploration	45
Résumé	46

Réponses aux exercices guidés	47
Réponses aux exercices d'exploration	48
THÈME 032: BALISAGE DE DOCUMENTS HTML	49
032.1 Anatomie d'un document HTML	50
032.1 Leçon 1	51
Introduction	51
Anatomie d'un document HTML	51
En-tête du document	55
Exercices guidés	59
Exercices d'exploration	60
Résumé	61
Réponses aux exercices guidés	62
Réponses aux exercices d'exploration	63
032.2 Sémantique HTML et hiérarchie des documents	65
032.2 Leçon 1	67
Introduction	67
Texte	68
Titres	68
Sauts de ligne	70
Lignes horizontales	71
Listes HTML	72
Mise en forme du texte	77
Texte préformaté	83
Regroupement des éléments	83
Structure de page HTML	85
Exercices guidés	93
Exercices d'exploration	94
Résumé	95
Réponses aux exercices guidés	97
Réponses aux exercices d'exploration	99
032.3 Références HTML et ressources intégrées	106
032.3 Leçon 1	107
Introduction	107
Contenu Intégré	107
Liens	111
Exercices guidés	114
Exercices d'exploration	115
Résumé	116
Réponses aux exercices guidés	117
Réponses aux exercices d'exploration	118

032.4 Formulaires HTML	119
032.4 Leçon 1	120
Introduction	120
Formulaires HTML simples	120
Saisie de textes volumineux : <code>textarea</code>	128
Listes d'options	129
Type d'élément <code>hidden</code>	133
Type d'entrée <code>file</code>	133
Boutons d'action	134
Action et méthodes de formulaire	135
Exercices guidés	137
Exercices d'exploration	138
Résumé	139
Réponses aux exercices guidés	140
Réponses aux exercices d'exploration	141
THÈME 033: STYLES DE CONTENU CSS	143
033.1 Bases du CSS	144
033.1 Leçon 1	145
Introduction	145
Appliquer des styles	146
Exercices guidés	153
Exercices d'exploration	154
Résumé	155
Réponses aux exercices guidés	156
Réponses aux exercices d'exploration	157
033.2 Sélecteurs CSS et application de style	158
033.2 Leçon 1	159
Introduction	159
Styles pour toute la page	159
Sélecteurs restrictifs	161
Sélecteurs spéciaux	167
Exercices guidés	168
Exercices d'exploration	169
Résumé	170
Réponses aux exercices guidés	171
Réponses aux exercices d'exploration	172
033.3 Stylisation CSS	173
033.3 Leçon 1	174
Introduction	174
Propriétés et valeurs communes de CSS	174

Couleurs	174
Arrière-plan	177
Bordures	179
Valeurs d'unité	179
Unités relatives	180
Propriétés des polices et du texte	181
Exercices guidés	184
Exercices d'exploration	185
Résumé	186
Réponses aux exercices guidés	187
Réponses aux exercices d'exploration	188
033.4 Modèle de boîte CSS et mise en page	189
033.4 Leçon 1	190
Introduction	190
Flux normal	190
Personnalisation du flux normal	197
Conception réactive	202
Exercices guidés	203
Exercices d'exploration	204
Résumé	205
Réponses aux exercices guidés	206
Réponses aux exercices d'exploration	207
THÈME 034: PROGRAMMATION JAVASCRIPT	208
034.1 Exécution et syntaxe JavaScript	209
034.1 Leçon 1	210
Introduction	210
Exécution de JavaScript dans le navigateur	210
Console du navigateur	213
Instructions JavaScript	214
Commentaires JavaScript	215
Exercices guidés	217
Exercices d'exploration	218
Résumé	219
Réponses aux exercices guidés	220
Réponses aux exercices d'exploration	221
034.2 Structures de données JavaScript	222
034.2 Leçon 1	223
Introduction	223
Langages de haut niveau	223
Déclaration des constantes et des variables	224

Types de valeurs	226
Opérateurs	231
Exercices Guidés	234
Exercices d'exploration	235
Résumé	236
Réponses aux exercices guidés	237
Réponses aux exercices d'exploration	238
034.3 Structures de contrôle et fonctions JavaScript	239
034.3 Leçon 1	240
Introduction	240
Instructions If	240
Structures switch	245
Boucles	247
Exercices guidés	252
Exercices d'exploration	253
Résumé	254
Réponses aux exercices guidés	255
Réponses aux exercices d'exploration	256
034.3 Leçon 2	258
Introduction	258
Définir une fonction	258
Fonctions récursives	263
Exercices guidés	267
Exercices d'exploration	268
Résumé	269
Réponses aux exercices guidés	270
Réponses aux exercices d'exploration	271
034.4 Manipulation en JavaScript du contenu et du style des sites web	272
034.4 Leçon 1	273
Introduction	273
Interagir avec le DOM	273
Contenu HTML	274
Sélection d'éléments spécifiques	276
Travailler avec les attributs	277
Travailler avec les classes	281
Gestionnaires d'événements	282
Exercices guidés	285
Exercices d'exploration	286
Résumé	287
Réponses aux exercices guidés	288

Réponses aux exercices d'exploration	289
THÈME 035: PROGRAMMATION DE SERVEUR NODEJS	290
035.1 Bases de Node.js	291
035.1 Leçon 1	292
Introduction	292
Pour commencer	293
Exercices guidés	300
Exercices d'exploration	301
Résumé	302
Réponses aux exercices guidés	303
Réponses aux exercices d'exploration	304
035.2 Bases de NodeJS Express	305
035.2 Leçon 1	307
Introduction	307
Script initial pour le serveur	307
Routes	310
Ajustements de la réponse	314
Sécurité des cookies	317
Exercices guidés	318
Exercices d'exploration	319
Résumé	320
Réponses aux exercices guidés	321
Réponses aux exercices d'exploration	322
035.2 Leçon 2	323
Introduction	323
Fichiers statiques	324
Sortie formatée	325
Modèles	329
HTML Templates	331
Exercices guidés	335
Exercices d'exploration	336
Résumé	337
Réponses aux exercices guidés	338
Réponses aux exercices d'exploration	339
035.3 Bases de SQL	340
035.3 Leçon 1	341
Introduction	341
SQL	341
SQLite	342
Ouverture de la base de données	343

Structure d'une table	344
Entrée des données	345
Requêtes	346
Modification du contenu de la base de données.....	347
Fermeture de la base de données	349
Exercices guidés.....	350
Exercices d'exploration	351
Résumé	352
Réponses aux exercices guidés	353
Réponses aux exercices d'exploration	354
Impression	355



**Linux
Professional
Institute**

Thème 031: Développement de logiciels et technologies Web



031.1 Bases du développement logiciel

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 031.1

Valeur

1

Domaines de connaissance les plus importants

- Comprendre ce qu'est le code source
- Comprendre les principes des compilateurs et des interpréteurs
- Comprendre le concept de bibliothèque
- Comprendre les concepts de la programmation fonctionnelle, procédurale et orientée objet.
- Connaissance des caractéristiques communes des éditeurs de code source et des environnements de développement intégrés (IDE)
- Connaissance des systèmes de contrôle de version
- Connaissance des tests de logiciels
- Connaissance des principaux langages de programmation (C, C++, C#, Java JavaScript, Python, PHP)



031.1 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	031 Développement de logiciels et technologies Web
Objectif :	031.1 Bases du développement logiciel
Leçon :	1 sur 1

Introduction

Les tout premiers ordinateurs étaient programmés par le biais d'un processus épuisant consistant à brancher des câbles dans des prises. Les informaticiens se sont rapidement lancés dans une recherche sans fin de moyens simples pour indiquer à l'ordinateur ce qu'il doit faire. Ce chapitre présente les outils de programmation. Il aborde les principales façons dont les instructions textuelles—les langages de programmation—représentent les tâches qu'un programmeur veut accomplir, ainsi que les outils qui transforment le programme en une forme appelée *langage machine* qu'un ordinateur peut exécuter.

NOTE | Dans ce texte, les termes *programme* et *application* sont utilisés indifféremment.

Code source

Un programmeur développe normalement une application en écrivant une description textuelle, appelée *code source*, de la tâche souhaitée. Le code source est rédigé dans un *langage de programmation* soigneusement défini qui représente ce que l'ordinateur peut faire dans une abstraction de haut niveau que les humains peuvent comprendre. Des outils ont également été

développés pour permettre aux programmeurs et aux non-programmeurs d'exprimer leurs pensées visuellement, mais l'écriture du code source reste la manière prédominante de programmer.

De la même manière qu'un langage naturel comporte des noms, des verbes et des constructions pour exprimer des idées de manière structurée, les mots et la ponctuation d'un langage de programmation sont des représentations symboliques des opérations qui seront effectuées sur la machine.

En ce sens, le code source n'est pas très différent de tout autre texte dans lequel l'auteur utilise des règles bien établies d'un langage naturel pour communiquer avec le lecteur. Dans le cas du code source, le "lecteur" est la machine, donc le texte ne peut pas contenir des ambiguïtés ou des incohérences—même subtiles.

Comme tout texte qui traite d'un sujet en profondeur, le code source doit également être bien structuré et organisé logiquement lors du développement d'applications complexes. Les programmes très simples et les exemples didactiques peuvent être stockés dans les quelques lignes d'un seul fichier texte, qui contient tout le code source du programme. Les programmes plus complexes peuvent être subdivisés en milliers de fichiers, chacun comportant des milliers de lignes.

Le code source des applications professionnelles doit être organisé en différents dossiers, généralement associés à un objectif particulier. Par exemple, un programme de discussion peut être organisé en deux dossiers : l'un contenant les fichiers de code qui gèrent la transmission et la réception des messages sur le réseau, et l'autre contenant les fichiers qui construisent l'interface et réagissent aux actions de l'utilisateur. En effet, il est courant d'avoir de nombreux dossiers et sous-dossiers contenant des fichiers de code source dédiés à des tâches très spécifiques au sein de l'application.

En plus, le code source n'est pas toujours isolé dans ses propres fichiers, avec le tout écrit dans un seul langage. Dans les applications web, par exemple, un document HTML peut intégrer du code JavaScript pour compléter le document par des fonctionnalités supplémentaires.

Éditeurs de code et EDI

La diversité des méthodes d'écriture du code source peut être intimidante. C'est pourquoi de nombreux développeurs tirent parti d'outils qui les aident à écrire et à tester le programme.

Le fichier de code source n'est qu'un fichier texte. En tant que tel, il peut être édité par n'importe quel éditeur de texte, aussi simple soit-il. Pour faciliter la distinction entre le code source et le texte brut, chaque langage adopte une extension de nom de fichier explicite : `.c` pour le langage C, `.py` pour Python, `.js` pour JavaScript, etc. Les éditeurs généralistes comprennent souvent suffisamment bien le code source des langages populaires pour ajouter de l'italique, des couleurs et des indentations afin de rendre le code compréhensible.

Tous les développeurs ne choisissent pas d'éditer le code source dans un éditeur généraliste. Un *environnement de développement intégré* (EDI) fournit un éditeur de texte ainsi que des outils pour aider le programmeur à éviter les erreurs syntaxiques et les incohérences évidentes. Ces éditeurs sont particulièrement recommandés pour les programmeurs moins expérimentés, mais les programmeurs expérimentés les utilisent également.

Les EDI les plus utilisés, tels que Visual Studio, Eclipse et Xcode, observent intelligemment ce que le programmeur tape, suggérant fréquemment des mots à utiliser (autocomplétion) et vérifiant le code en temps réel. Les EDI peuvent même proposer un débogage et des tests automatisés pour identifier les problèmes chaque fois que le code source est modifié.

Certains programmeurs plus expérimentés optent pour des éditeurs moins intuitifs tels que Vim, qui offrent une plus grande flexibilité et ne nécessitent pas l'installation de paquets supplémentaires. Ces programmeurs utilisent des outils externes et autonomes pour ajouter les fonctionnalités qui sont intégrées lorsque vous utilisez un EDI.

Maintenance du code

Que ce soit dans un EDI ou à l'aide d'outils autonomes, il est important d'utiliser une sorte de *système de gestion de versions* (SGV). Le code source évolue constamment, car des défauts imprévus doivent être corrigés et des améliorations doivent être intégrées. Une conséquence inévitable de cette évolution est que les corrections et les améliorations peuvent interférer avec d'autres parties de l'application dans une grande base de code. Les outils de gestion de versions tels que Git, Subversion et Mercurial enregistrent toutes les modifications apportées au code et l'auteur de la modification, ce qui vous permet de retracer et éventuellement de récupérer une modification non réussie.

En outre, les outils de gestion de versions permettent à chaque développeur de l'équipe de travailler sur une copie des fichiers de code source sans interférer avec le travail des autres programmeurs. Une fois que les nouvelles versions du code source sont prêtes et testées, les corrections ou améliorations apportées à une copie peuvent être intégrées par les autres membres de l'équipe.

Git, le système de gestion de versions le plus populaire à l'heure actuelle, permet à plusieurs copies indépendantes d'un dépôt d'être maintenues par différentes personnes, qui partagent leurs modifications comme elles le souhaitent. Cependant, qu'elles utilisent un système de gestion de versions décentralisé ou centralisé, la plupart des équipes maintiennent un référentiel de confiance dont le code source et les ressources sont fiables. Plusieurs services en ligne proposent le stockage de référentiels de code source. Les plus populaires de ces services sont GitHub et GitLab, mais Savannah, du projet GNU, mérite également d'être mentionné.

Langages de programmation

Il existe une grande variété de langages de programmation ; chaque décennie voit l'invention de nouveaux langages. Chaque langage de programmation a ses propres règles et il est recommandé pour des objectifs particuliers. Bien que les langages présentent des différences superficielles en matière de syntaxe et de mots-clés, ce qui les distingue réellement, ce sont les approches conceptuelles profondes qu'ils représentent, appelées *paradigmes*.

Paradigmes

Les paradigmes définissent les principes sur lesquels repose un langage de programmation, notamment en ce qui concerne la manière dont le code source doit être structuré.

Le développeur part du paradigme du langage pour formuler les tâches à accomplir par la machine. Ces tâches, à leur tour, sont exprimées symboliquement avec les mots et les constructions syntaxiques offerts par le langage.

Le langage de programmation est *procédural* lorsque les instructions présentées dans le code source sont exécutées dans un ordre séquentiel, comme un script de film. Si le code source est segmenté en fonctions ou en sous-programmes, une fonction principale se charge d'appeler les autres fonctions dans l'ordre.

Le code suivant est un exemple de langage procédural. Écrit en C, il définit des variables pour représenter le côté `side`, la surface `area` et le volume `volume` de formes géométriques. La valeur de la variable `side` est assignée dans `main()`, qui est la fonction principale appelée lorsque le programme est exécuté. Les variables `area` et `volume` sont calculées dans les sous-programmes `square()` et `cube()` qui précèdent la fonction principale :

```
#include <stdio.h>

float side;
float area;
float volume;

void square(){ area = side * side; }

void cube(){ volume = area * side; }

int main(){
    side = 2;
    square();
    cube();
    printf("Volume: %f\n", volume);
```

```
return 0;
}
```

L'ordre des actions définies dans `main()` détermine la séquence des états du programme, caractérisée par la valeur des variables `side`, `area`, et `volume`. L'exemple se termine après avoir affiché la valeur de `volume` avec l'instruction `printf`.

D'autre part, le paradigme de la *programmation orientée objet* (POO) a pour principale caractéristique de séparer l'état du programme en sous-états indépendants. Ces sous-états et les opérations associées sont les *objets*, appelés ainsi parce qu'ils ont une existence plus ou moins indépendante au sein du programme et parce qu'ils ont des objectifs spécifiques.

Les paradigmes distincts ne limitent pas nécessairement le type de tâche qui peut être effectué par un programme. Le code de l'exemple précédent peut être réécrit selon le paradigme de la POO en utilisant le langage C++ :

```
#include <iostream>

class Cube {
    float side;
public:
    Cube(float s){ side = s; }
    float volume() { return side * side * side; }
};

int main(){
    float side = 2;
    Cube cube(side);
    std::cout << "Volume: " << cube.volume() << std::endl;
    return 0;
}
```

La fonction `main()` est toujours présente. Mais il y a maintenant un nouveau mot, `class`, qui introduit la définition d'un objet. La classe définie, nommée `Cube`, contient ses propres variables et sous-programmes. En POO, une variable est aussi appelée un *attribut* et un sous-programme est appelé une *méthode*.

L'explication de tout le code C++ de l'exemple n'entre pas dans le cadre de ce chapitre. Ce qui est important pour nous ici est que `Cube` contient l'attribut `side` et deux méthodes. La méthode `volume()` calcule le volume du cube.

Il est possible de créer plusieurs objets indépendants à partir d'une même classe, et les classes peuvent être composées d'autres classes.

N'oubliez pas que ces mêmes fonctionnalités peuvent être écrites différemment et que les exemples de ce chapitre sont simplifiés à l'extrême. Le C et le C++ possèdent des fonctionnalités beaucoup plus sophistiquées qui permettent des constructions beaucoup plus complexes et plus pratiques.

La plupart des langages de programmation n'imposent pas rigoureusement un paradigme, mais permettent aux programmeurs de choisir divers aspects d'un paradigme ou d'un autre. JavaScript, par exemple, intègre des aspects de différents paradigmes. Le programmeur peut décomposer l'ensemble du programme en fonctions qui ne partagent pas d'état commun entre elles :

```
function cube(side){  
  return side*side*side;  
}  
  
console.log("Volume: " + cube(2));
```

Bien que cet exemple soit similaire à la programmation procédurale, notez que la fonction reçoit une copie de toutes les informations nécessaires à son exécution et qu'elle produit toujours le même résultat pour le même paramètre, indépendamment des changements qui se produisent en dehors de la portée de la fonction. Ce paradigme, appelé *fonctionnel*, est fortement influencé par le formalisme mathématique, où chaque opération est autosuffisante.

Un autre paradigme couvre les langages *déclaratifs*, qui décrivent les états dans lesquels vous voulez que le système se trouve. Un langage déclaratif peut déterminer comment atteindre les états spécifiés. SQL, le langage universel d'interrogation des bases de données, est parfois qualifié de langage déclaratif, bien qu'il occupe en réalité une niche unique dans le panthéon de la programmation.

Il n'existe pas de paradigme universel qui puisse être adopté dans n'importe quel contexte. Le choix du langage peut également être restreint par les langages supportés par la plate-forme ou l'environnement d'exécution où le programme sera utilisé.

Une application web qui sera utilisée par le navigateur, par exemple, devra être écrite en JavaScript, qui est un langage universellement supporté par les navigateurs. (Quelques autres langages peuvent être utilisés car ils fournissent des convertisseurs pour créer du JavaScript). Ainsi, pour le navigateur Web—parfois appelé *côté client* ou *front end* de l'application Web—le développeur devra utiliser les paradigmes autorisés en JavaScript. Le côté serveur ou back-end de l'application, qui traite les requêtes du navigateur, est normalement programmé dans un langage différent ; PHP est le plus utilisé à cette fin.

Indépendamment du paradigme, chaque langage dispose de bibliothèques de fonctions prédéfinies qui peuvent être incorporées dans le code. Les fonctions mathématiques—comme celles illustrées dans l'exemple—n'ont pas besoin d'être implémentées à partir de zéro, car le langage dispose déjà de la fonction prête à être utilisée. JavaScript, par exemple, fournit l'objet `Math` avec les opérations mathématiques les plus courantes.

Des fonctions encore plus spécialisées sont généralement disponibles auprès du fournisseur du langage ou de développeurs tiers. Ces bibliothèques de ressources supplémentaires peuvent être sous forme de code source, c'est-à-dire dans des fichiers supplémentaires qui sont incorporés dans le fichier où ils seront utilisés. En JavaScript, l'incorporation se fait avec `import from` :

```
import { OrbitControls } from 'modules/OrbitControls.js';
```

Ce type d'importation, où la ressource incorporée est également un fichier de code source, est le plus souvent utilisé dans les langages dits *interprétés*. Les *langues compilées* permettent, entre autres, l'incorporation de fonctionnalités pré-compilées dans le langage machine, c'est-à-dire des bibliothèques *compilées*. La section suivante explique les différences entre ces types de langages.

Compilateurs et interpréteurs

Comme nous le savons déjà, le code source est une représentation symbolique d'un programme qui doit être traduit en langage machine pour être exécuté.

Grosso modo, il y a deux façons de faire la traduction : convertir le code source au préalable pour une exécution future, ou convertir le code au moment de son exécution. Les langages de la première modalité sont appelés *langages compilés* et les langages de la seconde modalité sont appelés *langages interprétés*. Certains langages interprétés proposent la compilation en option, afin que le programme puisse être exécuté plus rapidement.

Dans les langages compilés, il existe une distinction claire entre le code source du programme et le programme lui-même, qui sera exécuté par l'ordinateur. Une fois compilé, le programme ne fonctionne généralement que sur le système d'exploitation et la plate-forme pour lesquels il a été compilé.

Dans un langage interprété, le code source lui-même est traité comme le programme, et le processus de conversion en langage machine est transparent pour le programmeur. Pour un langage interprété, il est courant d'appeler le code source un *script*. L'interpréteur traduit le script en langage machine pour le système sur lequel il est exécuté.

Compilation et compilateurs

Le langage de programmation C est l'un des exemples les plus connus de langage compilé. Les principaux atouts du langage C sont sa flexibilité et ses performances. Le langage C permet de programmer aussi bien les superordinateurs de haute performance que les microcontrôleurs des appareils ménagers. D'autres exemples de langages compilés populaires sont C++ et C# (C sharp). Comme leur nom l'indique, ces langages s'inspirent du C, mais incluent des fonctionnalités qui prennent en charge le paradigme orienté objet.

Le même programme écrit en C ou en C++ peut être compilé pour différentes plates-formes, en ne nécessitant que peu ou pas de modifications du code source. C'est le compilateur qui définit la plate-forme cible du programme. Il existe des compilateurs spécifiques à une plate-forme ainsi que des compilateurs multi-plateformes tels que GCC (qui signifie *GNU Compiler Collection*) qui peuvent produire des programmes binaires pour de nombreuses architectures distinctes.

NOTE

Il existe également des outils qui automatisent le processus de compilation. Au lieu d'invoquer directement le compilateur, le programmeur crée un fichier indiquant les différentes étapes de compilation à exécuter automatiquement. L'outil traditionnel utilisé à cette fin est `make`, mais un certain nombre d'outils plus récents tels que Maven et Gradle sont également largement utilisés. L'ensemble du processus de construction est automatisé lorsque vous utilisez un EDI.

Le processus de compilation ne génère pas toujours un programme binaire en langage machine. Il existe des langages compilés qui produisent un programme dans un format appelé génériquement *bytecode*. Comme un script, le bytecode n'est pas dans un langage spécifique à une plate-forme, il nécessite donc un programme interpréteur qui le traduit en langage machine. Dans ce cas, le programme interpréteur est simplement appelé un *environnement d'exécution* ou *runtime*.

Le langage Java adopte cette approche, de sorte que les programmes compilés écrits en Java peuvent être utilisés sur différents systèmes d'exploitation. Malgré son nom, Java n'est pas lié à JavaScript.

Le bytecode est plus proche du langage machine que du code source, de sorte que son exécution tend à être comparativement plus rapide. Comme il y a toujours un processus de conversion pendant l'exécution du bytecode, il est difficile d'obtenir les mêmes performances qu'un programme équivalent compilé en langage machine.

Interprétation et interpréteurs

Dans les langages interprétés tels que JavaScript, Python et PHP, le programme n'a pas besoin d'être pré-compilé, ce qui facilite son développement et sa modification. Au lieu de le compiler, le script est exécuté par un autre programme appelé interpréteur. En général, l'interpréteur d'un langage porte le nom du langage lui-même. L'interpréteur d'un script Python, par exemple, est un programme appelé

python. L'interpréteur de JavaScript est le plus souvent le navigateur web, mais les scripts peuvent aussi être exécutés par le programme node en dehors d'un navigateur. Parce qu'il est converti en instructions binaires à chaque fois qu'il est exécuté, un programme en langage interprété a tendance à être plus lent que son équivalent en langage compilé.

Rien n'empêche une même application d'avoir des composants écrits dans des langages différents. Si nécessaire, ces composants peuvent communiquer par le biais d'une *interface de programmation d'applications* (API : *application programming interface*) compréhensible par tous.

Le langage Python, par exemple, possède des capacités très sophistiquées d'exploration de données et de tabulation de données. Le développeur peut choisir Python pour écrire les parties du programme qui traitent de ces aspects et un autre langage, tel que C++, pour effectuer le traitement numérique plus lourd. Il est possible d'adopter cette stratégie même lorsqu'il n'existe pas d'API permettant une communication directe entre les deux composants. Par exemple, le code écrit en Python peut générer un fichier dans le format approprié pour être utilisé par un programme écrit en C++.

Bien qu'il soit possible d'écrire presque n'importe quel programme dans n'importe quel langage, le développeur doit adopter celui qui correspond le mieux à l'objectif de l'application. Ce faisant, vous bénéficiez de la réutilisation de composants déjà testés et bien documentés.

Exercices guidés

1. Quel type de programmes peut être utilisé pour éditer le code source ?

2. Quel type d'outils permet d'intégrer le travail de différents développeurs dans la même base de code ?

Exercices d'exploration

1. Supposons que vous vouliez écrire un jeu en 3D à jouer dans le navigateur. Les applications et les jeux Web sont programmés en JavaScript. Bien qu'il soit possible d'écrire toutes les fonctions graphiques à partir de zéro, il est plus productif d'utiliser une bibliothèque prête à l'emploi à cette fin. Quelles bibliothèques tierces offrent des possibilités d'animation 3D en JavaScript ?

2. Outre le PHP, quels autres langages peuvent être utilisés du côté serveur d'une application web ?

Résumé

Cette leçon couvre les concepts les plus essentiels du développement de logiciels. Le développeur doit connaître les principaux langages de programmation et le scénario d'utilisation approprié pour chacun d'eux. Cette leçon aborde les concepts et procédures suivants :

- Ce qu'est le code source.
- Les éditeurs de code source et les outils associés.
- Les paradigmes de programmation procédurale, orientée objet, fonctionnelle et déclarative.
- Caractéristiques des langages compilés et interprétés.

Réponses aux exercices guidés

1. Quel type de programmes peut être utilisé pour éditer le code source ?

En principe, tout programme capable d'éditer du texte brut.

2. Quel type d'outils permet d'intégrer le travail de différents développeurs dans la même base de code ?

Un système de gestion de versions ou de sources, tel que Git.

Réponses aux exercices d'exploration

1. Supposons que vous vouliez écrire un jeu en 3D à jouer dans le navigateur. Les applications et les jeux Web sont programmés en JavaScript. Bien qu'il soit possible d'écrire toutes les fonctions graphiques à partir de zéro, il est plus productif d'utiliser une bibliothèque prête à l'emploi à cette fin. Quelles bibliothèques tierces offrent des possibilités d'animation 3D en JavaScript ?

Il existe de nombreuses bibliothèques graphiques 3D pour JavaScript, telles que threejs et BabylonJS.

2. Outre le PHP, quels autres langages peuvent être utilisés du côté serveur d'une application web ?

Tout langage pris en charge par l'application du serveur HTTP utilisée sur la machine serveur. Quelques exemples sont Python, Ruby, Perl et JavaScript lui-même.



031.2 Architecture des applications Web

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 031.2

Valeur

2

Domaines de connaissance les plus importants

- Comprendre le principe du modèle client/serveur
- Comprendre le rôle des navigateurs web et connaître les navigateurs web les plus utilisés.
- Comprendre le rôle des serveurs web et des serveurs d'applications.
- Comprendre les technologies et les normes courantes de développement web
- Comprendre les principes des API
- Comprendre le principe des bases de données relationnelles et non relationnelles (NoSQL).
- Connaissance des systèmes de gestion de bases de données open source les plus utilisés
- Connaissance de REST et GraphQL
- Connaissance des applications à page unique
- Connaissance du packaging des applications web
- Connaissance de WebAssembly
- Connaissance des systèmes de gestion de contenu

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- Chrome, Edge, Firefox, Safari, Internet Explorer
- HTML, CSS, JavaScript

- SQLite, MySQL, MariaDB, PostgreSQL
- MongoDB, CouchDB, Redis



Linux
Professional
Institute

031.2 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	031 Développement de logiciels et technologies Web
Objectif :	031.2 Architecture des applications Web
Leçon :	1 sur 1

Introduction

Le mot *application* a un sens large dans le jargon technologique. Lorsque l'application est un programme traditionnel, exécuté localement et autosuffisant dans sa finalité, l'interface d'exploitation de l'application et les composants de traitement des données sont intégrés dans un seul "paquet". Une *application web* est différente car elle adopte le modèle client/serveur et sa partie client est basée sur le langage HTML, qui est obtenu du serveur et, en général, rendu par un navigateur.

Clients et serveurs

Dans le modèle client/serveur, une partie du travail est effectuée localement, du *côté du client*, et une partie du travail est effectuée à distance, du *côté du serveur*. Les tâches effectuées par chaque partie varient en fonction de l'objectif de l'application, mais en général, c'est au client de fournir une

interface à l'utilisateur et de présenter le contenu de manière attrayante. C'est au serveur qu'il revient d'exécuter l'aspect business de l'application, en traitant et en répondant aux requêtes faites par le client. Dans une application d'achat, par exemple, l'application client affiche une interface permettant à l'utilisateur de choisir et de payer les produits, mais la source de données et les enregistrements de la transaction sont conservés sur le serveur distant, auquel on accède via le réseau. Les applications web effectuent cette communication sur Internet, généralement via le protocole de transfert hypertexte (HTTP).

Une fois chargé par le navigateur, le côté client de l'application initie une interaction avec le serveur chaque fois que cela est nécessaire ou pratique. Les serveurs d'applications web offrent une *interface de programmation d'applications* (API : *Application Programming Interface*) qui définit les requêtes disponibles et la manière dont ces requêtes doivent être effectuées. Ainsi, le client construit une requête dans le format défini par l'API et l'envoie au serveur, qui vérifie les conditions préalables à la requête et renvoie la réponse appropriée.

Alors que le client, sous la forme d'une application mobile ou d'un navigateur de bureau, est un programme autonome en ce qui concerne l'interface utilisateur et les instructions de communication avec le serveur, le navigateur doit obtenir la page HTML et les composants associés (images, CSS et JavaScript) qui définissent l'interface et les instructions de communication avec le serveur.

Les langages de programmation et les plates-formes utilisés par le client et le serveur sont indépendants, mais utilisent un protocole de communication mutuellement compréhensible. La partie serveur est presque toujours exécutée par un programme sans interface graphique, qui fonctionne dans des environnements informatiques hautement disponibles afin d'être toujours prêt à répondre aux requêtes. En revanche, la partie client s'exécute sur tout appareil capable de restituer une interface HTML, comme les smartphones.

En plus d'être indispensable à certains usages, l'adoption du modèle client/serveur permet à une application d'optimiser plusieurs aspects du développement et de la maintenance, puisque chaque partie peut être conçue pour son usage spécifique. Une application qui affiche des cartes et des itinéraires, par exemple, n'a pas besoin d'avoir toutes les cartes stockées localement. Seules les cartes relatives à l'endroit qui intéresse l'utilisateur sont nécessaires, et seules ces cartes sont demandées au serveur central.

Les développeurs ont un contrôle direct sur le serveur, ils peuvent donc également modifier le client qui est fourni par celui-ci. Cela permet aux développeurs d'améliorer l'application, dans une plus ou moins grande mesure, sans que l'utilisateur ait besoin d'installer explicitement de nouvelles versions.

Côté client

Une application web devrait fonctionner de la même manière sur n'importe quel navigateur parmi

les plus populaires, pour autant que le navigateur soit à jour. Certains navigateurs peuvent être incompatibles avec des innovations récentes, mais seules les applications expérimentales utilisent des fonctionnalités qui ne sont pas encore largement adoptées.

Les problèmes d'incompatibilité étaient plus fréquents dans le passé, lorsque les différents navigateurs possédaient leur propre *moteur de rendu* et qu'il y avait moins de coopération dans la formulation et l'adoption de normes. Le moteur de rendu est le principal composant du navigateur, car il est chargé de transformer le HTML et les autres composants associés en éléments visuels et interactifs de l'interface. Certains navigateurs, notamment Internet Explorer, nécessitaient un traitement spécial dans le code afin de ne pas casser le fonctionnement attendu des pages.

Aujourd'hui, les différences entre les principaux navigateurs sont minimales, et les incompatibilités sont rares. En fait, les navigateurs Chrome et Edge utilisent le même moteur de rendu (appelé Blink). Le navigateur Safari, et d'autres navigateurs proposés sur l'App Store d'iOS, utilisent le moteur WebKit. Firefox utilise un moteur appelé Gecko. Ces trois moteurs représentent la quasi-totalité des navigateurs utilisés aujourd'hui. Bien que développés séparément, ces trois moteurs sont des projets open source et il existe une coopération entre leurs développeurs, ce qui facilite la compatibilité, la maintenance et l'adoption de normes.

Comme les développeurs de navigateurs ont fait beaucoup d'efforts pour rester compatibles, le serveur n'est normalement pas lié à un seul type de client. En principe, un serveur HTTP peut communiquer avec n'importe quel client qui est également capable de communiquer via HTTP. Dans une application cartographique, par exemple, le client peut être une application mobile ou un navigateur qui charge l'interface HTML depuis le serveur.

Variétés de clients Web

Il existe des applications mobiles et de bureau dont l'interface est rendue à partir du HTML et qui, comme les navigateurs, peuvent utiliser JavaScript comme langage de programmation. Cependant, contrairement au client chargé dans le navigateur, le HTML et les composants nécessaires au fonctionnement du client natif sont présents localement depuis l'installation de l'application. En fait, une application qui fonctionne de cette manière est pratiquement identique à une page HTML (les deux sont même susceptibles d'être rendues par le même moteur). Il existe également des *applications web progressives* (PWA : *Progressive Web Apps*), un mécanisme qui permet de packager des clients d'applications web pour une utilisation hors connexion, limitée aux fonctions qui ne nécessitent pas de communication immédiate avec le serveur. En ce qui concerne ce que l'application peut faire, il n'y a pas de différence entre l'exécution dans le navigateur ou le conditionnement dans une PWA, mais dans cette dernière, le développeur a plus de contrôle sur ce qui est stocké localement.

Le rendu d'interfaces à partir de HTML est une activité tellement récurrente que le moteur est

généralement un composant logiciel distinct, présent dans le système d'exploitation. Sa présence en tant que composant séparé permet à différentes applications de l'incorporer sans avoir à l'intégrer dans le paquetage de l'application. Ce modèle délègue également la maintenance du moteur de rendu au système d'exploitation, ce qui facilite les mises à jour. Il est très important de maintenir à jour un composant aussi crucial afin d'éviter d'éventuelles défaillances.

Quelle que soit leur méthode de livraison, les applications écrites en HTML s'exécutent sur une couche d'abstraction créée par le moteur, qui fonctionne comme un environnement d'exécution isolé. En particulier, dans le cas d'un client qui s'exécute sur le navigateur, l'application n'a à sa disposition que les ressources offertes par le navigateur. Les fonctions de base, telles que l'interaction avec les éléments de la page et la demande de fichiers par HTTP, sont toujours disponibles. Les ressources susceptibles de contenir des informations sensibles, telles que l'accès aux fichiers locaux, à la localisation géographique, à la caméra et au microphone, nécessitent une autorisation explicite de l'utilisateur avant que l'application ne puisse les utiliser.

Langages d'un client Web

L'élément central d'un client d'application web qui s'exécute sur le serveur est le document HTML. En plus de présenter de manière structurée les éléments d'interface que le navigateur affiche, le document HTML contient les adresses de tous les fichiers nécessaires à la présentation et au fonctionnement corrects du client.

Le HTML seul n'offre pas une grande polyvalence pour construire des interfaces plus élaborées et ne dispose pas de fonctions de programmation à usage général. Pour cette raison, un document HTML qui doit fonctionner comme une application client est toujours accompagné d'un ou plusieurs ensembles de CSS et de JavaScript.

Le CSS peut être fourni sous forme de fichier séparé ou directement dans le fichier HTML lui-même. L'objectif principal du CSS est d'ajuster l'apparence et la disposition des éléments de l'interface HTML. Bien que cela ne soit pas strictement nécessaire, les interfaces plus sophistiquées nécessitent généralement des modifications des propriétés CSS des éléments pour répondre à leurs besoins.

JavaScript est un composant pratiquement indispensable. Les procédures écrites en JavaScript répondent à des événements dans le navigateur. Ces événements peuvent être provoqués par l'utilisateur ou être non interactifs. Sans JavaScript, un document HTML est pratiquement limité au texte et aux images. L'utilisation de JavaScript dans les documents HTML permet d'étendre l'interactivité au-delà des hyperliens et des formulaires, faisant de la page affichée par le navigateur une interface d'application conventionnelle.

JavaScript est un langage de programmation polyvalent, mais il est principalement utilisé dans les applications web. Les fonctionnalités de l'environnement d'exécution du navigateur sont accessibles

par des mots-clés JavaScript, utilisés dans un script pour effectuer l'opération souhaitée. Le terme `document`, par exemple, est utilisé dans le code JavaScript pour désigner le document HTML associé au code JavaScript. Dans le contexte du langage JavaScript, le `document` est un *objet global* doté de propriétés et de méthodes qui peuvent être utilisées pour obtenir des informations de n'importe quel élément du document HTML. Plus important encore, vous pouvez utiliser l'objet `document` pour modifier ses éléments et les associer à des actions personnalisées écrites en JavaScript.

Une application client basée sur les technologies web est multiplateforme, car elle peut fonctionner sur tout appareil doté d'un navigateur web compatible.

Le fait d'être confiné au navigateur impose toutefois des limites aux applications web par rapport aux applications natives. L'intermédiation effectuée par le navigateur permet une programmation de plus haut niveau et accroît la sécurité, mais elle augmente également le traitement et la consommation de la mémoire.

Les développeurs travaillent continuellement sur les navigateurs pour offrir davantage de fonctionnalités et améliorer les performances des applications JavaScript, mais il existe des aspects intrinsèques à l'exécution de scripts tels que JavaScript qui leur imposent un désavantage par rapport aux programmes natifs pour le même matériel.

WebAssembly est une fonctionnalité qui améliore considérablement les performances des applications JavaScript exécutées dans le navigateur. WebAssembly est un type de JavaScript compilé qui produit un code source écrit dans un langage plus efficace et de plus bas niveau, tel que le langage C. WebAssembly peut accélérer principalement les activités gourmandes en ressources processeur, car il évite une grande partie de la traduction effectuée par le navigateur lors de l'exécution d'un programme écrit en JavaScript classique.

Quels que soient les détails de la mise en œuvre de l'application, tous les fichiers de code HTML, CSS, JavaScript et multimédia doivent d'abord être obtenus du serveur. Le navigateur obtient ces fichiers comme une page Internet, c'est-à-dire avec une adresse à laquelle le navigateur accède.

Une page web qui sert d'interface à une application web ressemble à un document HTML ordinaire, mais ajoute des comportements supplémentaires. Sur les pages classiques, l'utilisateur est dirigé vers une autre page après avoir cliqué sur un lien. Les applications web peuvent présenter leur interface et répondre aux événements de l'utilisateur sans charger de nouvelles pages dans la fenêtre du navigateur. La modification de ce comportement standard dans les pages HTML se fait via la programmation JavaScript.

Un client de messagerie web, par exemple, affiche les messages et passe d'un dossier de messages à un autre sans quitter la page. Cela est possible parce que le client utilise JavaScript pour réagir aux actions de l'utilisateur et faire les requêtes appropriées au serveur. Si l'utilisateur clique sur le sujet

d'un message dans la boîte de réception, un code JavaScript associé à cet événement demande le contenu de ce message au serveur (en utilisant l'appel API correspondant). Dès que le client reçoit la réponse, le navigateur affiche le message dans la partie appropriée de la même page. Les différents clients webmail peuvent adopter des stratégies différentes, mais ils utilisent tous ce même principe.

Par conséquent, en plus de fournir au navigateur les fichiers qui composent le client, le serveur doit également être en mesure de traiter des requêtes telles que celle du client webmail, lorsqu'il demande le contenu d'un message spécifique. Chaque requête que le client peut faire a une procédure prédéfinie pour répondre sur le serveur, dont l'API peut définir différentes méthodes pour identifier à quelle procédure la requête se réfère. Les méthodes les plus courantes sont les suivantes :

- Les adresses, par le biais d'un localisateur uniforme de ressource (URL : *Uniform Resource Locator*)
- Les champs de l'en-tête HTTP
- Les méthodes GET/POST
- Les WebSockets

Une méthode peut être plus adaptée qu'une autre, en fonction de l'objet de la requête et d'autres critères pris en compte par le développeur. En général, les applications web utilisent une combinaison de méthodes, chacune dans une circonstance spécifique.

Le modèle REST (*Representational State Transfer*) est largement utilisé pour la communication dans les applications web, car il repose sur les méthodes de base disponibles dans HTTP. L'en-tête d'une requête HTTP commence par un mot clé qui définit l'opération de base à effectuer : GET, POST, PUT, DELETE, etc., accompagné d'une URL correspondante où l'action sera appliquée. Si l'application nécessite des opérations plus spécifiques, avec une description plus détaillée de l'opération demandée, le protocole GraphQL peut être un choix plus approprié.

Les applications développées à l'aide du modèle client/serveur sont sujettes à des instabilités dans la communication. Pour cette raison, l'application cliente doit toujours adopter des stratégies de transfert de données efficaces pour favoriser sa cohérence et ne pas nuire à l'expérience utilisateur.

Côté serveur

Bien qu'il soit l'acteur principal d'une application web, le serveur est le côté passif de la communication, se contentant de répondre aux requêtes formulées par le client. Dans le jargon du web, le *serveur* peut désigner la machine qui reçoit les requêtes, le programme qui traite spécifiquement les requêtes HTTP ou le script destinataire qui produit une réponse à la requête. Cette dernière définition est la plus pertinente dans le contexte de l'architecture des applications web, mais elles sont toutes étroitement liées. Bien qu'ils ne soient que partiellement dans le champ

d'action du développeur de serveur d'applications, la machine, le système d'exploitation et le serveur HTTP ne peuvent être ignorés, car ils sont fondamentaux pour le fonctionnement du serveur d'applications et se croisent souvent.

Traitement des chemins à partir de requêtes

Les serveurs HTTP, tels que Apache et NGINX, nécessitent généralement des modifications de configuration spécifiques pour répondre aux besoins de l'application. Par défaut, les serveurs HTTP traditionnels associent directement le chemin indiqué dans la requête à un fichier du système de fichiers local. Si le serveur HTTP d'un site web conserve ses fichiers HTML dans le répertoire `/srv/www`, par exemple, une requête avec le chemin `/en/about.html` recevra le contenu du fichier `/srv/www/en/about.html` comme réponse, si le fichier existe. Les sites web plus sophistiqués, et surtout les applications web, exigent des traitements personnalisés pour différents types de requêtes. Dans ce scénario, une partie de la mise en œuvre de l'application consiste à modifier les paramètres du serveur HTTP pour répondre aux exigences de l'application.

Il existe également des *frameworks* qui permettent d'intégrer la gestion des requêtes HTTP et l'implémentation du code de l'application en un seul endroit, ce qui permet au développeur de se concentrer davantage sur l'objectif de l'application que sur les détails de la plate-forme. Dans Node.js Express, par exemple, tout le mappage des requêtes et la programmation correspondante sont mis en œuvre en utilisant JavaScript. Comme la programmation des clients est généralement effectuée en JavaScript, de nombreux développeurs considèrent que c'est une bonne idée, du point de vue de la maintenance du code, d'utiliser le même langage pour le client et pour le serveur. Les autres langages couramment utilisés pour mettre en œuvre le côté serveur, que ce soit dans les frameworks ou dans les serveurs HTTP traditionnels, sont PHP, Python, Ruby, Java et C#.

Systèmes de gestion des bases de données

La manière dont les données reçues ou demandées par le client sont stockées sur le serveur est laissée à la discrétion de l'équipe de développement, mais il existe des directives générales qui s'appliquent à la plupart des cas. Il est pratique de conserver le contenu statique : images, code JavaScript et CSS qui ne changent pas à court terme sous forme de fichiers classiques, soit sur le propre système de fichiers du serveur, soit distribué sur un *réseau de diffusion de contenu* (CDN : *Content Delivery Network*). D'autres types de contenu, comme les messages électroniques d'une application de messagerie web, les détails d'un produit dans une application d'achat et les journaux de transactions, sont plus facilement stockés dans un *système de gestion de base de données* (SGBD).

Le type le plus traditionnel de système de gestion de base de données est la *base de données relationnelle*. Dans celle-ci, le concepteur de l'application définit les tables de données et le format d'entrée accepté par chaque table. L'ensemble des tables de la base de données contient toutes les données dynamiques consommées et produites par l'application. Une application de shopping, par

exemple, peut avoir une table qui contient une entrée avec les détails de chaque produit dans le magasin et une table qui enregistre les articles achetés par un utilisateur. La table des articles achetés contient des références aux entrées de la table des produits, créant ainsi des relations entre les tables. Cette approche peut optimiser le stockage et l'accès aux données, ainsi que permettre des requêtes dans des tables combinées en utilisant le langage adopté par le système de gestion de base de données. Le langage de base de données relationnel le plus populaire est le *langage de requête structuré* (*Structured Query Language* : SQL, prononcé "sequel"), adopté par les bases de données open source SQLite, MySQL, MariaDB et PostgreSQL.

Une alternative aux bases de données relationnelles est une forme de base de données qui ne nécessite pas une structure rigide pour les données. Ces bases de données sont appelées *bases de données non relationnelles* ou simplement *NoSQL*. Bien qu'elles puissent intégrer des fonctionnalités similaires à celles des bases de données relationnelles, l'objectif est de permettre une plus grande flexibilité dans le stockage et l'accès aux données stockées, en confiant la tâche de traiter ces données à l'application elle-même. MongoDB, CouchDB et Redis sont des systèmes de gestion de bases de données non relationnelles courants.

Maintenance du contenu

Quel que soit le modèle de base de données adopté, les applications doivent ajouter des données et probablement les mettre à jour au cours de leur durée de vie. Dans certaines applications, comme le webmail, les utilisateurs fournissent eux-mêmes des données à la base de données lorsqu'ils utilisent le client pour envoyer et recevoir des messages. Dans d'autres cas, comme dans l'application de shopping, il est important de permettre aux responsables de l'application de modifier la base de données sans avoir à recourir à la programmation. De nombreuses organisations adoptent donc une sorte de *système de gestion de contenu* (CMS : *Content Management System*), qui permet aux utilisateurs non techniques d'administrer l'application. Par conséquent, pour la plupart des applications web, il est nécessaire de mettre en œuvre au moins deux types de clients : un client non privilégié, utilisé par les utilisateurs ordinaires, et des clients privilégiés, utilisés par des utilisateurs spéciaux pour maintenir et mettre à jour les informations présentées par l'application.

Exercices guidés

1. Quel langage de programmation est utilisé avec le HTML pour créer des applications web clientes ?

2. En quoi l'accès à une application web diffère-t-il de celui d'une application native ?

3. En quoi une application web diffère-t-elle d'une application native en matière d'accès au matériel local ?

4. Citez une caractéristique du client d'une application web qui le distingue d'une page web ordinaire.

Exercices d'exploration

1. Quelle fonction les navigateurs modernes offrent-ils pour pallier les mauvaises performances des clients d'applications web à forte utilisation du processeur ?

2. Si une application web utilise le modèle REST pour la communication client/serveur, quelle méthode HTTP doit être utilisée lorsque le client demande au serveur d'effacer une ressource spécifique ?

3. Citer cinq langages de script pris en charge par le serveur HTTP Apache.

4. Pourquoi les bases de données non relationnelles sont-elles considérées comme plus faciles à maintenir et à mettre à niveau que les bases de données relationnelles ?

Résumé

Cette leçon couvre les concepts et les normes de la technologie et de l'architecture du développement web. Le principe est simple : le navigateur web exécute l'application cliente, qui communique avec l'application centrale exécutée dans le serveur. Bien que simple dans son principe, les applications web doivent combiner de nombreuses technologies pour adopter le principe de l'informatique client et serveur sur le Web. La leçon passe en revue les concepts suivants :

- Rôle des navigateurs et des serveurs web.
- Technologies et normes de développement web courantes.
- Comment les clients web peuvent communiquer avec le serveur.
- Types de serveurs web et de serveurs de bases de données.

Réponses aux exercices guidés

1. Quel langage de programmation est utilisé avec le HTML pour créer des applications web clientes ?

JavaScript

2. En quoi l'accès à une application web diffère-t-il de celui d'une application native ?

Une application web n'est pas installée. Au lieu de cela, certaines de ses parties s'exécutent sur le serveur et l'interface client s'exécute dans un navigateur web ordinaire.

3. En quoi une application web diffère-t-elle d'une application native en matière d'accès au matériel local ?

Tous les accès aux ressources locales, comme le stockage, les caméras ou les microphones, sont gérés par le navigateur et nécessitent une autorisation explicite de l'utilisateur pour fonctionner.

4. Citez une caractéristique du client d'une application web qui le distingue d'une page web ordinaire.

L'interaction avec les pages web traditionnelles se limite essentiellement aux hyperliens et à l'envoi de formulaires, tandis que les clients des applications web sont plus proches d'une interface d'application conventionnelle.

Réponses aux exercices d'exploration

1. Quelle fonction les navigateurs modernes offrent-ils pour pallier les mauvaises performances des clients d'applications web à forte utilisation du processeur ?

Les développeurs peuvent utiliser WebAssembly pour mettre en œuvre les parties de l'application client qui requièrent une forte utilisation du processeur. Le code WebAssembly est généralement plus performant que le JavaScript traditionnel, car il nécessite moins de traduction d'instructions.

2. Si une application web utilise le modèle REST pour la communication client/serveur, quelle méthode HTTP doit être utilisée lorsque le client demande au serveur d'effacer une ressource spécifique ?

REST s'appuie sur des méthodes standards HTTP, il devrait donc utiliser la méthode standard DELETE dans ce cas.

3. Citer cinq langages de script pris en charge par le serveur HTTP Apache.

PHP, Go, Perl, Python, et Ruby.

4. Pourquoi les bases de données non relationnelles sont-elles considérées comme plus faciles à maintenir et à mettre à niveau que les bases de données relationnelles ?

Contrairement aux bases de données relationnelles, les bases de données non relationnelles n'exigent pas que les données correspondent à des structures rigides prédéfinies, ce qui facilite la mise en œuvre des changements dans les structures de données sans affecter les données existantes.



031.3 Bases du HTTP

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 031.3

Valeur

3

Domaines de connaissance les plus importants

- Comprendre les méthodes HTTP GET et POST, les codes d'état, les en-têtes et les types de contenu.
- Comprendre la différence entre le contenu statique et le contenu dynamique
- Comprendre les URL HTTP
- Comprendre comment les URL HTTP sont mises en correspondance avec les chemins du système de fichiers.
- Télécharger des fichiers vers la racine du document d'un serveur web.
- Comprendre la mise en cache
- Comprendre les cookies
- Connaissance des sessions et du détournement de session
- Connaissance des serveurs HTTP les plus utilisés
- Connaissance de HTTPS et TLS
- Connaissance des sockets web
- Connaissance des hôtes virtuels
- Connaissance des serveurs HTTP courants

- Connaissance des exigences et des limites de la bande passante et de la latence du réseau

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- GET, POST
- 200, 301, 302, 401, 403, 404, 500
- Apache HTTP Server (httpd), NGINX



031.3 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	031 Développement de logiciels et technologies Web
Objectif :	031.3 Bases du HTTP
Leçon :	1 sur 1

Introduction

Le *protocole de transfert hypertexte* (HTTP : *HyperText Transfer Protocol*) définit la manière dont un client demande une ressource spécifique au serveur. Son principe de fonctionnement est assez simple : le client crée un message de requête identifiant la ressource dont il a besoin et transmet ce message au serveur via le réseau. À son tour, le serveur HTTP évalue où extraire la ressource demandée et renvoie un message de réponse au client. Le message de réponse contient des détails sur la ressource demandée, suivis de la ressource elle-même.

Plus précisément, HTTP est l'ensemble des règles qui définissent comment l'application client doit formater les messages de *requête* (*request*) qui seront envoyés au serveur. Le serveur suit ensuite les règles HTTP pour interpréter la requête et formater les messages de *réponse* (*reply*). Outre la requête ou le transfert du contenu demandé, les messages HTTP contiennent des informations supplémentaires sur le client et le serveur concernés, sur le contenu lui-même, voire sur son indisponibilité. Si une ressource ne peut être envoyée, un code dans la réponse explique la raison de cette indisponibilité et, si possible, indique où la ressource a été déplacée.

La partie du message qui définit les détails de la ressource et d'autres informations contextuelles est

appelée l'*en-tête* du message. La partie qui suit l'en-tête, qui contient le contenu de la ressource correspondante, est appelée *charge utile (payload)* du message. Les messages de requête et les messages de réponse peuvent tous deux avoir une charge utile, mais dans la plupart des cas, seul le message de réponse en a une.

Requête du client

La première étape d'un échange de données HTTP entre le client et le serveur est initiée par le client, lorsqu'il écrit un message de requête au serveur. Prenons l'exemple d'une tâche courante du navigateur : charger une page HTML à partir d'un serveur hébergeant un site web, tel que `https://learning.lpi.org/en/`. L'adresse web, ou URL, fournit plusieurs éléments d'information pertinents. Trois éléments d'information apparaissent dans cet exemple particulier :

- Le protocole : *HyperText Transfer Protocol Secure* (https), une version chiffrée du protocole HTTP.
- Le nom de réseau de l'hôte web (`learning.lpi.org`)
- L'emplacement de la ressource demandée sur le serveur (le répertoire `/en/` dans ce cas, la page d'accueil en anglais).

NOTE

Un *localisateur uniforme de ressource* (URL : *Uniform Resource Locator*) est une adresse qui pointe vers une ressource sur l'internet. Cette ressource est généralement un fichier qui peut être copié à partir d'un serveur distant, mais les URL peuvent également indiquer un contenu généré dynamiquement et des flux de données.

Comment le client traite une URL

Avant de contacter le serveur, le client doit convertir `learning.lpi.org` en son adresse IP correspondante. Le client utilise un autre service internet, le *système de noms de domaine* (DNS : *Domain Name System*), pour demander l'adresse IP d'un nom d'hôte à un ou plusieurs serveurs DNS prédéfinis (les serveurs DNS sont généralement définis automatiquement par le fournisseur d'accès internet, FAI).

Avec l'adresse IP du serveur, le client essaie de se connecter au port HTTP ou HTTPS. Les ports réseau sont des numéros d'identification définis par le *protocole de contrôle de transmissions* (TCP : *Transmission Control Protocol*) pour entrelacer et identifier des canaux de communication distincts dans une connexion client/serveur. Par défaut, les serveurs HTTP reçoivent des requêtes sur les ports TCP 80 (HTTP) et 443 (HTTPS).

NOTE

Il existe d'autres protocoles utilisés par les applications web pour mettre en œuvre la communication client/serveur. Pour les appels audio et vidéo, par exemple, il est

plus approprié d'utiliser les WebSockets, un protocole de niveau inférieur qui est plus efficace que le protocole HTTP pour transférer des flux de données dans les deux sens.

Le format du message de requête que le client envoie au serveur est le même en HTTP et en HTTPS. HTTPS est déjà plus largement utilisé que HTTP, car tous les échanges de données entre le client et le serveur sont chiffrés, ce qui est une caractéristique indispensable pour promouvoir la confidentialité et la sécurité sur les réseaux publics. La connexion chiffrée est établie entre le client et le serveur avant même l'échange de tout message HTTP, à l'aide du protocole de chiffrement *sécurité de la couche de transport* (TLS : *Transport Layer Security*). Ce faisant, toutes les communications HTTPS sont encapsulées par TLS. Une fois déchiffré, la requête ou la réponse transmise par HTTPS n'est pas différente d'une requête ou d'une réponse effectuée exclusivement par HTTP.

Le troisième élément de notre URL, `/en/`, sera interprété par le serveur comme l'emplacement ou le chemin de la ressource demandée. Si le chemin n'est pas fourni dans l'URL, l'emplacement par défaut `/` sera utilisé. L'implémentation la plus simple d'un serveur HTTP associe les chemins d'accès dans les URL aux fichiers du système de fichiers sur lequel le serveur est exécuté, mais ce n'est qu'une des nombreuses options disponibles sur les serveurs HTTP plus sophistiqués.

Message de requête

HTTP fonctionne par le biais d'une connexion déjà établie entre le client et le serveur, généralement implémentée en TCP et chiffrée avec TLS. En fait, dès qu'une connexion répondant aux exigences imposées par le serveur est prête, une requête HTTP tapée à la main en texte clair pourrait générer la réponse du serveur. En pratique, cependant, les programmeurs ont rarement besoin d'implémenter des routines pour composer des messages HTTP, car la plupart des langages de programmation fournissent des mécanismes qui automatisent la fabrication du message HTTP. Dans le cas de l'URL de notre exemple, `https://learning.lpi.org/en/`, le message de requête le plus simple possible aurait le contenu suivant :

```
GET /en/ HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: text/html
```

Le premier mot de la première ligne identifie la *méthode* HTTP. Il définit l'opération que le client veut effectuer sur le serveur. La méthode GET informe le serveur que le client demande la ressource qui suit : `/en/`. Le client et le serveur pouvant prendre en charge plusieurs versions du protocole HTTP, la version à adopter dans l'échange de données est également indiquée sur la première ligne : `HTTP/1.1`.

NOTE

La version la plus récente du protocole HTTP est HTTP/2. Entre autres différences, les messages écrits en HTTP/2 sont codés dans une structure binaire, alors que les messages écrits en HTTP/1.1 sont envoyés en texte clair. Ce changement permet d'optimiser les taux de transmission des données, mais le contenu des messages reste fondamentalement le même.

L'en-tête peut contenir d'autres lignes après la première pour contextualiser et aider à identifier la requête au serveur. Le champ d'en-tête `Host`, par exemple, peut sembler redondant, car l'hôte du serveur a évidemment été identifié par le client afin d'établir la connexion et il est raisonnable de supposer que le serveur connaît sa propre identité. Néanmoins, il est important d'informer l'hôte du nom d'hôte attendu dans l'en-tête de la requête, car il est courant d'utiliser le même serveur HTTP pour héberger plus d'un site Web. (Dans ce scénario, chaque hôte spécifique est appelé *hôte virtuel*). Par conséquent, le champ `Host` est utilisé par le serveur HTTP pour identifier celui auquel la requête fait référence.

Le champ d'en-tête `User-Agent` contient des détails sur le programme client qui effectue la requête. Ce champ peut être utilisé par le serveur pour adapter la réponse aux besoins d'un client spécifique, mais il est plus souvent utilisé pour produire des statistiques sur les clients qui utilisent le serveur.

Le champ `Accept` a une valeur plus immédiate, car il informe le serveur du format de la ressource demandée. Si le client est indifférent au format de la ressource, le champ `Accept` peut spécifier `*/*` comme format d'entrée.

De nombreux autres champs d'en-tête peuvent être utilisés dans un message HTTP, mais les champs présentés dans l'exemple sont suffisants pour demander une ressource au serveur.

En plus des champs de l'en-tête de la requête, le client peut inclure d'autres données complémentaires dans la requête HTTP qui sera envoyée au serveur. Si ces données consistent uniquement en de simples paramètres textuels, au format `nom=valeur`, ils peuvent être ajoutés au chemin de la méthode GET. Les paramètres sont incorporés dans le chemin après un point d'interrogation et sont séparés par des caractères d'esperluette (&) :

```
GET /cgi-bin/receive.cgi?name=LPI&email=info@lpi.org HTTP/1.1
```

Dans cet exemple, `/cgi-bin/receive.cgi` est le chemin d'accès au script sur le serveur qui traitera et utilisera éventuellement les paramètres `name` et `email`, obtenus à partir du chemin de la requête. La chaîne qui correspond aux champs, au format `name=LPI&email=info@lpi.org`, est appelée *chaîne de requête* et elle est fournie au script `receive.cgi` par le serveur HTTP qui reçoit la requête.

Lorsque les données sont constituées de plus de champs de texte court, il est plus approprié de les envoyer dans la charge utile (*payload*) du message. Dans ce cas, la méthode HTTP POST doit être

utilisée pour que le serveur reçoive et traite les données utiles du message, conformément aux spécifications indiquées dans l'en-tête de la requête. Lorsque la méthode POST est utilisée, l'en-tête de la requête doit indiquer la taille de la charge utile qui sera envoyée ensuite et le mode de formatage du corps du message :

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
Content-Length: 1503
Content-Type: multipart/form-data; boundary=-----
405f7edfd646a37d
```

Le champ `Content-Length` indique la taille en octets de la charge utile et le champ `Content-Type` indique son format. Le format `multipart/form-data` est le plus couramment utilisé dans les formulaires HTML traditionnels qui utilisent la méthode POST. Dans ce format, chaque champ inséré dans les données utiles de la requête est séparé par le code indiqué par le mot-clé `boundary`. La méthode POST ne doit être utilisée que lorsque cela est approprié, car elle utilise une quantité de données légèrement supérieure à celle d'une requête équivalente effectuée avec la méthode GET. Comme la méthode GET envoie les paramètres directement dans l'en-tête de message de la requête, l'échange total de données a une latence plus faible, car une étape de connexion supplémentaire pour transmettre le corps du message ne sera pas nécessaire.

En-tête de la réponse

Après avoir reçu l'en-tête du message de requête, le serveur HTTP renvoie un message de réponse au client. Une requête de fichier HTML comporte généralement un en-tête de réponse comme celui-ci :

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 18170
Content-Type: text/html
Date: Mon, 05 Apr 2021 13:44:25 GMT
Etag: "606adcd4-46fa"
Last-Modified: Mon, 05 Apr 2021 09:48:04 GMT
Server: nginx/1.17.10
```

La première ligne indique la version du protocole HTTP utilisée dans le message de réponse, qui doit correspondre à la version utilisée dans l'en-tête de la requête. Ensuite, toujours sur la première ligne, apparaît le code d'état de la réponse, qui indique comment le serveur a interprété et a généré la réponse à la requête.

Le code d'état est un nombre à trois chiffres, le chiffre le plus à gauche définit la classe de réponse. Il existe cinq classes de codes d'état, numérotées de 1 à 5, chacune indiquant un type d'action prise par le serveur :

1xx (Information)

La requête a été reçue, ce qui a permis de poursuivre le processus.

2xx (Succès)

La requête a été reçue, comprise et acceptée avec succès.

3xx (Redirection)

D'autres mesures doivent être prises afin de compléter la requête.

4xx (Erreur du client)

La requête contient une mauvaise syntaxe ou ne peut pas être satisfaite.

5xx (Erreur du serveur)

Le serveur n'a pas réussi à satisfaire une requête apparemment valide.

Les deuxième et troisième chiffres sont utilisés pour indiquer des détails supplémentaires. Le code 200, par exemple, indique que la requête a pu être satisfaite sans problème. Comme le montre l'exemple, une brève description textuelle suivant le code de réponse (OK) peut également être fournie. Certains codes spécifiques présentent un intérêt particulier pour garantir que le client HTTP peut accéder à la ressource dans des situations défavorables ou pour aider à identifier la raison de l'échec en cas de requête infructueuse :

301 Moved Permanently

La ressource cible s'est vue attribuer une nouvelle URL permanente, fournie par le champ d'entête `Location` de la réponse.

302 Found

La ressource cible réside temporairement sous une autre URL.

401 Unauthorized

La requête n'a pas été appliquée car il manque des informations d'authentification valides pour la ressource cible.

403 Forbidden

La réponse `Forbidden` indique que, bien que la requête soit valide, le serveur est configuré pour ne pas la fournir.

404 Not Found

Le serveur d'origine n'a pas trouvé de représentation actuelle pour la ressource cible ou n'est pas disposé à révéler qu'il en existe une.

500 Internal Server Error

Le serveur a rencontré une condition inattendue qui l'a empêché de répondre à la requête.

502 Bad Gateway

Le serveur, agissant comme une passerelle ou un proxy, a reçu une réponse invalide d'un serveur entrant auquel il a accédé en tentant de répondre à la requête.

Bien qu'ils indiquent qu'il n'a pas été possible de répondre à la requête, les codes d'état 4xx et 5xx indiquent au moins que le serveur HTTP fonctionne et il est capable de recevoir des requêtes. Les codes 4xx nécessitent une action de la part du client, car son URL ou ses informations d'identification sont erronées. En revanche, les codes 5xx indiquent que quelque chose ne va pas du côté du serveur. Par conséquent, dans le contexte des applications web, ces deux classes de codes d'état indiquent que la source de l'erreur se trouve dans l'application elle-même, soit le client ou le serveur, et non dans l'infrastructure sous-jacente.

Contenu statique et dynamique

Les serveurs HTTP utilisent deux mécanismes de base pour répondre au contenu demandé par le client. Le premier mécanisme fournit un *contenu statique* : c'est-à-dire que le chemin indiqué dans le message de requête correspond à un fichier sur le système de fichiers local du serveur. Le second mécanisme fournit un *contenu dynamique* : le serveur HTTP transmet la requête à un autre programme, probablement un script, pour construire la réponse à partir de différentes sources, telles que des bases de données et d'autres fichiers.

Bien qu'il existe différents serveurs HTTP, ils utilisent tous le même protocole de communication HTTP et adoptent plus ou moins les mêmes conventions. Une application qui n'a pas de besoin spécifique peut être mise en œuvre avec n'importe quel serveur traditionnel, comme Apache ou NGINX. Tous deux sont capables de générer du contenu dynamique et de fournir du contenu statique, mais il existe des différences subtiles dans la configuration de chacun.

L'emplacement des fichiers statiques à servir, par exemple, est défini de différentes manières dans Apache et NGINX. La convention est de conserver ces fichiers dans un répertoire spécifique à cet effet, ayant un nom associé à l'hôte, par exemple `/var/www/learning.lpi.org/`. Dans Apache, ce chemin est défini par la directive de configuration `DocumentRoot /var/www/learning.lpi.org`, dans une section qui définit un hôte virtuel. Dans NGINX, la directive utilisée est `root /var/www/learning.lpi.org` dans une section serveur du fichier de configuration.

Quel que soit le serveur que vous choisissiez, les fichiers de `/var/www/learning.lpi.org/` seront servis via HTTP de manière presque identique. Certains champs de l'en-tête de réponse et leur contenu peuvent varier entre les deux serveurs, mais des champs comme `Content-Type` doivent être présents dans l'en-tête de réponse et doivent être cohérents sur tous les serveurs.

Mise en cache

HTTP a été conçu pour fonctionner sur tout type de connexion Internet, rapide ou lente. En outre, la plupart des échanges HTTP doivent traverser de nombreux nœuds de réseau en raison de l'architecture distribuée d'internet. Par conséquent, il est important d'adopter une stratégie de mise en cache du contenu afin d'éviter le transfert redondant de contenu précédemment téléchargé. Les transferts HTTP peuvent fonctionner avec deux types de cache de base : *partagé* et *privé*.

Un cache partagé est utilisé par plus d'un seul client. Par exemple, un grand fournisseur de contenu peut utiliser des caches sur des serveurs géographiquement répartis, de sorte que les clients obtiennent les données du serveur le plus proche. Une fois qu'un client a fait une requête et que sa réponse a été stockée dans un cache partagé, les autres clients qui font la même requête dans la même zone recevront la réponse en cache.

Un cache privé est créé par le client lui-même pour son usage exclusif. Il s'agit du type de cache que le navigateur web utilise pour les images, les fichiers CSS, le JavaScript ou le document HTML lui-même, afin qu'ils ne soient pas téléchargés à nouveau s'ils sont demandés dans un avenir proche.

NOTE

Toutes les requêtes HTTP ne doivent pas être mises en cache. Une requête utilisant la méthode POST, par exemple, implique une réponse associée exclusivement à cette requête particulière, son contenu ne doit donc pas être réutilisé. Par défaut, seules les réponses aux requêtes effectuées à l'aide de la méthode GET sont mises en cache. En outre, seules les réponses comportant des codes d'état concluants tels que 200 (OK), 206 (contenu partiel), 301 (déplacé de façon permanente) et 404 (non trouvé) peuvent être mises en cache.

Les stratégies de cache partagé et privé utilisent toutes les deux des en-têtes HTTP pour contrôler la manière dont le contenu téléchargé doit être mis en cache. Pour le cache privé, le client consulte l'en-tête de réponse et vérifie si le contenu du cache local correspond toujours au contenu distant actuel. Si c'est le cas, le client renonce au transfert des données utiles de la réponse et utilise la version locale.

La validité de la ressource mise en cache peut être évaluée de plusieurs façons. Le serveur peut fournir une date d'expiration dans l'en-tête de réponse de la première requête, de sorte que le client se débarrasse de la ressource mise en cache à la fin de la période et la demande à nouveau pour obtenir la version mise à jour. Cependant, le serveur n'est pas toujours en mesure de déterminer la

date d'expiration d'une ressource, il est donc courant d'utiliser le champ d'en-tête de réponse ETag pour identifier la version de la ressource, par exemple Etag : "606adcd4-46fa".

Pour vérifier qu'une ressource mise en cache doit être mise à jour, le client demande uniquement son en-tête de réponse au serveur. Si le champ ETag correspond à celui de la version stockée localement, le client réutilise le contenu mis en cache. Sinon, le contenu mis à jour de la ressource est téléchargé depuis le serveur.

Sessions HTTP

Dans un site web ou une application web classique, les fonctions qui gèrent le contrôle des sessions sont basées sur les en-têtes HTTP. Le serveur ne peut pas supposer, par exemple, que toutes les requêtes provenant de la même adresse IP proviennent du même client. La méthode la plus traditionnelle qui permet au serveur d'associer différentes requêtes à un même client est l'utilisation de *cookies*, une étiquette d'identification qui est donnée au client par le serveur et qui est fournie dans l'en-tête HTTP.

Les cookies permettent au serveur de conserver des informations sur un client spécifique, même si la personne qui exécute le client ne s'identifie pas explicitement. Grâce aux cookies, il est possible de mettre en place des sessions où les identifiants, les paniers d'achat, les préférences, etc., sont conservés entre différentes requêtes adressées au même serveur qui les a fournis. Les cookies sont également utilisés pour suivre la navigation des utilisateurs, il est donc important de demander leur consentement avant de les envoyer.

Le serveur définit le cookie dans l'en-tête de réponse à l'aide du champ Set-Cookie. La valeur du champ est une paire nom=valeur choisie pour représenter un attribut associé à un client spécifique. Le serveur peut, par exemple, créer un numéro d'identification pour un client qui demande une ressource pour la première fois et le transmettre au client dans l'en-tête de réponse :

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Set-Cookie: client_id=62b5b719-fcbf
```

Si le client autorise l'utilisation des cookies, les nouvelles requêtes adressées à ce même serveur comportent le champ cookie dans l'en-tête :

```
GET /en/ HTTP/1.1
Host: learning.lpi.org
Cookie: client_id=62b5b719-fcbf
```

Grâce à ce numéro d'identification, le serveur peut récupérer des définitions spécifiques pour le client et générer une réponse personnalisée. Il est également possible d'utiliser plusieurs champs `Set-Cookie` pour délivrer différents cookies au même client. De cette façon, plusieurs définitions peuvent être conservées du côté du client.

Les cookies soulèvent à la fois des problèmes de confidentialité et des failles de sécurité potentielles, car il est possible qu'ils soient transférés à un autre client, qui sera identifié par le serveur comme le client d'origine. Les cookies utilisés pour préserver les sessions peuvent donner accès à des informations sensibles du client d'origine. Il est donc très important que les clients adoptent des mécanismes de protection locaux pour empêcher que leurs cookies soient extraits et réutilisés sans autorisation.

Exercices guidés

1. Quelle méthode HTTP le message de requête suivant utilise-t-il ?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

2. Lorsqu'un serveur HTTP héberge de nombreux sites Web, comment peut-il identifier celui auquel s'adresse une requête ?

3. Quel paramètre est fourni par la chaîne de requête de l'URL `https://www.google.com/search?q=LPI` ?

4. Pourquoi la requête HTTP suivante ne convient-elle pas à la mise en cache ?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

Exercices d'exploration

1. Comment pouvez-vous utiliser le navigateur web pour surveiller les requêtes et les réponses faites par une page HTML ?

2. Les serveurs HTTP qui fournissent du contenu statique font généralement correspondre le chemin d'accès demandé à un fichier dans le système de fichiers du serveur. Que se passe-t-il lorsque le chemin d'accès de la requête pointe vers un répertoire ?

3. Le contenu des fichiers envoyés par HTTPS est protégé par chiffrement, de sorte qu'il ne peut pas être lu par les ordinateurs situés entre le client et le serveur. Malgré cela, ces ordinateurs intermédiaires peuvent-ils identifier la ressource que le client a demandée au serveur ?

Résumé

Cette leçon couvre les bases du protocole HTTP, le principal protocole utilisé par les applications clientes pour demander des ressources aux serveurs web. La leçon aborde les concepts suivants :

- Messages de requêtes, champs d'en-tête et méthodes.
- Codes d'état de la réponse.
- Comment les serveurs HTTP génèrent des réponses.
- Fonctions HTTP utiles pour la mise en cache et la gestion des sessions.

Réponses aux exercices guidés

1. Quelle méthode HTTP le message de requête suivant utilise-t-il ?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

La méthode `POST`.

2. Lorsqu'un serveur HTTP héberge de nombreux sites Web, comment peut-il identifier celui auquel s'adresse une requête ?

Le champ `Host` de l'en-tête de la requête fournit le site web ciblé.

3. Quel paramètre est fourni par la chaîne de requête de l'URL `https://www.google.com/search?q=LPI` ?

Le paramètre nommé `q` avec la valeur `LPI`.

4. Pourquoi la requête HTTP suivante ne convient-elle pas à la mise en cache ?

```
POST /cgi-bin/receive.cgi HTTP/1.1
Host: learning.lpi.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101
Firefox/87.0
Accept: */*
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
```

Comme les requêtes effectuées avec la méthode `POST` impliquent une opération d'écriture sur le serveur, elles ne doivent pas être mises en cache.

Réponses aux exercices d'exploration

1. Comment pouvez-vous utiliser le navigateur web pour surveiller les requêtes et les réponses faites par une page HTML ?

Tous les navigateurs populaires offrent des *outils de développement* qui, entre autres, peuvent montrer toutes les transactions réseau qui ont été effectuées par la page actuelle.

2. Les serveurs HTTP qui fournissent du contenu statique font généralement correspondre le chemin d'accès demandé à un fichier dans le système de fichiers du serveur. Que se passe-t-il lorsque le chemin d'accès de la requête pointe vers un répertoire ?

Cela dépend de la façon dont le serveur est configuré. Par défaut, la plupart des serveurs HTTP recherchent un fichier nommé `index.html` (ou un autre nom prédéfini) dans ce même répertoire et l'envoient comme réponse. Si le fichier n'est pas là, le serveur envoie une réponse 404 Not Found.

3. Le contenu des fichiers envoyés par HTTPS est protégé par chiffrement, de sorte qu'il ne peut pas être lu par les ordinateurs situés entre le client et le serveur. Malgré cela, ces ordinateurs intermédiaires peuvent-ils identifier la ressource que le client a demandée au serveur ?

Non, car les en-têtes HTTP de la requête et de la réponse sont eux aussi chiffrés par TLS.



**Linux
Professional
Institute**

Thème 032: Balisage de documents HTML



032.1 Anatomie d'un document HTML

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 032.1

Valeur

2

Domaines de connaissance les plus importants

- Créer un document HTML simple
- Comprendre le rôle du HTML
- Comprendre le squelette du HTML
- Comprendre la syntaxe HTML (balises, attributs, commentaires)
- Comprendre l'en-tête HTML
- Comprendre les balises meta
- Comprendre le codage des caractères

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `<!DOCTYPE html>`
- `<html>`
- `<head>`
- `<body>`
- `<meta>`, y compris le codage des caractères (UTF-8), les attributs `name` et `content`



032.1 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	032 Balisage de documents HTML
Objectif :	032.1 Anatomie d'un document HTML
Leçon :	1 sur 1

Introduction

Le HTML (*HyperText Markup Language*) est un langage de balisage qui indique aux navigateurs web comment structurer et afficher les pages web. La version actuelle est la 5.0, qui a été publiée en 2012. La syntaxe HTML est définie par le *World Wide Web Consortium* (W3C).

Le HTML est une compétence fondamentale dans le développement web, car il définit la structure et une grande partie de l'apparence d'un site web. Si vous souhaitez faire carrière dans le développement web, le HTML est certainement un bon point de départ.

Anatomie d'un document HTML

Une page HTML basique a la structure suivante :

```
<!DOCTYPE html>
<html>
  <head>
    <title>My HTML Page</title>
    <!-- This is the Document Header -->
  </head>
```

```
<body>
  <!-- This is the Document Body -->
</body>
</html>
```

Maintenant, analysons cette structure en détail.

Balises HTML

Le langage HTML utilise des *éléments* et des *balises* pour décrire et formater le contenu. Les balises sont des noms placés entre des crochets, par exemple `<title>`. Le nom de la balise n'est pas sensible à la casse, bien que le World Wide Web Consortium (W3C) recommande d'utiliser des lettres minuscules dans les versions actuelles du HTML. Ces balises HTML sont utilisées pour construire des éléments HTML. La balise `<title>` est un exemple de *balise ouvrante* d'un élément HTML qui définit le titre d'un document HTML. Cependant, un élément possède deux autres composants. Un élément `<title>` complet ressemble à ceci :

```
<title>My HTML Page</title>
```

Ici, `My HTML Page` sert de *contenu* à l'élément, tandis que `</title>` sert de *balise fermante* qui déclare que cet élément est complet.

NOTE

Tous les éléments HTML n'ont pas besoin d'être fermés ; dans ce cas, on parle d'éléments vides ou d'éléments auto-fermants.

Voici les autres éléments HTML de l'exemple précédent :

`<html>`

Entoure l'ensemble du document HTML. Il contient toutes les balises qui composent la page. Il indique également que le contenu de ce fichier est en langage HTML. La balise fermante correspondante est `</html>`.

`<head>`

Un conteneur pour toutes les informations méta concernant la page. La balise fermante correspondante de cet élément est `</head>`.

`<body>`

Un conteneur pour le contenu de la page et sa représentation structurelle. La balise fermante correspondante est `</body>`.

Les balises `<html>`, `<head>`, `<body>` et `<title>` sont appelées *balises squelettes*, qui fournissent la structure de base d'un document HTML. En particulier, elles indiquent au navigateur Web qu'il est en train de lire une page HTML.

NOTE

Parmi ces éléments HTML, le seul qui est obligatoire pour qu'un document HTML soit validé est la balise `<title>`.

Comme vous pouvez le constater, chaque page HTML est un document bien structuré et pourrait même être représenté comme un arbre, où l'élément `<html>` représente la racine du document et les éléments `<head>` et `<body>` sont les premières branches. L'exemple montre qu'il est possible d'imbriquer des éléments : Par exemple, l'élément `<title>` est imbriqué dans l'élément `<head>`, qui est à son tour imbriqué dans l'élément `<html>`.

Pour que votre code HTML soit lisible et facile à maintenir, assurez-vous que tous les éléments HTML sont fermés correctement et dans l'ordre. Les navigateurs Web peuvent toujours rendre votre site Web comme prévu, mais l'imbrication incorrecte des éléments et de leurs balises est une pratique sujette aux erreurs.

Enfin, une mention spéciale va à la déclaration *doctype* tout en haut de la structure du document d'exemple. `<!DOCTYPE>` n'est pas une balise HTML, mais une instruction pour le navigateur web qui spécifie la version HTML utilisée dans le document. Dans la structure du document HTML de base présentée précédemment, `<!DOCTYPE html>` a été utilisé, spécifiant que le HTML5 est utilisé dans ce document.

Commentaires en HTML

Lors de la création d'une page HTML, il est bon d'insérer des commentaires dans le code afin d'en améliorer la lisibilité et de décrire l'objectif des plus grands blocs de code. Un commentaire est inséré entre les balises `<!--` et `-->`, comme le montre l'exemple suivant :

```
<!-- This is a comment. -->

<!--
  This is a
  multiline
  comment.
-->
```

L'exemple montre que les commentaires HTML peuvent être placés sur une seule ligne, mais peuvent aussi s'étendre sur plusieurs lignes. Dans tous les cas, le résultat est que le texte entre `<!--` et `-->` est ignoré par le navigateur Web et n'est donc pas affiché dans la page HTML. Sur la base de

ces considérations, vous pouvez déduire que la page HTML basique présentée dans la section précédente n'affiche aucun texte, car les lignes `<!-- This is the Document Header -->` et `<!-- This is the Document Body -->` ne sont que deux commentaires.

WARNING | Les commentaires ne peuvent pas être imbriqués.

Attributs HTML

Les balises HTML peuvent inclure un ou plusieurs *attributs* pour préciser les détails de l'élément HTML. Une balise simple avec deux attributs a la forme suivante :

```
<tag attribute-a="value-a" attribute-b="value-b">
```

Les attributs doivent toujours être définis dans la balise ouvrante.

Un attribut se compose d'un nom, qui indique la propriété à définir, d'un signe égal et de la valeur souhaitée entre guillemets. Les guillemets simples et les guillemets doubles sont tous deux acceptables, mais il est recommandé d'utiliser les guillemets simples ou les guillemets doubles de manière cohérente tout au long d'un projet. Il est important de ne pas mélanger les guillemets simples et doubles pour une même valeur d'attribut, car le navigateur Web ne reconnaîtra pas les guillemets mixtes comme une seule unité.

NOTE

Vous pouvez inclure un type de guillemets dans un autre type sans aucun problème. Par exemple, si vous devez utiliser `'` dans une valeur d'attribut, vous pouvez inclure cette valeur dans `"`. Toutefois, si vous souhaitez utiliser le même type de guillemets à l'intérieur de la valeur que celui que vous utilisez pour envelopper la valeur, vous devez utiliser `"` ; pour `"` et `'` ; pour `'`.

Les attributs peuvent être classés en *attributs de base* et en *attributs spécifiques*, comme expliqué dans les sections suivantes.

Attributs de base

Les attributs de base sont des attributs qui peuvent être utilisés sur n'importe quel élément HTML. Ils comprennent :

title

Décrit le contenu de l'élément. Sa valeur est souvent affichée sous la forme d'une infobulle qui s'affiche lorsque l'utilisateur déplace son curseur sur l'élément.

id

Associe un identifiant unique à un élément. Cet identifiant doit être unique dans le document, et le document ne sera pas validé si plusieurs éléments partagent le même `id`.

style

Affecte des propriétés graphiques (styles CSS) à l'élément.

class

Spécifie une ou plusieurs classes pour l'élément dans une liste de noms de classe séparés par des espaces. Ces classes peuvent être référencées dans les feuilles de style CSS.

lang

Spécifie la langue du contenu de l'élément en utilisant les codes de langue à deux caractères de la norme ISO-639.

NOTE

Le développeur peut stocker des informations personnalisées sur un élément en définissant ce que l'on appelle un attribut `data-`, qui est indiqué en faisant précéder le nom souhaité par `data-` comme dans `data-infoadditionnelle`. Vous pouvez attribuer une valeur à cet attribut comme à tout autre attribut.

Attributs spécifiques

D'autres attributs sont spécifiques à chaque élément HTML. Par exemple, l'attribut `src` d'un élément HTML `` spécifie l'URL d'une image. Il existe de nombreux autres attributs spécifiques, qui seront abordés dans les leçons suivantes.

En-tête du document

L'en-tête du document définit les méta-informations concernant la page et il est décrit par l'élément `<head>`. Par défaut, les informations contenues dans l'en-tête du document ne sont pas rendues par le navigateur Web. Bien qu'il soit possible d'utiliser l'élément `<head>` pour contenir des éléments HTML qui pourraient être affichés sur la page, il n'est pas recommandé de le faire.

Titre

Le titre du document est spécifié à l'aide de l'élément `<title>`. Le titre défini entre les balises apparaît dans la barre de titre du navigateur web et constitue le nom suggéré pour le signet lorsque vous essayez de mettre la page en signet. Il est également affiché dans les résultats des moteurs de recherche comme titre de la page.

Voici un exemple de cet élément :

```
<title>My test page</title>
```

La balise `<title>` est obligatoire dans tous les documents HTML et ne doit apparaître qu'une seule fois dans chaque document.

NOTE

Ne pas confondre le titre du document avec le titre de la page, qui est placé dans le corps du document.

Métadonnées

L'élément `<meta>` est utilisé pour spécifier des méta informations afin de décrire plus précisément le contenu d'un document HTML. C'est un élément dit auto-fermant, ce qui signifie qu'il n'a pas de balise fermante. Outre les attributs de base qui sont valables pour chaque élément HTML, l'élément `<meta>` utilise également les attributs suivants :

name

Définit les métadonnées qui seront décrites dans cet élément. Il peut être défini par n'importe quelle valeur personnalisée, mais les valeurs couramment utilisées sont `author` (auteur), `description` (description) et `keywords` (mots-clés).

http-equiv

Fournit un en-tête HTTP pour la valeur de l'attribut `content`. Une valeur courante est `refresh`, qui sera expliquée plus tard. Si cet attribut est défini, l'attribut `name` ne doit pas être défini.

content

Fournit la valeur associée à l'attribut `name` ou `http-equiv`.

charset

Spécifie le codage des caractères pour le document HTML, par exemple `utf-8` pour définir le codage à *Unicode Transformation Format — 8 bits*.

Ajouter un auteur, une description et des mot-clés

En utilisant la balise `<meta>`, vous pouvez spécifier des informations supplémentaires sur l'auteur de la page HTML et décrire le contenu de la page comme ceci :

```
<meta name="author" content="Name Surname">
<meta name="description" content="A short summary of the page content.">
```


Essayez d'inclure une série de mots-clés liés au contenu de la page dans la description. Cette description est souvent la première chose qu'un utilisateur voit lorsqu'il navigue avec un moteur de recherche.

Si vous souhaitez également fournir aux moteurs de recherche des mots-clés supplémentaires liés à la page Web, vous pouvez ajouter cet élément :

```
<meta name="keywords" content="keyword1, keyword2, keyword3, keyword4, keyword5">
```

NOTE

Dans le passé, les spammeurs saisissaient des centaines de mots-clés et de descriptions sans rapport avec le contenu réel de la page, de sorte que celle-ci apparaissait également dans des recherches sans rapport avec les termes recherchés par les internautes. Aujourd'hui, les balises `<meta>` sont reléguées à une position d'importance secondaire et ne servent qu'à consolider les sujets traités dans la page web, de sorte qu'il n'est plus possible de tromper les nouveaux algorithmes des moteurs de recherche, plus sophistiqués.

Rediriger une page HTML et définir un intervalle pour l'autorafrachissement du document

En utilisant la balise `<meta>`, vous pouvez rafraîchir automatiquement une page HTML après une certaine période (par exemple après 30 secondes) de cette manière :

```
<meta http-equiv="refresh" content="30">
```

Vous pouvez également rediriger une page web vers une autre page web après le même laps de temps avec le code suivant :

```
<meta http-equiv="refresh" content="30; url=http://www.lpi.org">
```

Dans cet exemple, l'utilisateur est redirigé de la page actuelle vers `http://www.lpi.org` après 30 secondes. Les valeurs peuvent être ce que vous voulez. Par exemple, si vous spécifiez `content="0 ; url=http://www.lpi.org"`, la page est redirigée immédiatement.

Spécifier le codage des caractères

L'attribut `charset` spécifie le codage des caractères pour le document HTML. Un exemple courant est :

```
<meta charset="utf-8">
```

Cet élément spécifie que le codage des caractères du document est utf-8, qui est un jeu de caractères universel comprenant pratiquement tous les caractères de toutes les langues humaines. Par conséquent, en l'utilisant, vous éviterez les problèmes d'affichage de certains caractères que vous pourriez rencontrer en utilisant d'autres jeux de caractères tels que ISO-8859-1 (l'alphabet latin).

Autres exemples utiles

Deux autres applications utiles de la balise `<meta>` sont :

- Définition des cookies pour garder la trace d'un visiteur du site.
- Prise du contrôle de la fenêtre d'affichage (la zone visible d'une page web à l'intérieur d'une fenêtre de navigateur web), qui dépend de la taille de l'écran de l'appareil de l'utilisateur (par exemple, un téléphone mobile ou un ordinateur).

Cependant, ces deux exemples dépassent le cadre de l'examen et leur étude est laissée au lecteur curieux qui pourra l'explorer ailleurs.

Exercices guidés

1. Pour chacune des balises suivantes, indiquez la balise fermante correspondante :

<code><body></code>	
<code><head></code>	
<code><html></code>	
<code><meta></code>	
<code><title></code>	

2. Quelle est la différence entre une balise et un élément ? Utilisez ce bout de code comme référence :

```
<title>HTML Page Title</title>
```

3. Quelles sont les balises entre lesquelles un commentaire doit être placé ?

4. Expliquez ce qu'est un attribut et donnez quelques exemples pour la balise `<meta>`.

Exercices d'exploration

1. Créez un document HTML version 5 simple avec le titre `My first HTML document` et un seul paragraphe dans le corps, contenant le texte `Hello World`. Utilisez la balise de paragraphe `<p>` dans le corps.

2. Ajouter l'auteur (`Kevin Author`) et la description (`This is my first HTML page.`) au document HTML.

3. Ajoutez les mots-clés suivants relatifs au document HTML : `HTML`, `Example`, `Test`, et `Metadata`.

4. Ajoutez l'élément `<meta charset="ISO-8859-1">` à l'en-tête du document et changez le texte `Hello World` au japonais (`こんにちは`). Que se passe-t-il ? Comment pouvez-vous résoudre le problème ?

5. Après le changement du texte du paragraphe en revenant à `Hello World`, redirigez la page HTML vers `https://www.google.com` après 30 secondes et ajoutez un commentaire expliquant cela dans l'en-tête du document.

Résumé

Dans cette leçon, vous avez appris :

- Le rôle du HTML
- Le squelette HTML
- La syntaxe HTML (balises, attributs, commentaires)
- L'en-tête HTML
- Les balises meta
- Comment créer un document HTML simple

Les termes suivants ont été abordés dans cette leçon :

<!DOCTYPE html>

La balise de déclaration.

<html>

Le conteneur de toutes les balises qui composent la page HTML.

<head>

Le conteneur pour tous les éléments de l'en-tête.

<body>

Le conteneur pour tous les éléments du corps.

<meta>

La balise pour les métadonnées, utilisée pour spécifier des informations supplémentaires pour la page HTML (comme l'auteur, la description et le codage des caractères).

Réponses aux exercices guidés

1. Pour chacune des balises suivantes, indiquez la balise fermante correspondante :

<code><body></code>	<code></body></code>
<code><head></code>	<code></head></code>
<code><html></code>	<code></html></code>
<code><meta></code>	None
<code><title></code>	<code></title></code>

2. Quelle est la différence entre une balise et un élément ? Utilisez ce bout de code comme référence :

```
<title>HTML Page Title</title>
```

Un élément HTML se compose d'une balise de début, éventuellement d'une balise de fin et de tout ce qui se trouve entre les deux. Une balise HTML est utilisée pour marquer le début ou la fin d'un élément. Par conséquent, `<title>HTML Page Title</title>` est un élément HTML, tandis que `<title>` et `</title>` sont respectivement les balises ouvrante et fermante.

3. Quelles sont les balises entre lesquelles un commentaire doit être placé ?

Un commentaire est inséré entre les balises `<!--` et `-->` et peut être placé sur une seule ligne ou s'étendre sur plusieurs lignes.

4. Expliquez ce qu'est un attribut et donnez quelques exemples pour la balise `<meta>`.

Un attribut est utilisé pour spécifier plus précisément un élément HTML. Par exemple, la balise `<meta>` utilise la paire d'attributs `name` et `content` pour ajouter l'auteur et la description d'une page HTML. En outre, l'attribut `charset` permet de spécifier le codage des caractères pour le document HTML.

Réponses aux exercices d'exploration

1. Créez un document HTML version 5 simple avec le titre My first HTML document et un seul paragraphe dans le corps, contenant le texte Hello World. Utilisez la balise de paragraphe <p> dans le corps.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

2. Ajouter l'auteur (Kevin Author) et la description (This is my first HTML page.) au document HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```

3. Ajoutez les mots-clés suivants relatifs au document HTML : HTML, Example, Test, et Metadata.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
```

```
<meta name="keywords" content="HTML, Example, Test, Metadata">
</head>

<body>
  <p>Hello World</p>
</body>
</html>
```

4. Ajoutez l'élément `<meta charset="ISO-8859-1">` à l'en-tête du document et changez le texte Hello World au japonais (こんにちは). Que se passe-t-il ? Comment pouvez-vous résoudre le problème ?

Si l'exemple est exécuté comme décrit, le texte japonais ne s'affiche pas correctement. Cela est dû au fait que la norme ISO-8859-1 représente le codage des caractères pour l'alphabet latin. Pour afficher le texte, vous devez modifier le codage des caractères, en utilisant par exemple UTF-8 (`<meta charset="utf-8">`).

5. Après le changement du texte du paragraphe en revenant à Hello World, redirigez la page HTML vers `https://www.google.com` après 30 secondes et ajoutez un commentaire expliquant cela dans l'en-tête du document.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML document</title>
    <meta name="author" content="Kevin Author">
    <meta name="description" content="This is my first HTML page.">
    <meta name="keywords" content="HTML, Example, Test, Metadata">
    <meta charset="utf-8">
    <!-- The page is redirected to Google after 30 seconds -->
    <meta http-equiv="refresh" content="30; url=https://www.google.com">
  </head>

  <body>
    <p>Hello World</p>
  </body>
</html>
```




Linux
Professional
Institute

032.2 Sémantique HTML et hiérarchie des documents

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 032.2

Valeur

2

Domaines de connaissance les plus importants

- Créer des balises pour le contenu d'un document HTML
- Comprendre la structure hiérarchique du texte HTML
- Différencier les éléments HTML en bloc et en ligne
- Comprendre les éléments HTML structurels sémantiques importants

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`
- `<p>`
- ``, ``, ``
- `<dl>`, `<dt>`, `<dd>`
- `<pre>`
- `<blockquote>`
- ``, ``, `<code>`
- ``, `<i>`, `<u>`
- ``
- `<div>`

- `<main>`, `<header>`, `<nav>`, `<section>`, `<footer>`



032.2 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	032 Balisage de documents HTML
Objectif :	032.2 Sémantique HTML et hiérarchie des documents
Leçon :	1 sur 1

Introduction

Dans la leçon précédente, nous avons appris que le HTML est un langage de balisage qui permet de décrire sémantiquement le contenu d'un site web. Un document HTML contient ce que l'on appelle un squelette qui se compose des éléments HTML `<html>`, `<head>` et `<body>`. Alors que l'élément `<head>` décrit un bloc de méta-informations pour le document HTML qui sera invisible pour le visiteur du site web, l'élément `<body>` peut contenir de nombreux autres éléments pour définir la structure et le contenu du document HTML.

Dans cette leçon, nous allons aborder le formatage du texte, les éléments sémantiques fondamentaux du HTML et leur utilité, et la manière de structurer un document HTML. Nous utiliserons une liste de courses comme exemple.

NOTE

Tous les exemples de code suivants se trouvent dans l'élément `<body>` d'un document HTML contenant le squelette complet. Pour des raisons de lisibilité, nous ne montrerons pas le squelette HTML dans tous les exemples de cette leçon.

Texte

En HTML, aucun bloc de texte ne doit être nu, en dehors d'un élément. Même un court paragraphe doit être enveloppé dans les balises HTML `<p>`, qui est le nom raccourci pour *paragraphe*.

```
<p>Short text element spanning only one line.</p>  
<p>A text element containing much longer text that may span across multiple lines,  
depending on the size of the web browser window.</p>
```

Ouvert dans un navigateur web, ce code HTML produit le résultat indiqué dans la [Figure 1](#).

Short text element spanning only one line

A text element containing much longer text that may span across multiple lines depending on the size of the web browser window.

Figure 1. Représentation du code HTML dans le navigateur web, montrant deux paragraphes de texte. Le premier paragraphe est très court. Le deuxième paragraphe est un peu plus long et s'étend sur une deuxième ligne.

Par défaut, les navigateurs web ajoutent un espace avant et après les éléments `<p>` pour améliorer la lisibilité. Pour cette raison, `<p>` est appelé *élément de bloc*.

Titres

Le langage HTML définit six niveaux de titres (*headings*) pour décrire et structurer le contenu d'un document HTML. Ces titres sont marqués par les balises HTML `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` et `<h6>`.

```
<h1>Heading level 1 to uniquely identify the page</h1>  
<h2>Heading level 2</h2>  
<h3>Heading level 3</h3>  
<h4>Heading level 4</h4>  
<h5>Heading level 5</h5>  
<h6>Heading level 6</h6>
```

Un navigateur web rend ce code HTML comme indiqué dans la [Figure 2](#).

Headline level 1 to uniquely identify the page

Headline level 2

Headline level 3

Headline level 4

Headline level 5

Headline level 6

Figure 2. Représentation du code HTML dans le navigateur web, montrant les différents niveaux de titres dans un document HTML. La hiérarchie des titres est indiquée par la taille du texte.

Si vous êtes familier avec les logiciels de traitements de texte tels que LibreOffice ou Microsoft Word, vous remarquerez peut-être certaines similitudes dans la façon dont un document HTML utilise différents niveaux de titres et comment ils sont rendus dans le navigateur web. Par défaut, le HTML utilise la taille pour indiquer la hiérarchie et l'importance des titres et ajoute un espace avant et après chaque titre pour le séparer visuellement du contenu.

Un titre utilisant l'élément `<h1>` se trouve au sommet de la hiérarchie et il est donc considéré comme le titre le plus important qui identifie le contenu de la page. Il est comparable à l'élément `<title>` abordé dans la leçon précédente, mais au sein du contenu du document HTML. Les éléments titres suivants peuvent être utilisés pour structurer davantage le contenu. Veillez à ne pas sauter de niveaux de titre entre eux. La hiérarchie du document doit commencer par `<h1>`, se poursuivre par `<h2>`, puis `<h3>` et ainsi de suite. Il n'est pas nécessaire d'utiliser tous les éléments de titre jusqu'à `<h6>` si votre contenu ne l'exige pas.

NOTE

Les titres sont des outils importants pour structurer un document HTML, tant sur le plan sémantique que visuel. Cependant, les titres ne doivent jamais être utilisés pour augmenter la taille d'un texte sans importance structurelle. Selon le même principe, il ne faut pas mettre un court paragraphe en gras ou en italique pour le faire ressembler à un titre ; utilisez les balises de titre pour marquer les titres.

Commençons par créer le document HTML de la liste de courses en définissant son contour. Nous allons créer un élément `<h1>` pour contenir le titre de la page, dans ce cas `Garden Party`, suivi d'informations courtes enveloppées dans un élément `<p>`. De plus, nous utilisons deux éléments `<h2>`

pour introduire les deux sections de contenu `Agenda` et `` Please bring``.

```
<h1>Garden Party</h1>
<p>Invitation to John's garden party on Saturday next week.</p>
<h2>Agenda</h2>
<h2>Please bring</h2>
```

Ouvert dans un navigateur web, ce code HTML produit le résultat indiqué dans la [Figure 3](#).

Garden Party

Invitation to John's garden party on Saturday next week.

Agenda

Please bring

Figure 3. Représentation du navigateur web du code HTML montrant un exemple simple de document décrivant une invitation à une garden-party, avec deux rubriques pour l'agenda et la liste des choses à apporter.

Sauts de ligne

Il peut parfois être nécessaire de provoquer un *saut de ligne* (*line break*) sans insérer un autre élément `<p>` ou tout autre élément de bloc similaire. Dans ce cas, vous pouvez utiliser l'élément `
` à fermeture automatique. Notez qu'il ne doit être utilisé que pour insérer des sauts de ligne qui appartiennent au contenu, comme c'est le cas pour les poèmes, les paroles de chansons ou les adresses. Si le contenu est séparé par le sens, il est préférable d'utiliser plutôt un élément `<p>`.

Par exemple, nous pourrions diviser le texte du paragraphe d'information de notre exemple précédent comme suit :

```
<p>
  Invitation to John's garden party.<br>
  Saturday, next week.
</p>
```

Dans un navigateur web, ce code HTML produit le résultat indiqué dans la [Figure 4](#).

Invitation to John's garden party.
Saturday, next week.

Figure 4. Représentation du code HTML dans le navigateur web, montrant un exemple simple de document avec un saut de ligne forcé.

Lignes horizontales

L'élément `<hr>` définit une ligne horizontale, également appelée *règle horizontale* (*horizontal rule*). Par défaut, il s'étend sur toute la largeur de son élément parent. L'élément `<hr>` peut vous aider à définir un changement thématique dans le contenu ou à séparer les sections du document. L'élément se ferme automatiquement et n'a donc pas de balise de fermeture.

Pour notre exemple, nous pourrions séparer les deux rubriques :

```
<h1>Garden Party</h1>
<p>Invitation to John's garden party on Saturday next week.</p>
<h2>Agenda</h2>
<hr>
<h2>Please bring</h2>
```

La [Figure 5](#) montre le résultat de ce code.

Garden Party

Invitation to John's garden party on Saturday next week.

Agenda

Please bring

Figure 5. Représentation par le navigateur web d'un exemple simple de document décrivant une liste de courses avec deux sections séparées par une ligne horizontale.

Listes HTML

En HTML, vous pouvez définir trois types de listes :

Listes ordonnées

où l'ordre des éléments de la liste est important

Listes non ordonnées

où l'ordre des éléments de la liste n'est pas particulièrement important

Listes de description

pour décrire plus précisément certains termes

Chacune contient un nombre quelconque d'*éléments de liste* (*list items*). Nous allons décrire chaque type de liste.

Listes ordonnées

Une *liste ordonnée* (*ordered list*) en HTML, désignée par l'élément HTML ``, est une collection d'un nombre quelconque d'*éléments de liste*. La particularité de cet élément est que l'ordre de ses éléments de liste est important. Pour le souligner, les navigateurs web affichent par défaut des chiffres avant les éléments de liste enfants.

NOTE

Les éléments `` sont les seuls éléments enfants valides au sein d'un élément ``.

Pour notre exemple, nous pouvons remplir l'agenda de la garden-party en utilisant un élément `` avec le code suivant :

```
<h2>Agenda</h2>
<ol>
  <li>Welcome</li>
  <li>Barbecue</li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Dans un navigateur web, ce code HTML produit le résultat indiqué dans la [Figure 6](#).

Agenda

1. Welcome
2. Barbecue
3. Dessert
4. Fireworks

Figure 6. Représentation par le navigateur web d'un exemple simple de document contenant un titre de second niveau suivi d'une liste ordonnée de quatre éléments décrivant l'ordre du jour d'une garden-party.

Options

Comme vous pouvez le voir dans cet exemple, les éléments de la liste sont numérotés avec des chiffres entiers commençant par 1 par défaut. Toutefois, vous pouvez modifier ce comportement en spécifiant l'attribut `type` de la balise ``. Les valeurs valides pour cet attribut sont `1` pour les chiffres entiers, `A` pour les lettres majuscules, `a` pour les lettres minuscules, `I` pour les chiffres romains en majuscules et `i` pour les chiffres romains en minuscules.

Si vous le souhaitez, vous pouvez également définir la valeur de départ en utilisant l'attribut `start` de la balise ``. L'attribut `start` prend toujours une valeur numérique entière, même si l'attribut `type` définit un autre type de numérotation.

Par exemple, nous pouvons ajuster la liste ordonnée de l'exemple précédent de sorte que les éléments de la liste soient préfixés par des lettres majuscules, en commençant par la lettre C, comme le montre l'exemple suivant :

```
<h2>Agenda</h2>
<ol type="A" start="3">
  <li>Welcome</li>
  <li>Barbecue</li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Dans un navigateur web, ce code HTML est rendu comme dans la [Figure 7](#).

Agenda

- C. Welcome
- D. Barbecue
- E. Dessert
- F. Fireworks

Figure 7. Représentation par le navigateur web d'un exemple simple de document contenant un titre de second niveau suivi d'une liste ordonnée dont les éléments sont préfixés par des lettres majuscules commençant par la lettre C.

L'ordre des éléments de la liste peut également être inversé en utilisant l'attribut `reversed` sans valeur.

NOTE

Dans une liste ordonnée, vous pouvez également définir la valeur initiale d'un élément de liste spécifique en utilisant l'attribut `value` de la balise ``. Les éléments de liste qui suivent s'incrémenteront à partir de ce nombre. L'attribut `value` prend toujours une valeur numérique entière.

Listes non ordonnées

Une *liste non ordonnée* (*unordered list*) contient une série d'éléments de liste qui, contrairement à ceux d'une liste ordonnée, n'ont pas d'ordre ou de séquence particulière. L'élément HTML de cette liste est ``. Une fois de plus, `` est l'élément HTML pour marquer ses éléments de liste.

NOTE

Les éléments `` sont les seuls éléments enfants valides au sein d'un élément ``.

Pour notre site web d'exemple, nous pouvons utiliser la liste non ordonnée pour énumérer les articles que les invités peuvent apporter à la fête. Nous pouvons y parvenir avec le code HTML suivant :

```
<h2>Please bring</h2>
<ul>
  <li>Salad</li>
  <li>Drinks</li>
  <li>Bread</li>
  <li>Snacks</li>
  <li>Desserts</li>
</ul>
```

Dans un navigateur web, ce code HTML produit l’affichage montré dans la [Figure 8](#).

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

Figure 8. Représentation par le navigateur web d’un document simple contenant un titre de deuxième niveau suivi d’une liste non ordonnée d’éléments de liste concernant les denrées alimentaires que les invités sont incités à apporter à la garden-party.

Par défaut, chaque élément de la liste est représenté par une puce de forme circulaire (disque). Vous pouvez modifier son apparence à l’aide du CSS, qui sera abordé dans les leçons suivantes.

Listes imbriquées

Les listes peuvent être imbriquées dans d’autres listes, comme les listes ordonnées dans les listes non ordonnées et vice versa. Pour ce faire, la liste imbriquée doit faire partie d’un élément de liste ``, car `` est le seul élément enfant valide des listes non ordonnées et ordonnées. Lors de l’imbrication, veillez à ne pas faire chevaucher les balises HTML.

Pour notre exemple, nous pouvons ajouter quelques informations sur l’agenda que nous avons créé auparavant, comme le montre l’exemple suivant :

```
<h2>Agenda</h2>
<ol type="A" start="3">
  <li>Welcome</li>
  <li>
    Barbecue
    <ul>
      <li>Vegetables</li>
      <li>Meat</li>
      <li>Burgers, including vegetarian options</li>
    </ul>
  </li>
  <li>Dessert</li>
  <li>Fireworks</li>
</ol>
```

Un navigateur web affiche le code comme indiqué dans la [Figure 9](#).

Agenda

- C. Welcome
- D. Barbecue
 - Vegetables
 - Meat
 - Burgers, including vegetarian options
- E. Dessert
- F. Fireworks

Figure 9. Représentation du navigateur web du code HTML montrant une liste non ordonnée imbriquée dans une liste ordonnée, pour représenter l'ordre du jour d'une garden-party.

Vous pouvez même aller plus loin et imbriquer plusieurs niveaux de profondeur. Théoriquement, il n'y a pas de limite au nombre de listes que vous pouvez imbriquer. Toutefois, tenez compte de la lisibilité pour vos visiteurs.

Listes de description

Une *liste de description* (*description list*) est définie à l'aide de l'élément `<dl>` et elle représente un dictionnaire de *clés* et de *valeurs*. La clé est un terme ou un nom que vous souhaitez décrire, et la valeur est la description. Les listes de description peuvent aller de simples paires clé-valeur à des définitions exhaustives.

Une clé (ou *terme*) est définie à l'aide de l'élément `<dt>`, tandis que sa description est définie à l'aide de l'élément `<dd>`.

Un exemple d'une telle liste descriptive pourrait être une liste de fruits exotiques expliquant à quoi ils ressemblent.

```
<h3>Exotic Fruits</h3>
<dl>
  <dt>Banana</dt>
  <dd>
    A long, curved fruit that is yellow-skinned when ripe. The fruit's skin
    may also have a soft green color when underripe and get brown spots when
    overripe.
  </dd>
</dl>
```

```

<dt>Kiwi</dt>
<dd>
  A small, oval fruit with green flesh, black seeds, and a brown, hairy
  skin.
</dd>

<dt>Mango</dt>
<dd>
  A fruit larger than a fist, with a green skin, orange flesh, and one big
  seed. The skin may have spots ranging from green to yellow or red.
</dd>
</dl>

```

Dans un navigateur web, cela donne le résultat indiqué dans la [Figure 10](#).

Exotic Fruits

Banana

A long, curved fruit that is yellow-skinned when ripe. The fruit's skin may also have a soft green color when underripe and get brown spots when overripe.

Kiwi

A small, oval fruit with green flesh, black seeds and a brown, hairy skin.

Mango

A fruit larger than a fist, with a green skin, orange flesh, and one big seed. The skin may have spots ranging from green to yellow or red.

Figure 10. Un exemple de liste de description HTML utilisant des fruits exotiques. La liste décrit l'apparence de trois fruits exotiques différents.

NOTE

Contrairement aux listes ordonnées et aux listes non ordonnées, dans une liste de description, tout élément HTML peut être considéré comme un enfant direct. Cela vous permet de regrouper des éléments et de les styliser ailleurs à l'aide de CSS.

Mise en forme du texte

En HTML, vous pouvez utiliser des éléments de mise en forme pour modifier l'apparence du texte. Ces éléments peuvent être classés comme des *éléments de présentation* ou des *éléments de phrase*.

Éléments de présentation

Les éléments de présentation de base modifient la police ou l'apparence du texte ; ce sont les éléments ``, `<i>`, `<u>` et `<tt>`. Ces éléments ont été définis à l'origine avant que les CSS ne

permettent de mettre du texte en gras, en italique, etc. Il existe aujourd'hui de meilleures façons de modifier l'apparence du texte, mais vous verrez encore ces éléments.

Texte en gras

Pour rendre le texte en gras (*bold*), vous pouvez l'envelopper dans l'élément `` comme illustré dans l'exemple suivant. Le résultat apparaît dans la [Figure 11](#).

```
This <b>word</b> is bold.
```

This **word** is bold.

Figure 11. Balise `` utilisée pour rendre le texte en gras.

Selon la spécification HTML5, l'élément `` doit être utilisé uniquement lorsqu'il n'existe pas de balises plus appropriées. L'élément qui produit le même résultat visuel, mais qui ajoute une importance sémantique au texte marqué, est ``.

Texte en italique

Pour mettre du texte en italique, vous pouvez l'envelopper dans l'élément `<i>` comme illustré dans l'exemple suivant. Le résultat apparaît dans la [Figure 12](#).

```
This <i>word</i> is in italics.
```

This *word* is in italics.

Figure 12. Balise `<i>` utilisée pour mettre le texte en italique.

Selon la spécification HTML5, l'élément `<i>` ne doit être utilisé que lorsqu'il n'existe pas de balises plus appropriées.

Texte souligné

Pour souligner (*underline*) du texte, vous pouvez l'envelopper dans l'élément `<u>` comme illustré dans l'exemple suivant. Le résultat apparaît dans la [Figure 13](#).

```
This <u>word</u> is underlined.
```

This word is underlined.

Figure 13. Balise `<u>` utilisée pour souligner du texte.

Selon la spécification HTML5, l'élément `<u>` ne doit être utilisé que lorsqu'il n'existe pas de meilleur moyen de souligner du texte. CSS fournit une alternative moderne.

Police à largeur fixe ou monospace

Pour afficher du texte dans une police monospace (à largeur fixe), souvent utilisée pour afficher du code informatique, vous pouvez utiliser l'élément `<tt>` comme illustré dans l'exemple suivant. Le résultat apparaît dans la [Figure 14](#).

This `<tt>word</tt>` is in fixed-width font.

This word is in fixed-width font.

Figure 14. Balise `<tt>` utilisée pour afficher du texte dans une police à largeur fixe.

La balise `<tt>` n'est pas prise en charge par HTML5. Les navigateurs web la rendent toujours comme prévu. Toutefois, vous devez utiliser des balises plus appropriées, notamment `<code>`, `<kbd>`, `<var>` et `<samp>`.

Éléments de phrase

Les éléments de phrase ne modifient pas seulement l'apparence du texte, mais ajoutent également une importance sémantique à un mot ou à une phrase. En les utilisant, vous pouvez mettre en valeur un mot ou le marquer comme important. Ces éléments, contrairement aux éléments de présentation, sont reconnus par les lecteurs d'écran, ce qui rend le texte plus accessible aux visiteurs malvoyants et permet aux moteurs de recherche de mieux lire et évaluer le contenu de la page. Les éléments de phrase que nous utilisons tout au long de cette leçon sont ``, ``, et `<code>`.

Texte mis en évidence

Pour mettre du texte en évidence (*Emphasized Text*), vous pouvez l'entourer de l'élément ``, comme le montre l'exemple suivant :

This `word` is emphasized.

This *word* is emphasized.

Figure 15. Balise `` utilisée pour mettre en évidence le texte.

Comme vous pouvez le constater, les navigateurs web affichent `` de la même manière que `<i>`, mais `` ajoute une importance sémantique en tant qu'élément de phrase, ce qui améliore l'accessibilité pour les visiteurs malvoyants.

Texte important

Pour marquer le texte comme important, vous pouvez l'envelopper dans l'élément `` comme illustré dans l'exemple suivant. Le résultat apparaît dans la [Figure 16](#).

This **word** is important.

This **word** is important.

Figure 16. Balise `` utilisée pour marquer le texte comme important.

Comme vous pouvez le constater, les navigateurs web affichent `` de la même manière que ``, mais `` ajoute une importance sémantique en tant qu'élément de phrase, ce qui améliore l'accessibilité pour les visiteurs malvoyants.

Code informatique

Pour insérer un morceau de code informatique, vous pouvez l'envelopper dans l'élément `<code>` comme illustré dans l'exemple suivant. Le résultat apparaît dans la [Figure 17](#).

The Markdown code `# Heading` creates a heading at the highest level in the hierarchy.

The Markdown code `# Heading` creates a heading at the highest level in the hierarchy.

Figure 17. Balise `<code>` utilisée pour insérer un morceau de code informatique.

Texte marqué

Pour mettre en évidence un texte sur un fond jaune, à la manière d'un surligneur, vous pouvez utiliser l'élément `<mark>` comme illustré dans l'exemple suivant. Le résultat apparaît dans la [Figure](#)

18.

This **<mark>word</mark>** is highlighted.

This **word** is highlighted.

Figure 18. Balise `<mark>` utilisée pour mettre en évidence du texte sur un fond jaune.

Formatage du texte de notre liste d'achats en HTML

En nous inspirant de nos exemples précédents, nous allons insérer quelques éléments de phrase pour modifier l'apparence du texte tout en lui donnant une importance sémantique. Le résultat apparaît dans la [Figure 19](#).

```
<h1>Garden Party</h1>
<p>
  Invitation to <strong>John's garden party</strong>.<br>
  <strong>Saturday, next week.</strong>
</p>

<h2>Agenda</h2>
<ol>
  <li>Welcome</li>
  <li>
    Barbecue
    <ul>
      <li><em>Vegetables</em></li>
      <li><em>Meat</em></li>
      <li><em>Burgers</em>, including vegetarian options</li>
    </ul>
  </li>
  <li>Dessert</li>
  <li><mark>Fireworks</mark></li>
</ol>

<hr>

<h2>Please bring</h2>
<ul>
  <li>Salad</li>
  <li>Drinks</li>
  <li>Bread</li>
  <li>Snacks</li>
```

```
<li>Desserts</li>
</ul>
```

Garden Party

Invitation to **John's garden party**.
Saturday, next week.

Agenda

1. Welcome
2. Barbecue
 - *Vegetables*
 - *Meat*
 - *Burgers*, including vegetarian options
3. Dessert
4. **Fireworks**

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

Figure 19. Page HTML avec quelques éléments de mise en forme.

Dans cet exemple de document HTML, les informations les plus importantes concernant la garden-party elle-même sont marquées comme importantes à l'aide de l'élément ``. Les aliments disponibles pour le barbecue sont mis en évidence à l'aide de l'élément ``. Les feux d'artifice sont simplement mis en évidence à l'aide de l'élément `<mark>`.

À titre d'exercice, vous pouvez essayer de formater d'autres morceaux de texte en utilisant également les autres éléments de formatage.

Texte préformaté

Dans la plupart des éléments HTML, l'espace blanc est généralement réduit à un seul espace, voire entièrement ignoré. Toutefois, il existe un élément HTML nommé `<pre>` qui permet de définir ce que l'on appelle du *texte préformaté*. L'espace blanc dans le contenu de cet élément, y compris les espaces et les sauts de ligne, est préservé et rendu dans le navigateur web. En outre, le texte s'affiche dans une police à largeur fixe, comme l'élément `<code>`.

```
<pre>
field() {
    shift $1 ; echo $1
}
</pre>
```

```
field() {
    shift $1 ; echo $1
}
```

Figure 20. Représentation du code HTML dans le navigateur web, illustrant comment l'élément HTML `<pre>` préserve les espaces blancs.

Regroupement des éléments

Par convention, les éléments HTML sont divisés en deux catégories :

Éléments de niveau bloc

Ils apparaissent sur une nouvelle ligne et occupent toute la largeur disponible. Les exemples d'éléments de niveau bloc dont nous avons déjà parlé sont `<p>`, ``, et `<h2>`.

Éléments de niveau ligne

Ils apparaissent sur la même ligne (*inline*) que les autres éléments et le texte, n'occupant que l'espace nécessaire à leur contenu. Des exemples d'éléments de niveau ligne sont ``, ``, et `<i>`.

NOTE

HTML5 a introduit des catégories d'éléments plus précises et plus exactes, en essayant d'éviter toute confusion avec les blocs CSS et les boîtes en ligne. Pour simplifier, nous nous en tiendrons ici à la subdivision classique en éléments en bloc et éléments en ligne.

Les éléments fondamentaux pour regrouper plusieurs éléments sont les éléments `<div>` et ``.

L'élément `<div>` est un conteneur de niveau bloc pour d'autres éléments HTML et n'ajoute pas de valeur sémantique par lui-même. Vous pouvez utiliser cet élément pour diviser un document HTML en sections et structurer votre contenu, tant pour la lisibilité du code que pour appliquer des styles CSS à un groupe d'éléments, comme vous l'apprendrez dans une leçon ultérieure.

Par défaut, les navigateurs web insèrent toujours un saut de ligne avant et après chaque élément `<div>` afin qu'ils s'affichent sur leur propre ligne.

En revanche, l'élément `` sert de conteneur pour le texte HTML et il est généralement utilisé pour regrouper d'autres éléments en ligne afin d'appliquer des styles à l'aide de CSS à une plus petite partie du texte.

L'élément `` se comporte comme du texte ordinaire et ne commence pas sur une nouvelle ligne. Il s'agit donc d'un élément en ligne.

L'exemple suivant compare la représentation visuelle de l'élément sémantique `<p>` et des éléments de regroupement `<div>` et `` :

```
<p>Text within a paragraph</p>
<p>Another paragraph of text</p>
<hr>
<div>Text wrapped within a <code>div</code> element</div>
<div>Another <code>div</code> element with more text</div>
<hr>
<span>Span content</span>
<span>and more span content</span>
```

Un navigateur web affiche ce code comme indiqué dans la [Figure 21](#).

Text within a paragraph

Another paragraph of text

Text wrapped within a `div` element
Another `div` element with more text

Span content and more span content

Figure 21. Représentation par le navigateur web d'un document de test illustrant les différences entre les éléments `p` (paragraph), `div` et `span` du langage HTML.

Nous avons déjà vu que par défaut, le navigateur web ajoute un espacement avant et après les éléments `<p>`. Cet espacement n'est appliqué à aucun des éléments de regroupement `<div>` et ``. Cependant, les éléments `<div>` sont formatés comme leurs propres blocs, tandis que le texte des éléments `` sont affichés sur la même ligne.

Structure de page HTML

Nous avons vu comment utiliser les éléments HTML pour décrire le contenu d'une page web d'un point de vue sémantique, c'est-à-dire pour donner un sens et un contexte au texte. Un autre groupe d'éléments est conçu dans le but de décrire la *structure sémantique* d'une page web, une expression ou sa structure. Ces éléments sont des éléments bloc, c'est-à-dire qu'ils se comportent visuellement de la même manière qu'un élément `<div>`. Leur but est de définir la structure sémantique d'une page web en spécifiant des zones bien définies telles que les en-têtes, les pieds de page et le contenu principal de la page. Ces éléments permettent le regroupement sémantique du contenu afin qu'il puisse être compris par un ordinateur également, y compris les moteurs de recherche et les lecteurs d'écran.

Élément `<header>`

L'élément `<header>` contient des informations d'introduction aux éléments sémantiques dans un document HTML. Un en-tête est différent d'un titre, mais un en-tête comprend souvent un élément de titre (`<h1>`, ... , `<h6>`).

En pratique, cet élément est le plus souvent utilisé pour représenter l'en-tête de la page, comme une bannière avec un logo. Il peut également être utilisé pour introduire le contenu de l'un des éléments suivants : `<body>`, `<section>`, `<article>`, `<nav>`, ou `<aside>`.

Un document peut comporter plusieurs éléments `<header>`, mais un élément `<header>` ne peut pas être imbriqué dans un autre élément `<header>`. Un élément `<footer>` ne peut pas non plus être utilisé à l'intérieur d'un `<header>`.

Par exemple, pour ajouter un en-tête de page à notre document d'exemple, nous pouvons procéder comme suit :

```
<header>
  <h1>Garden Party</h1>
</header>
```

Il n'y aura aucun changement visible dans le document HTML, car `<h1>` (comme tous les autres éléments d'en-tête) est un élément de niveau bloc sans autres propriétés visuelles.

Élément de contenu <main>

L'élément <main> est un conteneur pour le contenu central d'une page web. Il ne doit pas y avoir plus d'un élément <main> dans un document HTML.

Dans notre document d'exemple, tout le code HTML que nous avons écrit jusqu'à présent serait placé à l'intérieur de l'élément <main>.

```
<main>
  <header>
    <h1>Garden Party</h1>
  </header>
  <p>
    Invitation to <strong>John's garden party</strong>.<br>
    <strong>Saturday, next week.</strong>
  </p>

  <h2>Agenda</h2>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>

  <hr>

  <h2>Please bring</h2>
  <ul>
    <li>Salad</li>
    <li>Drinks</li>
    <li>Bread</li>
    <li>Snacks</li>
    <li>Desserts</li>
  </ul>
</main>
```

Comme l'élément `<header>`, l'élément `<main>` ne provoque aucun changement visuel dans notre exemple.

Élément `<footer>`

L'élément `<footer>` contient des notes de bas de page, par exemple des informations sur l'auteur, des informations de contact ou des documents connexes, pour les éléments sémantiques environnants, par exemple `<section>`, `<nav>`, ou `<aside>`. Un document peut avoir plusieurs éléments `<footer>` qui permettent de mieux décrire les éléments sémantiques. Toutefois, un élément `<footer>` ne peut pas être imbriqué dans un autre élément `<footer>`, et un élément `<header>` ne peut pas non plus être utilisé dans un `<footer>`.

Pour notre exemple, nous pouvons ajouter des informations de contact pour l'hôte (John) comme indiqué dans l'exemple suivant :

```
<footer>
  <p>John Doe</p>
  <p>john.doe@example.com</p>
</footer>
```

Élément `<nav>`

L'élément `<nav>` décrit une unité de navigation majeure, telle qu'un menu, qui contient une série d'hyperliens.

NOTE

Les hyperliens ne doivent pas tous être enveloppés dans un élément `<nav>`. C'est utile pour lister un groupe de liens.

Les hyperliens n'ayant pas encore été abordés, l'élément de navigation ne sera pas inclus dans l'exemple de cette leçon.

Élément `<aside>`

L'élément `<aside>` est un conteneur pour le contenu qui n'est pas nécessaire dans l'ordre du contenu de la page principale, mais qui est généralement indirectement lié ou complémentaire. Cet élément est souvent utilisé pour les barres latérales qui affichent des informations secondaires, comme un glossaire.

Dans notre exemple, nous pouvons ajouter des informations sur l'adresse et le trajet, qui ne sont qu'indirectement liées au reste du contenu, en utilisant l'élément `<aside>`.

```
<aside>
  <p>
    10, Main Street<br>
    Newville
  </p>
  <p>Parking spaces available.</p>
</aside>
```

Élément <section>

L'élément <section> définit une section logique dans un document qui fait partie de l'élément sémantique qui l'entoure, mais qui ne fonctionnerait pas comme un contenu autonome, tel qu'un chapitre.

Dans notre document d'exemple, nous pouvons envelopper les sections de contenu pour l'ordre du jour et introduire des sections de liste comme le montre l'exemple suivant :

```
<section>
  <header>
    <h2>Agenda</h2>
  </header>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</section>

<hr>

<section>
  <header>
    <h2>Please bring</h2>
  </header>
  <ul>
```



```

<li>Salad</li>
<li>Drinks</li>
<li>Bread</li>
<li>Snacks</li>
<li>Desserts</li>
</ul>
</section>

```

Cet exemple ajoute également d'autres éléments `<header>` à l'intérieur des sections, de sorte que chaque section contient son propre élément `<header>`.

Élément `<article>`

L'élément `<article>` définit un contenu indépendant et autonome qui a un sens par lui-même sans le reste de la page. Son contenu est potentiellement redistribuable ou réutilisable dans un autre contexte. Les exemples typiques ou le matériel approprié pour un élément `<article>` sont un article de blog, une liste de produits pour une boutique et une publicité pour un produit. La publicité pourrait alors exister à la fois en tant que telle et dans une page plus grande. Dans notre exemple, nous pouvons remplacer la première `<section>` qui enveloppe l'agenda par un élément `<article>`.

```

<article>
  <header>
    <h2>Agenda</h2>
  </header>
  <ol>
    <li>Welcome</li>
    <li>
      Barbecue
      <ul>
        <li><em>Vegetables</em></li>
        <li><em>Meat</em></li>
        <li><em>Burgers</em>, including vegetarian options</li>
      </ul>
    </li>
    <li>Dessert</li>
    <li><mark>Fireworks</mark></li>
  </ol>
</article>

```

L'élément `<header>` que nous avons ajouté dans l'exemple précédent peut persister ici aussi, car les éléments `<article>` peuvent avoir leurs propres éléments `<header>`.

Dernier exemple

En combinant tous les exemples précédents, le document HTML final de notre invitation se présente comme suit :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Garden Party</title>
  </head>

  <body>
    <main>
      <h1>Garden Party</h1>
      <p>
        Invitation to <strong>John's garden party</strong>.<br>
        <strong>Saturday, next week.</strong>
      </p>

      <article>
        <h2>Agenda</h2>
        <ol>
          <li>Welcome</li>
          <li>
            Barbecue
            <ul>
              <li><em>Vegetables</em></li>
              <li><em>Meat</em></li>
              <li><em>Burgers</em>, including vegetarian options</li>
            </ul>
          </li>
          <li>Dessert</li>
          <li><mark>Fireworks</mark></li>
        </ol>
      </article>

      <hr>

      <section>
        <h2>Please bring</h2>
        <ul>
          <li>Salad</li>
          <li>Drinks</li>
          <li>Bread</li>
          <li>Snacks</li>
        </ul>
      </section>
    </main>
  </body>
</html>
```

```
</li>Desserts</li>
</ul>
</section>
</main>

<aside>
  <p>
    10, Main Street<br>
    Newville
  </p>
  <p>Parking spaces available.</p>
</aside>

<footer>
  <p>John Doe</p>
  <p>john.doe@example.com</p>
</footer>
</body>
</html>
```

Dans un navigateur web, la page entière est affichée comme indiqué dans la [Figure 22](#).

Garden Party

Invitation to **John's garden party**.
Saturday, next week.

Agenda

1. Welcome
2. Barbecue
 - *Vegetables*
 - *Meat*
 - *Burgers*, including vegetarian options
3. Dessert
4. **Fireworks**

Please bring

- Salad
- Drinks
- Bread
- Snacks
- Desserts

10, Main Street
Newville

Parking spaces available.

John Doe

john.doe@example.com

Figure 22. Représentation dans le navigateur web du document HTML résultant de la combinaison des exemples précédents. La page représente une invitation à une garden-party et décrit le programme de la soirée ainsi qu'une liste de nourriture à apporter par les invités.

Exercices guidés

1. Pour chacune des balises suivantes, indiquez la balise fermante correspondante :

<h5>	
	
<dd>	
<hr>	
	
<tt>	
<main>	

2. Pour chacune des balises suivantes, indiquez si elle marque le début d'un élément de bloc ou d'un élément en ligne :

<h3>	
	
	
<div>	
	
<dl>	
	
<nav>	
<code>	
<pre>	

3. Quels types de listes pouvez-vous créer en HTML ? Quelles balises devez-vous utiliser pour chacune d'entre elles ?

4. Quelles balises entourent les éléments de bloc que vous pouvez utiliser pour structurer une page HTML ?

Exercices d'exploration

1. Créez une page HTML basique avec le titre “Form Rules”. Vous utiliserez cette page HTML pour tous les exercices d’exploration, chacun d’entre eux étant basé sur les précédents. Ajoutez ensuite un titre de niveau 1 avec le texte “How to fill in the request form”, un paragraphe avec le texte “To receive the PDF document with the complete HTML course, it is necessary to fill in the following fields:” et une liste non ordonnée avec les éléments de liste suivants : “Name”, “Surname”, “Email Address”, “Nation”, “Country”, et “Zip/Postal Code”.

2. Mettez les trois premiers champs (“Name”, “Surname” et “Email Address”) en gras, tout en leur donnant une importance sémantique. Ajoutez ensuite un titre de niveau 2 avec le texte “Required fields” et un paragraphe avec le texte “Bold fields are mandatory.”

3. Ajoutez un autre titre de niveau 2 avec le texte “Steps to follow”, un paragraphe avec le texte “There are four steps to follow:”, et une liste ordonnée avec les éléments de liste suivants : “Fill in the fields”, “Click the Submit button”, “Check your e-mail and confirm your request by clicking on the link you receive”, et “Check your e-mail - You will receive the full HTML course in minutes”.

4. En utilisant `<div>`, créez un bloc pour chaque section qui commence par un titre de niveau 2.

5. En utilisant `<div>`, créez un autre bloc pour la section commençant par le titre de niveau 1. Divisez ensuite cette section des deux autres avec une ligne horizontale.

6. Ajoutez l’élément header avec le texte “Form Rules - 2021” et l’élément `<footer>` avec le texte “Copyright Note - 2021”. Enfin, ajoutez l’élément `<main>` qui doit contenir les trois blocs `<div>`.

Résumé

Dans cette leçon, vous avez appris :

- Comment créer des balises pour le contenu d'un document HTML
- La structure hiérarchique du texte HTML
- La différence entre les éléments HTML en bloc et en ligne
- Comment créer des documents HTML avec une structure sémantique

Les termes suivants ont été abordés dans cette leçon :

<h1>, <h2>, <h3>, <h4>, <h5>, <h6>

Les balises de titre.

<p>

La balise de paragraphe.

La balise de liste ordonnée (*ordered list*).

La balise de liste non ordonnée (*unordered list*).

La balise de l'élément de liste (*list item*).

<dl>

La balise de la liste de description (*description list*).

<dt>, <dd>

Les balises de chaque terme et de chaque description pour une liste de descriptions.

<pre>

La balise de mise en forme préservée.

, <i>, <u>, <tt>, , , <code>, <mark>

Les balises de mise en forme.

**<div>, **

Les balises de regroupement.

<header>, <main>, <nav>, <aside>, <footer>

Les balises utilisées pour fournir une structure et une mise en page simples à une page HTML.

Réponses aux exercices guidés

1. Pour chacune des balises suivantes, indiquez la balise fermante correspondante :

<h5>	</h5>
 	N'existe pas
	
<dd>	</dd>
<hr>	N'existe pas
	
<tt>	</tt>
<main>	</main>

2. Pour chacune des balises suivantes, indiquez si elle marque le début d'un élément de bloc ou d'un élément en ligne :

<h3>	Élément de bloc
	Élément en ligne
	Élément en ligne
<div>	Élément de bloc
	Élément en ligne
<dl>	Élément de bloc
	Élément de bloc
<nav>	Élément de bloc
<code>	Élément en ligne
<pre>	Élément de bloc

3. Quels types de listes pouvez-vous créer en HTML ? Quelles balises devez-vous utiliser pour chacune d'entre elles ?

En HTML, vous pouvez créer trois types de listes : des listes ordonnées constituées d'une série d'éléments de liste numérotés, des listes non ordonnées constituées d'une série d'éléments de liste qui n'ont pas d'ordre ou de séquence particulière, et des listes de description représentant des entrées comme dans un dictionnaire ou une encyclopédie. Une liste ordonnée est comprise entre les balises et , une liste non ordonnée est comprise entre les balises et

``, et une liste de description est comprise entre les balises `<dl>` et `</dl>`. Chaque élément d'une liste ordonnée ou non ordonnée est inclus entre les balises `` et ``, tandis que chaque terme d'une liste de description est inclus entre les balises `<dt>` et `</dt>` et sa description est incluse entre les balises `<dd>` et `</dd>`.

4. Quelles balises contiennent les éléments de bloc que vous pouvez utiliser pour structurer une page HTML ?

Les balises `<header>` et `</header>` renferment l'en-tête de la page, les balises `<main>` et `</main>` renferment le contenu principal de la page HTML, les balises `<nav>` et `</nav>` contiennent la barre de navigation, les balises `<aside>` et `</aside>` contiennent la barre latérale et les balises `<footer>` et `</footer>` contiennent le pied de page.

Réponses aux exercices d'exploration

1. Créez une page HTML basique avec le titre “Form Rules”. Vous utiliserez cette page HTML pour tous les exercices d’exploration, chacun d’entre eux étant basé sur les précédents. Ajoutez ensuite un titre de niveau 1 avec le texte “How to fill in the request form”, un paragraphe avec le texte “To receive the PDF document with the complete HTML course, it is necessary to fill in the following fields:” et une liste non ordonnée avec les éléments de liste suivants : “Name”, “Surname”, “Email Address”, “Nation”, “Country”, et “Zip/Postal Code”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li>Name</li>
      <li>Surname</li>
      <li>Email Address</li>
      <li>Nation</li>
      <li>Country</li>
      <li>Zip/Postal Code</li>
    </ul>
  </body>
</html>
```

2. Mettez les trois premiers champs (“Name”, “Surname” et “Email Address”) en gras, tout en leur donnant une importance sémantique. Ajoutez ensuite un titre de niveau 2 avec le texte “Required fields” et un paragraphe avec le texte “Bold fields are mandatory.”

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>
```

```

<body>
  <h1>How to fill in the request form</h1>
  <p>
    To receive the PDF document with the complete HTML course, it is
    necessary
    to fill in the following fields:
  </p>
  <ul>
    <li><strong> Name </strong></li>
    <li><strong> Surname </strong></li>
    <li><strong> Email Address </strong></li>
    <li>Nation</li>
    <li>Country</li>
    <li>Zip/Postal Code</li>
  </ul>

  <h2>Required fields</h2>
  <p>Bold fields are mandatory.</p>
</body>
</html>

```

3. Ajoutez un autre titre de niveau 2 avec le texte “Steps to follow”, un paragraphe avec le texte “There are four steps to follow:”, et une liste ordonnée avec les éléments de liste suivants : “Fill in the fields”, “Click the Submit button”, “Check your e-mail and confirm your request by clicking on the link you receive”, et “Check your e-mail - You will receive the full HTML course in minutes”.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>

```

```

<li><strong> Surname </strong></li>
<li><strong> Email Address </strong></li>
<li>Nation</li>
<li>Country</li>
<li>Zip/Postal Code</li>
</ul>

<h2>Required fields</h2>
<p>Bold fields are mandatory.</p>

<h2>Steps to follow</h2>
<p>There are four steps to follow:</p>
<ol>
  <li>Fill in the fields</li>
  <li>Click the Submit button</li>
  <li>
    Check your e-mail and confirm your request by clicking on the link you
    receive
  </li>
  <li>
    Check your e-mail – You will receive the full HTML course in minutes
  </li>
</ol>
</body>
</html>

```

4. En utilisant `<div>`, créez un bloc pour chaque section qui commence par un titre de niveau 2.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <h1>How to fill in the request form</h1>
    <p>
      To receive the PDF document with the complete HTML course, it is
      necessary
      to fill in the following fields:
    </p>
    <ul>
      <li><strong> Name </strong></li>
      <li><strong> Surname </strong></li>
    </ul>
  </body>
</html>

```

```

    <li><strong> Email Address </strong></li>
    <li>Nation</li>
    <li>Country</li>
    <li>Zip/Postal Code</li>
</ul>

<div>
  <h2>Required fields</h2>
  <p>Bold fields are mandatory.</p>
</div>

<div>
  <h2>Steps to follow</h2>
  <p>There are four steps to follow:</p>
  <ol>
    <li>Fill in the fields</li>
    <li>Click the Submit button</li>
    <li>
      Check your e-mail and confirm your request by clicking on the link
you
      receive
    </li>
    <li>
      Check your e-mail – You will receive the full HTML course in minutes
    </li>
  </ol>
</div>
</body>
</html>

```

5. En utilisant `<div>`, créez un autre bloc pour la section commençant par le titre de niveau 1. Divisez ensuite cette section des deux autres avec une ligne horizontale.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Form Rules</title>
  </head>

  <body>
    <div>
      <h1>How to fill in the request form</h1>
      <p>
        To receive the PDF document with the complete HTML course, it is

```

```

        necessary to fill in the following fields:
    </p>
    <ul>
        <li><strong> Name </strong></li>
        <li><strong> Surname </strong></li>
        <li><strong> Email Address </strong></li>
        <li>Nation</li>
        <li>Country</li>
        <li>Zip/Postal Code</li>
    </ul>
</div>

<hr>

<div>
    <h2>Required fields</h2>
    <p>Bold fields are mandatory.</p>
</div>

<div>
    <h2>Steps to follow</h2>
    <p>There are four steps to follow:</p>
    <ol>
        <li>Fill in the fields</li>
        <li>Click the Submit button</li>
        <li>
            Check your e-mail and confirm your request by clicking on the link
            you
            receive
        </li>
        <li>
            Check your e-mail – You will receive the full HTML course in minutes
        </li>
    </ol>
</div>
</body>
</html>

```

6. Ajoutez l'élément `header` avec le texte "Form Rules - 2021" et l'élément `footer` avec le texte "Copyright Note - 2021". Enfin, ajoutez l'élément `main` qui doit contenir les trois blocs `div`.

```

<!DOCTYPE html>
<html>
  <head>

```

```
<title>Form Rules</title>
</head>

<body>
  <header>
    <h1>Form Rules – 2021</h1>
  </header>

  <main>
    <div>
      <h1>How to fill in the request form</h1>
      <p>
        To receive the PDF document with the complete HTML course, it is
        necessary to fill in the following fields:
      </p>
      <ul>
        <li><strong> Name </strong></li>
        <li><strong> Surname </strong></li>
        <li><strong> Email Address </strong></li>
        <li>Nation</li>
        <li>Country</li>
        <li>Zip/Postal Code</li>
      </ul>
    </div>

    <hr>

    <div>
      <h2>Required fields</h2>
      <p>Bold fields are mandatory.</p>
    </div>

    <div>
      <h2>Steps to follow</h2>
      <p>There are four steps to follow:</p>
      <ol>
        <li>Fill in the fields</li>
        <li>Click the Submit button</li>
        <li>
          Check your e-mail and confirm your request by clicking on the link
          you receive
        </li>
        <li>
          Check your e-mail – You will receive the full HTML course in
          minutes
        </li>
      </ol>
    </div>
  </main>
</body>
```



```
        </li>
      </ol>
    </div>
  </main>

  <footer>
    <p>Copyright Note – 2021</p>
  </footer>
</body>
</html>
```



032.3 Références HTML et ressources intégrées

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 032.3

Valeur

2

Domaines de connaissance les plus importants

- Créer des liens vers des ressources externes et des ancres de page
- Ajouter des images aux documents HTML
- Comprendre les principales propriétés des formats de fichiers médias courants, notamment PNG, JPG et SVG.
- Connaître les iframes

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- Attribut `id`
- `<a>`, y compris les attributs `href` et `target` (`_blank`, `_self`, `_parent`, `_top`)
- ``, y compris les attributs `src` et `alt`



Linux
Professional
Institute

032.3 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	032 Balisage de documents HTML
Objectif :	032.3 Références HTML et ressources intégrées
Leçon :	1 sur 1

Introduction

Une page web moderne est rarement constituée uniquement de texte. Elle comprend de nombreux autres types de contenus, tels que des images, du son, de la vidéo et même d'autres documents HTML. Outre le contenu externe, les documents HTML peuvent contenir des liens vers d'autres documents, ce qui simplifie considérablement la navigation sur Internet.

Contenu Intégré

L'échange de fichiers est possible sur l'internet sans pages web écrites en HTML, alors pourquoi le HTML est-il le format choisi pour les documents web et non le PDF ou tout autre format de traitement de texte ? Une raison importante est que le HTML conserve ses ressources multimédia dans des fichiers séparés. Dans un environnement tel que l'internet, où les informations sont souvent redondantes et distribuées à différents endroits, il est important d'éviter les transferts de données inutiles. La plupart du temps, les nouvelles versions d'une page web contiennent les mêmes images et autres fichiers de support que les versions précédentes, de sorte que le navigateur web peut utiliser les fichiers précédemment récupérés au lieu de tout recopier. En outre, la séparation des fichiers facilite la personnalisation du contenu multimédia en fonction des caractéristiques du client, telles que sa localisation, la taille de son écran et sa vitesse de connexion.

Images

Le type le plus courant de contenu intégré est constitué par les images qui accompagnent le texte. Les images sont conservées séparément et sont référencées à l'intérieur du fichier HTML avec la balise `` :

```

```

La balise `` ne nécessite pas de balise de fermeture. La propriété `src` indique l'emplacement de la source du fichier image. Dans cet exemple, le fichier image `logo.png` doit être situé dans le même répertoire que le fichier HTML, sinon le navigateur ne pourra pas l'afficher. La propriété d'emplacement source accepte les chemins relatifs, de sorte que la *notation par points* peut être utilisée pour indiquer le chemin d'accès à l'image :

```

```

Les deux points indiquent que l'image est située dans le répertoire parent par rapport au répertoire où se trouve le fichier HTML. Si le nom de fichier `../logo.png` est utilisé dans un fichier HTML dont l'URL est `http://example.com/library/periodicals/index.html`, le navigateur demandera le fichier image à l'adresse `http://example.com/library/logo.png`.

La notation par points s'applique également si le fichier HTML n'est pas un fichier réel dans le système de fichiers ; le navigateur HTML interprète l'URL comme s'il s'agissait d'un chemin d'accès à un fichier, mais c'est au serveur HTTP de décider si ce chemin d'accès fait référence à un fichier ou à un contenu généré dynamiquement. Le domaine et le chemin d'accès approprié sont automatiquement ajoutés à toutes les demandes adressées au serveur, au cas où le fichier HTML proviendrait d'une demande HTTP. De même, le navigateur ouvrira l'image appropriée si le fichier HTML a été ouvert directement à partir du système de fichiers local.

Les emplacements de source commençant par une barre oblique `/` sont traités comme des chemins absolus. Les chemins absolus contiennent des informations complètes sur l'emplacement de l'image, de sorte qu'ils fonctionnent indépendamment de l'emplacement du document HTML. Si le fichier image est situé sur un autre serveur, ce qui sera le cas lorsqu'un *réseau de diffusion de contenu* (CDN : *Content Diffusion Network*) est utilisé, le nom de domaine doit également être inclus.

NOTE

Les réseaux de diffusion de contenu sont composés de serveurs répartis géographiquement qui stockent du contenu statique pour d'autres sites web. Ils permettent d'améliorer les performances et la disponibilité des sites très fréquentés.

Si l'image ne peut pas être chargée, le navigateur HTML affichera le texte fourni par l'attribut `alt` à la place de l'image. Par exemple :

```

```

L'attribut `alt` est également important pour l'accessibilité. Les navigateurs en mode texte et les lecteurs d'écran l'utilisent comme description de l'image correspondante.

Types d'images

Les navigateurs web peuvent afficher tous les types d'images les plus courants, tels que JPEG, PNG, GIF et SVG. Les dimensions des images sont détectées dès que les images sont chargées, mais elles peuvent être prédéfinies avec les attributs `width` et `height` :

```

```

La seule raison d'inclure des attributs de dimension à la balise `` est d'éviter de casser la mise en page lorsque l'image prend trop de temps à se charger ou lorsqu'elle ne peut pas être chargée du tout. L'utilisation des attributs `width` et `height` pour modifier les dimensions originales de l'image peut donner des résultats indésirables :

- Les images seront déformées lorsque la taille originale est inférieure aux nouvelles dimensions ou lorsque le nouveau rapport de proportion diffère de l'original.
- La réduction de la taille des grandes images utilise une bande passante supplémentaire, ce qui allonge les délais de chargement.

SVG est le seul format qui ne souffre pas de ces effets, car toutes ses informations graphiques sont stockées dans des coordonnées numériques bien adaptées à la mise à l'échelle et ses dimensions n'affectent pas la taille du fichier (d'où le nom graphique vectoriel adaptable, en anglais *Scalable Vector Graphics*). Par exemple, seules les informations relatives à la position, aux dimensions latérales et à la couleur sont nécessaires pour dessiner un rectangle en SVG. La valeur particulière de chaque pixel sera ensuite rendue de manière dynamique. En fait, les images SVG sont similaires aux fichiers HTML, dans le sens où leurs éléments graphiques sont également définis par des balises dans un fichier texte. Les fichiers SVG sont destinés à représenter des dessins aux contours nets, tels que des graphiques ou des diagrammes.

Les images qui ne répondent pas à ces critères doivent être stockées sous forme d'image matricielle *bitmaps*. Contrairement aux formats d'images vectorielles, les images matricielles stockent au préalable les informations de couleur pour chaque pixel de l'image. Le stockage de la valeur de la

couleur pour chaque pixel de l'image génère une très grande quantité de données, c'est pourquoi les images matricielles sont généralement stockés dans des formats compressés, tels que JPEG, PNG ou GIF.

Le format JPEG est recommandé pour les photographies, car son algorithme de compression donne de bons résultats pour les ombres et les arrière-plans flous. Pour les images où les couleurs unies prédominent, le format PNG est plus approprié. Par conséquent, le format PNG doit être choisi lorsqu'il est nécessaire de convertir une image vectorielle en image matricielle.

Le format GIF offre la qualité d'image la plus faible de tous les formats d'images matricielles populaires. Néanmoins, il est encore largement utilisé en raison de sa prise en charge des animations. En effet, de nombreux sites web utilisent des fichiers GIF pour afficher de courtes vidéos, mais il existe de meilleures façons d'afficher du contenu vidéo.

Audio et vidéo

Les contenus audio et vidéo peuvent être ajoutés à un document HTML plus ou moins de la même manière que les images. Sans surprise, la balise permettant d'ajouter de l'audio est `<audio>` et celle permettant d'ajouter de la vidéo est `<video>`. De toute évidence, les navigateurs textuels ne sont pas en mesure de lire les contenus multimédias, c'est pourquoi les balises `<audio>` et `<video>` utilisent la balise de fermeture pour contenir le texte utilisé comme solution de rechange pour l'élément qui ne peut être affiché. Par exemple :

```
<audio controls src="/media/recording.mp3">
<p>Unable to play <em>recording.mp3</em></p>
</audio>
```

Si le navigateur ne prend pas en charge la balise `<audio>`, la ligne "Unable to play recording.mp3" s'affiche à la place. L'utilisation des balises de fermeture `</audio>` ou `</video>` permet à une page web d'inclure un contenu alternatif plus élaboré que la simple ligne de texte autorisée par l'attribut `alt` de la balise ``.

L'attribut `src` des balises `<audio>` et `<video>` fonctionne de la même manière que pour la balise ``, mais accepte également les URL pointant vers un flux en direct. Le navigateur se charge de la mise en mémoire tampon, du décodage et de l'affichage du contenu au fur et à mesure de sa réception. L'attribut `controls` affiche les commandes de lecture. Sans cet attribut, le visiteur ne pourra pas mettre en pause, revenir en arrière ou contrôler la lecture.

Contenu générique

Un document HTML peut être imbriqué dans un autre document HTML, de la même manière que l'insertion d'une image dans un document HTML, mais en utilisant la balise `<iframe>` :

```
<iframe name="viewer" src="gallery.html">
<p>Unsupported browser</p>
</iframe>
```

Les navigateurs textuels, plus simples, ne prennent pas en charge la balise `<iframe>` et affichent le texte inclus à la place. Comme pour les balises multimédia, l'attribut `src` définit l'emplacement de la source du document imbriqué. Les attributs `width` et `height` peuvent être ajoutés pour modifier les dimensions par défaut de l'élément `iframe`.

L'attribut `name` permet de faire référence à l'`iframe` et de modifier le document imbriqué. Sans cet attribut, le document imbriqué ne peut pas être modifié. Un élément d'ancrage (`anchor`) peut être utilisé pour charger un document à partir d'un autre emplacement à l'intérieur d'un `iframe` au lieu de la fenêtre actuelle du navigateur.

Liens

L'élément de page communément appelé *lien web* est également connu sous le terme technique *ancree* (*anchor*), d'où l'utilisation de la balise `<a>`. L'ancree conduit à un autre emplacement, qui peut être n'importe quelle adresse prise en charge par le navigateur. L'emplacement est indiqué par l'attribut `href` (*hyperling reference*) :

```
<a href="contact.html">Contact Information</a>
```

L'emplacement peut être écrit sous la forme d'un chemin d'accès relatif ou absolu, comme dans le cas du contenu intégré évoqué précédemment. Seul le contenu textuel inclus (par exemple, les informations de contact) est visible pour le visiteur, généralement sous forme de texte bleu souligné cliquable par défaut, mais l'élément affiché au-dessus du lien peut également être tout autre contenu visible, comme des images :

```
<a href="contact.html"></a>
```

Des préfixes spéciaux peuvent être ajoutés à l'emplacement pour indiquer au navigateur comment l'ouvrir. Si l'ancree pointe vers une adresse électronique, par exemple, son attribut `href` doit inclure le préfixe `mailto:` :

```
<a href="mailto:info@lpi.org">Contact by email</a>
```

Le préfixe `tel:` indique un numéro de téléphone. Il est particulièrement utile pour les visiteurs qui consultent la page sur des appareils mobiles :

```
<a href="tel:+123456789">Contact by phone</a>
```

Lorsque le lien est cliqué, le navigateur ouvre le contenu de l'emplacement avec l'application associée.

L'utilisation la plus courante des ancres consiste à charger d'autres documents web. Par défaut, le navigateur remplace le document HTML actuel par le contenu du nouvel emplacement. Ce comportement peut être modifié en utilisant l'attribut `target` (cible). La cible `_blank`, par exemple, indique au navigateur d'ouvrir l'emplacement donné dans une nouvelle fenêtre ou un nouvel onglet du navigateur, selon les préférences du visiteur :

```
<a href="contact.html" target="_blank">Contact Information</a>
```

La cible `_self` est la cible par défaut lorsque l'attribut `target` n'est pas fourni. Elle fait en sorte que le document référencé remplace le document actuel.

D'autres types de cibles sont liés à l'élément `<iframe>`. Pour charger un document référencé à l'intérieur d'un élément `<iframe>`, l'attribut `target` doit pointer vers le nom de l'élément `iframe` :

```
<p><a href="gallery.html" target="viewer">Photo Gallery</a></p>

<iframe name="viewer" width="800" height="600">
<p>Unsupported browser</p>
</iframe>
```

L'élément `iframe` fonctionne comme une fenêtre de navigateur distincte, de sorte que tout lien chargé à partir du document situé dans l'`iframe` ne remplacera que le contenu de l'`iframe`. Pour modifier ce comportement, les éléments d'ancrage à l'intérieur du document d'`iframe` peuvent également utiliser l'attribut `target`. La cible `_parent`, lorsqu'elle est utilisée à l'intérieur d'un document d'`iframe`, fait en sorte que l'emplacement référencé remplace le document parent contenant la balise `<iframe>`. Par exemple, le document `gallery.html` incorporé pourrait contenir une ancre qui se charge elle-même tout en remplaçant le document parent :


```
<p><a href="gallery.html" target="_parent">Open as parent document</a></p>
```

Les documents HTML prennent en charge plusieurs niveaux d'imbrication avec la balise `<iframe>`. La cible `_top`, lorsqu'elle est utilisée dans une ancre à l'intérieur d'un document d'`iframe`, fait en sorte que l'emplacement référencé remplace le document principal dans la fenêtre du navigateur, qu'il s'agisse du parent immédiat de la `<iframe>` correspondante ou d'un ancêtre plus éloigné dans la chaîne.

Emplacements à l'intérieur des documents

L'adresse d'un document HTML peut éventuellement contenir un *fragment* qui peut être utilisé pour identifier une ressource à l'intérieur du document. Ce fragment, également connu sous le nom *ancree URL*, est une chaîne de caractères qui suit le signe dièse `#` à la fin de l'URL. Par exemple, le mot `History` est l'ancre de l'URL `https://en.wikipedia.org/wiki/Internet#History`.

Lorsque l'URL comporte une ancre, le navigateur se rendra à l'élément correspondant dans le document, c'est-à-dire l'élément dont l'attribut `id` est égal à l'ancre dans l'URL. Dans le cas de l'URL donnée, `https://en.wikipedia.org/wiki/Internet#History`, le navigateur se rendra directement à la section "History". En examinant le code HTML de la page, nous constatons que le titre de la section possède l'attribut `id` correspondant :

```
<span class="mw-headline" id="History">History</span>
```

Les ancres URL peuvent être utilisées dans l'attribut `href` de la balise `<a>`, soit lorsqu'elles pointent vers des pages externes, soit lorsqu'elles pointent vers des emplacements à l'intérieur de la page actuelle. Dans ce dernier cas, il suffit de commencer par le signe dièse avec le fragment d'URL, comme dans `History`.

WARNING

L'attribut `id` ne doit pas contenir de caractères blancs (espaces, tabulations, etc.) et doit être unique dans le document.

Il existe des moyens de personnaliser la façon dont le navigateur réagit aux ancres URL. Il est possible, par exemple, d'écrire une fonction JavaScript qui écoute l'événement de fenêtre `_hashchange_` et déclenche une action personnalisée, comme une animation ou une requête HTTP. Il convient toutefois de noter que le fragment d'URL n'est jamais envoyé au serveur avec l'URL et qu'il ne peut donc pas être utilisé comme identifiant par le serveur HTTP.

Exercices guidés

1. Le document HTML situé à l'adresse `http://www.lpi.org/articles/linux/index.html` comporte une balise `` dont l'attribut `src` pointe vers `../logo.png`. Quel est le chemin absolu complet de cette image ?

2. Citez deux raisons pour lesquelles l'attribut `alt` est important dans les balises ``.

3. Quel format d'image donne une bonne qualité d'image et maintient une petite taille de fichier lorsqu'il est utilisé pour des photos avec des points flous et avec de nombreuses couleurs et nuances ?

4. Au lieu d'utiliser un fournisseur tiers tel que Youtube, quelle balise HTML vous permet d'intégrer un fichier vidéo dans un document HTML en utilisant uniquement les fonctionnalités HTML standard ?

Exercices d'exploration

1. Supposons qu'un document HTML comporte le lien hypertexte `First picture` et l'élément `iframe` `<iframe name="gallery"></iframe>`. Comment pourriez-vous modifier la balise hyperlien pour que l'image vers laquelle elle pointe se charge à l'intérieur de l'élément `iframe` donné après que l'utilisateur ait cliqué sur le lien ?

2. Que se passe-t-il lorsque le visiteur clique sur un lien hypertexte dans un document situé à l'intérieur d'un `iframe` et que l'attribut cible du lien hypertexte est défini sur `_self` ?

3. Vous remarquez que l'ancrage URL de la deuxième section de votre page HTML ne fonctionne pas. Quelle est la cause probable de cette erreur ?

Résumé

Cette leçon explique comment ajouter des images et d'autres contenus multimédias à l'aide des balises HTML appropriées. En outre, le lecteur apprend les différentes façons d'utiliser les liens hypertextes pour charger d'autres documents et pointer vers des emplacements spécifiques dans une page. La leçon aborde les concepts et procédures suivants :

- La balise `` et ses principaux attributs : `src` et `alt`.
- Chemins URL relatifs et absolus.
- Formats d'image populaires pour le web et leurs caractéristiques.
- Les balises multimédia `<audio>` et `<video>`.
- Comment insérer des documents imbriqués avec la balise `<iframe>`.
- La balise de lien hypertexte `<a>`, son attribut `href`, et les cibles spéciales.
- Comment utiliser les fragments d'URL, également connus sous le nom d'ancres de hachage.

Réponses aux exercices guidés

1. Le document HTML situé à l'adresse `http://www.lpi.org/articles/linux/index.html` comporte une balise `` dont l'attribut `src` pointe vers `../logo.png`. Quel est le chemin absolu complet de cette image ?

`http://www.lpi.org/articles/logo.png`

2. Citez deux raisons pour lesquelles l'attribut `alt` est important dans les balises ``.

Les navigateurs en mode texte pourront afficher une description de l'image manquante. Les lecteurs d'écran utilisent l'attribut `alt` pour décrire l'image.

3. Quel format d'image donne une bonne qualité d'image et maintient une petite taille de fichier lorsqu'il est utilisé pour des photos avec des points flous et avec de nombreuses couleurs et nuances ?

Le format JPEG.

4. Au lieu d'utiliser un fournisseur tiers tel que Youtube, quelle balise HTML vous permet d'intégrer un fichier vidéo dans un document HTML en utilisant uniquement les fonctionnalités HTML standard ?

La balise `<video>`.

Réponses aux exercices d'exploration

1. Supposons qu'un document HTML comporte le lien hypertexte `Première image` et l'élément `<iframe name="gallery"></iframe>`. Comment pourriez-vous modifier la balise hyperlien pour que l'image vers laquelle elle pointe se charge à l'intérieur de l'élément `iframe` donné après que l'utilisateur ait cliqué sur le lien ?

En utilisant l'attribut `target` de la balise `a` : `Première image`.

2. Que se passe-t-il lorsque le visiteur clique sur un lien hypertexte dans un document situé à l'intérieur d'un `iframe` et que l'attribut cible du lien hypertexte est défini sur `_self` ?

Le document sera chargé dans la même `iframe`, ce qui est le comportement par défaut.

3. Vous remarquez que l'ancrage URL de la deuxième section de votre page HTML ne fonctionne pas. Quelle est la cause probable de cette erreur ?

Le fragment d'URL après le signe dièse ne correspond pas à l'attribut `id` dans l'élément correspondant à la deuxième section, ou l'attribut `id` de l'élément n'est pas présent.



**Linux
Professional
Institute**

032.4 Formulaires HTML

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 032.4

Valeur

2

Domaines de connaissance les plus importants

- Créer des formulaires HTML simples
- Comprendre les méthodes de formulaire HTML
- Comprendre les types et les éléments de saisie HTML

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `<form>`, y compris les attributs `method` (`get`, `post`), `action` et `enctype`
- `<input>`, y compris les attributs `type` (`text`, `email`, `password`, `number`, `date`, `file`, `range`, `radio`, `checkbox`, `hidden`)
- `<button>`, y compris les attributs `type` (`submit`, `reset`, `hidden`)
- `<textarea>`
- Attributs communs des éléments de formulaire (`name`, `value`, `id`)
- `<label>`, y compris l'attribut `for`



032.4 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	032 Balisage de documents HTML
Objectif :	032.4 Formulaires HTML
Leçon :	1 sur 1

Introduction

Les formulaires web constituent un moyen simple et efficace pour demander des informations aux visiteurs d'une page HTML. Le développeur web frontal (*front end*) peut utiliser divers composants tels que des champs de texte, des cases à cocher, des boutons et bien d'autres pour construire des interfaces qui enverront des données au serveur de manière structurée.

Formulaire HTML simples

Avant d'aborder le code de balisage spécifique aux formulaires, commençons par un simple document HTML vierge, sans aucun contenu :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with HTML Forms</title>
</head>
<body>
```



```
<!-- The body content goes here -->
```

```
</body>
</html>
```

Enregistrez cet exemple de code comme un fichier texte brut avec une extension `.html` (comme dans `form.html`) et utilisez votre navigateur préféré pour l'ouvrir. Après l'avoir modifié, appuyez sur le bouton de réactualisation du navigateur pour afficher les modifications.

La structure de base du formulaire est donnée par la balise `<form>` elle-même et ses éléments internes :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Working with HTML Forms</title>
</head>
<body>

<!-- Form to collect personal information -->

<form>

<h2>Personal Information</h2>

<p>Full name:</p>
<p><input type="text" name="fullname" id="fullname"></p>

<p><input type="reset" value="Clear form"></p>
<p><input type="submit" value="Submit form"></p>

</form>

</body>
</html>
```

Les doubles guillemets ne sont pas nécessaires pour les attributs à un seul mot comme `type`, donc `type=text` fonctionne aussi bien que `type="text"`. Le développeur peut choisir la convention à utiliser.

Enregistrez le nouveau contenu et rechargez la page dans le navigateur. Vous devriez voir le résultat

présenté dans la [Figure 23](#).

Personal Information

Full name:

Clear form

Submit form

Figure 23. Un formulaire très basique.

La balise `<form>` en elle-même ne produit aucun résultat notable sur la page. Les éléments contenus entre les balises `<form>...</form>` vont définir les champs et les autres éléments visuels présentés au visiteur.

Le code de l'exemple contient à la fois des balises HTML génériques (`<h2>` et `<p>`) et la balise `<input>`, qui est une balise spécifique au formulaire. Alors que les balises génériques peuvent apparaître n'importe où dans le document, les balises spécifiques au formulaire ne doivent être utilisées qu'à l'intérieur de l'élément `<form>`, c'est-à-dire entre les balises ouvrantes `<form>` et fermantes `</form>`.

NOTE

HTML ne fournit que des balises et des propriétés basiques pour modifier l'apparence standard des formulaires. CSS fournit des mécanismes avancés pour modifier l'apparence du formulaire. Il est donc recommandé d'écrire un code HTML qui ne traite que des aspects fonctionnels du formulaire et de modifier son apparence avec CSS.

Comme le montre l'exemple, la balise de paragraphe `<p>` peut être utilisée pour décrire le champ au visiteur. Cependant, il n'y a pas de moyen évident pour le navigateur de faire le lien entre la description dans la balise `<p>` et l'élément d'entrée correspondant. La balise étiquette `<label>` est plus appropriée dans ces cas (à partir de maintenant, considérez que tous les exemples de code se trouvent dans le corps du document HTML) :

```

<form>

<h2>Personal Information</h2>

<label for="fullname">Full name:</label>
<p><input type="text" name="fullname" id="fullname"></p>

<p><input type="reset" value="Clear form"></p>
<p><input type="submit" value="Submit form"></p>

</form>

```

L'attribut `for` de la balise `<label>` contient l'identifiant `id` de l'élément d'entrée correspondant. Cela rend la page plus accessible, car les lecteurs d'écran pourront lire le contenu de l'élément `<label>` lorsque l'élément d'entrée est au premier plan. En outre, les visiteurs peuvent cliquer sur l'étiquette pour mettre l'accent sur le champ d'entrée correspondant.

L'attribut `id` fonctionne pour les éléments de formulaire comme pour tout autre élément du document. Il fournit un identifiant pour l'élément qui est unique dans tout le document. L'attribut `name` a un but similaire, mais il est utilisé pour identifier l'élément d'entrée dans le contexte du formulaire. Le navigateur utilise l'attribut `name` pour identifier le champ de saisie lors de l'envoi des données du formulaire au serveur. Il est donc important d'utiliser des attributs `name` significatifs et uniques dans le formulaire.

L'attribut `type` est le principal attribut de l'élément d'entrée, car il contrôle le type de données que l'élément accepte et sa présentation visuelle au visiteur. Si l'attribut `type` n'est pas fourni, l'entrée présente par défaut un champ de texte. Les types d'entrée suivants sont pris en charge par les navigateurs modernes :

Table 1. Types d'entrées de formulaire

Attribut de type	Type de données	Comment il est affiché
<code>hidden</code>	Une chaîne arbitraire	N/A
<code>text</code>	Texte sans retour à la ligne	Un champ de texte
<code>search</code>	Texte sans retour à la ligne	Un champ de recherche
<code>tel</code>	Texte sans retour à la ligne	Un champ de texte
<code>url</code>	Une URL absolue	Un champ de texte
<code>email</code>	Une adresse e-mail ou une liste d'adresses e-mail	Un champ de texte

Attribut de type	Type de données	Comment il est affiché
<code>password</code>	Un texte sans retour à la ligne (informations sensibles)	Un champ de texte qui masque la saisie des données
<code>date</code>	Une date (année, mois, jour) sans fuseau horaire	Un champ de contrôle de date
<code>month</code>	Une date composée d'une année et d'un mois sans fuseau horaire	Un champ de contrôle du mois
<code>week</code>	Une date composée d'un numéro de semaine et d'un numéro d'année sans fuseau horaire	Un champ de contrôle de semaine
<code>time</code>	Un temps (heure, minute, seconde, fraction de seconde) sans fuseau horaire	Un champ de contrôle de l'heure
<code>datetime-local</code>	Une date et un temps (année, mois, jour, heure, minute, seconde, fraction de seconde) sans fuseau horaire	Un champ de contrôle de date et d'heure
<code>number</code>	Une valeur numérique	Un champ de texte ou un champ avec bouton fléché
<code>range</code>	Une valeur numérique, avec la particularité sémantique que la valeur exacte n'est pas importante	Une glissière ou similaire
<code>color</code>	Une couleur sRGB avec des composantes rouge, verte et bleue de 8 bits	Un sélecteur de couleur
<code>checkbox</code>	Un ensemble de zéro ou de plusieurs valeurs provenant d'une liste prédéfinie	Une case à cocher (offre des choix et permet de sélectionner plusieurs choix)
<code>radio</code>	Une valeur énumérée	Un bouton radio (offre des choix et ne permet de sélectionner qu'un seul choix)
<code>file</code>	Zéro ou plusieurs fichiers, chacun avec un type MIME et un nom de fichier facultatif	Une étiquette et un bouton

Attribut de type	Type de données	Comment il est affiché
submit	Une valeur énumérée, qui met fin au processus de saisie et entraîne la soumission du formulaire	Un bouton
image	Une coordonnée, relative à la taille d'une image particulière, qui met fin au processus de saisie et provoque l'envoi du formulaire	Soit une image cliquable, soit un bouton
button	N/A	Un bouton générique
reset	N/A	Un bouton dont la fonction est de réinitialiser tous les autres champs à leur valeur initiale

L'apparence des types d'entrée `password`, `search`, `tel`, `url`, et `email` ne diffère pas du type standard `text`. Leur but est d'offrir des indications au navigateur sur le contenu prévu pour ce champ d'entrée, afin que le navigateur ou le script s'exécutant du côté client puisse prendre des mesures personnalisées pour un type d'entrée spécifique. La seule différence entre le type d'entrée de texte et le type de champ de mot de passe, par exemple, est que le contenu du champ de mot de passe n'est pas affiché lorsque le visiteur le saisit. Sur les appareils à écran tactile, où le texte est tapé à l'aide d'un clavier à l'écran, le navigateur peut faire apparaître uniquement le clavier numérique lorsqu'une entrée de type `tel` est sélectionnée. Une autre action possible consiste à proposer une liste d'adresses électroniques connues lorsqu'une entrée de type `email` est sélectionnée.

Le type `number` apparaît également comme une simple entrée de texte, mais avec des flèches d'incrément/décroissement sur le côté. Son utilisation fera en sorte que le clavier numérique s'affiche sur les écrans tactiles lorsqu'il a le focus.

Les autres éléments d'entrée ont leur propre apparence et leur propre comportement. Le type `date`, par exemple, est affiché selon les paramètres du format de date local et un calendrier s'affiche lorsque le champ devient actif :

```
<form>

<p>
  <label for="date">Date:</label>
  <input type="date" name="date" id="date">
</p>
```

```
</form>
```

La [Figure 24](#) montre comment la version de bureau de Firefox affiche actuellement ce champ.

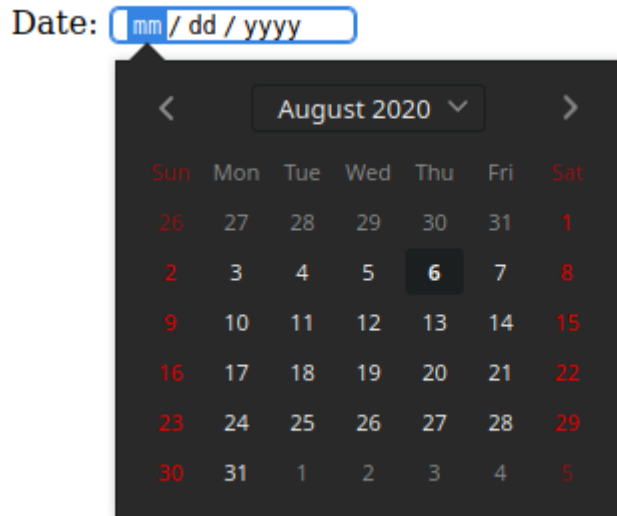


Figure 24. Le type d'entrée date.

NOTE

Certains éléments peuvent apparaître légèrement différents dans les différents navigateurs ou systèmes d'exploitation, mais leur fonctionnement et leur utilisation sont toujours les mêmes.

Il s'agit d'une fonctionnalité standard de tous les navigateurs modernes, qui ne nécessite aucune option ou programmation supplémentaire.

Quel que soit le type d'entrée, le contenu d'un champ de saisie est appelé sa *valeur*. Toutes les valeurs de champ sont vides par défaut, mais l'attribut `value` peut être utilisé pour définir une valeur par défaut pour le champ. La valeur du type date doit utiliser le format `AAAA-MM-jj`. La valeur par défaut du champ date suivant est le 6 août 2020 :

```
<form>

<p>
  <label for="date">Date:</label>
  <input type="date" name="date" id="date" value="2020-08-06">
</p>
```

```
</form>
```

Les types d'entrée spécifiques aident le visiteur à remplir les champs, mais n'empêchent pas le visiteur de contourner les restrictions et de saisir des valeurs arbitraires dans n'importe quel champ. C'est pourquoi il est important que les valeurs des champs soient validées lorsqu'elles arrivent sur le serveur.

Les éléments de formulaire dont les valeurs doivent être saisies par le visiteur peuvent avoir des attributs spéciaux qui aident à les remplir. L'attribut `placeholder` insère une valeur d'exemple dans l'élément de saisie :

```
<p>Address: <input type="text" name="address" id="address" placeholder="e.g. 41 John St., Upper Suite 1"></p>
```

La valeur d'exemple apparaît à l'intérieur de l'élément de saisie, comme le montre la [Figure 25](#).



Address: e.g. 41 John St., Upper Suite 1

Figure 25. Exemple de l'attribut `placeholder`.

Dès que le visiteur commence à taper dans le champ, le texte d'exemple disparaît. Le texte d'exemple n'est pas envoyé comme valeur du champ si le visiteur laisse le champ vide.

L'attribut `required` oblige le visiteur à saisir une valeur dans le champ correspondant avant de soumettre le formulaire :

```
<p>Address: <input type="text" name="address" id="address" required placeholder="e.g. 41 John St., Upper Suite 1"></p>
```

L'attribut `required` est un attribut booléen, il peut donc être placé seul (sans le signe égal). Il est important de marquer les champs qui sont obligatoires, sinon le visiteur ne sera pas en mesure de dire quels champs manquent et empêchent la soumission du formulaire.

L'attribut `autocomplete` indique si la valeur de l'élément d'entrée peut être complétée automatiquement par le navigateur. S'il a pour valeur `autocomplete="off"`, le navigateur ne suggère pas de valeurs antérieures pour remplir l'entrée. Les éléments d'entrée d'informations sensibles, comme les numéros de carte de crédit, doivent avoir l'attribut `auto-complete` défini sur `off`.

Saisie de textes volumineux : `textarea`

Contrairement au champ de texte, dans lequel on ne peut insérer qu'une seule ligne de texte, l'élément `textarea` permet au visiteur d'entrer plus d'une ligne de texte. L'élément `textarea` est un élément distinct, il n'est pas basé sur l'élément `input` :

```
<p> <label for="comment">Type your comment here:</label> <br>  
  
<textarea id="comment" name="comment" rows="10" cols="50">  
My multi-line, plain-text comment.  
</textarea>  
  
</p>
```

La [Figure 26](#) montre l'apparence typique d'une zone de texte `textarea`.



Figure 26. L'élément `textarea`.

Une autre différence avec l'élément `input` est que l'élément `textarea` a une balise de fermeture (`</textarea>`), donc son contenu (c'est-à-dire sa valeur) se place entre les deux balises. Les attributs `rows` et `cols` ne limitent pas la quantité de texte ; ils servent uniquement à définir la disposition. La zone de texte possède également une poignée dans le coin inférieur droit, qui permet au visiteur de la redimensionner.

Listes d'options

Plusieurs types de champs de formulaire peuvent être utilisés pour présenter une liste d'options au visiteur : l'élément `<select>` et les types d'entrée `radio` et `checkbox`.

L'élément `<select>` est un champ de contrôle de liste déroulante avec des entrées prédéfinies :

```
<p><label for="browser">Favorite Browser:</label>
<select name="browser" id="browser">
  <option value="firefox">Mozilla Firefox</option>
  <option value="chrome">Google Chrome</option>
  <option value="opera">Opera</option>
  <option value="edge">Microsoft Edge</option>
</select>
</p>
```

La balise `<option>` représente une seule entrée dans l'élément `<select>` correspondant. La liste entière apparaît lorsque le visiteur tape ou clique sur le champ, comme le montre la [Figure 27](#).

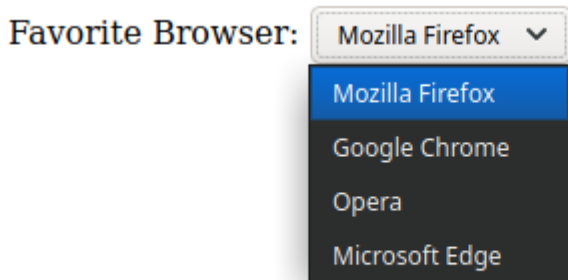


Figure 27. L'élément de formulaire `select`.

La première entrée de la liste est sélectionnée par défaut. Pour modifier ce comportement, vous pouvez ajouter l'attribut `selected` à une autre entrée afin qu'elle soit sélectionnée au chargement de la page.

Le type d'entrée `radio` est similaire à l'élément `<select>`, mais au lieu d'afficher une liste déroulante, il affiche toutes les entrées afin que le visiteur puisse en sélectionner une. Les résultats du code suivant sont présentés dans la [Figure 28](#).

```
<p>Favorite Browser:</p>

<p>
```

```
<input type="radio" id="browser-firefox" name="browser" value="firefox"
checked>
<label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
  <input type="radio" id="browser-chrome" name="browser" value="chrome">
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
  <input type="radio" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>

<p>
  <input type="radio" id="browser-edge" name="browser" value="edge">
  <label for="browser-edge">Microsoft Edge</label>
</p>
```

Favorite Browser:

- ☒ Mozilla Firefox
- ☐ Google Chrome
- ☐ Opera
- ☐ Microsoft Edge

Figure 28. Éléments d'entrée de type `radio`.

Notez que tous les types d'entrée `radio` du même groupe ont le même attribut `name`. Chacun d'eux est exclusif, donc l'attribut `value` correspondant à l'entrée choisie sera celui associé à l'attribut `name` partagé. L'attribut `checked` fonctionne comme l'attribut `selected` de l'élément `<select>`. Il marque l'entrée correspondante lorsque la page se charge pour la première fois. La balise `<label>` est particulièrement utile pour les entrées `radio`, car elle permet au visiteur de vérifier une entrée en cliquant ou en tapant sur le texte correspondant en plus du champ lui-même.

Alors que les champs de contrôle `radio` sont destinés à ne sélectionner qu'une seule entrée d'une liste, le type d'entrée `checkbox` permet au visiteur de sélectionner plusieurs entrées :

```

<p>Favorite Browser:</p>

<p>
  <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
  <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
  <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
  <input type="checkbox" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>

<p>
  <input type="checkbox" id="browser-edge" name="browser" value="edge">
  <label for="browser-edge">Microsoft Edge</label>
</p>

```

Les cases à cocher peuvent également utiliser l'attribut `checked` pour que les entrées soient sélectionnées par défaut. Au lieu des champs ronds de l'entrée radio, les cases à cocher sont affichées comme des champs carrés, comme le montre la [Figure 29](#).

Favorite Browser:

- ☒ Mozilla Firefox
- ☒ Google Chrome
- ☐ Opera
- ☐ Microsoft Edge

Figure 29. Éléments d'entrée de type `checkbox`.

Si plusieurs entrées sont sélectionnées, le navigateur les soumettra avec le même nom, ce qui oblige le développeur back-end à écrire un code spécifique pour lire correctement les données de

formulaires contenant des cases à cocher.

Pour une meilleure utilisation, les champs d'entrée peuvent être regroupés à l'intérieur d'une balise `<fieldset>` :

```
<fieldset>
<legend>Favorite Browser</legend>

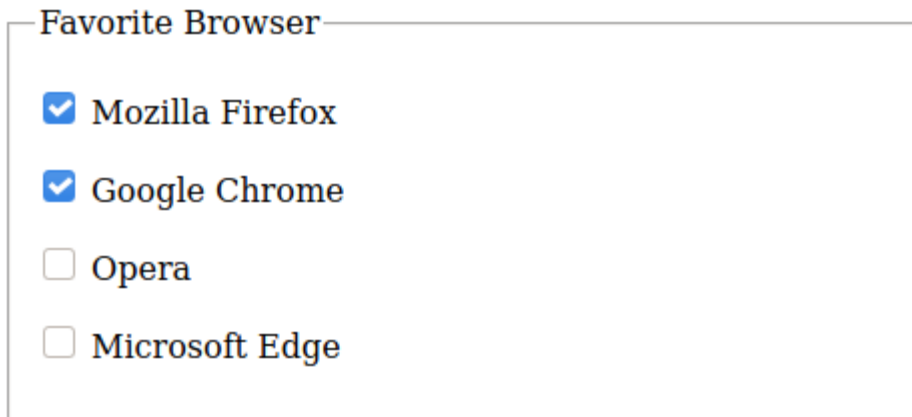
<p>
  <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
  <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
  <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
  <input type="checkbox" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>

<p>
  <input type="checkbox" id="browser-edge" name="browser" value="edge">
  <label for="browser-edge">Microsoft Edge</label>
</p>
</fieldset>
```

La balise `<legend>` contient le texte qui est placé en haut du cadre que la balise `<fieldset>` dessine autour des champs ([Figure 30](#)).



Favorite Browser

☒ Mozilla Firefox

☒ Google Chrome

☐ Opera

☐ Microsoft Edge

Figure 30. Regroupement d'éléments avec la balise `fieldset`.

La balise `<fieldset>` ne change pas la façon dont les valeurs des champs sont soumises au serveur, mais elle permet au développeur web frontal (*frontend*) de contrôler plus facilement les champs imbriqués. Par exemple, en définissant l'attribut `disabled` dans un attribut `<fieldset>`, tous ses éléments internes seront rendus indisponibles pour le visiteur.

Type d'élément `hidden`

Il existe des situations où le développeur souhaite inclure dans le formulaire des informations qui ne peuvent pas être manipulées par le visiteur, c'est-à-dire soumettre une valeur choisie par le développeur sans présenter un champ de formulaire où le visiteur peut taper ou modifier la valeur. Le développeur peut vouloir, par exemple, inclure un jeton d'identification pour ce formulaire particulier qui ne doit pas être vu par le visiteur. Un élément de formulaire `hidden` (caché) est codé comme dans l'exemple suivant :

```
<input type="hidden" id="form-token" name="form-token" value="e730a375-b953-4393-847d-2dab065bbc92">
```

La valeur d'un champ d'entrée caché est généralement ajoutée au document côté serveur, lors de l'affichage du document. Les entrées cachées sont traitées comme des champs ordinaires lorsque le navigateur les envoie au serveur, qui les lit également comme des champs d'entrée ordinaires.

Type d'entrée `file`

En plus des données textuelles, saisies ou sélectionnées dans une liste, les formulaires HTML peuvent également soumettre des fichiers arbitraires au serveur. Le type d'entrée `file` permet au visiteur de choisir un fichier dans le système de fichiers local et de l'envoyer directement depuis la page web :

```
<p>
<label for="attachment">Attachment:</label><br>
<input type="file" id="attachment" name="attachment">
</p>
```

Au lieu d'un champ de formulaire pour écrire ou sélectionner une valeur, le type d'entrée `file` montre un bouton `browse` (parcourir) qui ouvrira une boîte de dialogue de fichier. Tout type de fichier est accepté par le type d'entrée `file`, mais le développeur backend limitera probablement les types de fichiers autorisés et leur taille maximale. La vérification du type de fichier peut également être effectuée dans le front-end en ajoutant l'attribut `accept`. Par exemple, pour accepter uniquement les images JPEG et PNG l'attribut `accept` doit être `accept="image/jpeg, image/png"`.

Boutons d'action

Par défaut, le formulaire est soumis lorsque le visiteur appuie sur la touche Entrée dans n'importe quel champ de saisie. Pour rendre les choses plus intuitives, il faut ajouter un bouton d'envoi avec le type d'entrée `submit` :

```
<input type="submit" value="Submit form">
```

Le texte contenu dans l'attribut `value` s'affiche sur le bouton, comme indiqué dans la [Figure 31](#).

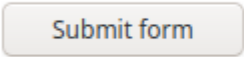


Figure 31. Un bouton de soumission standard.

Un autre bouton utile à inclure dans les formulaires complexes est le bouton `reset`, qui efface le formulaire et le remet dans son état initial :

```
<input type="reset" value="Clear form">
```

Comme pour le bouton `submit`, le texte de l'attribut `value` est utilisé pour étiqueter le bouton. Alternativement la balise `<button>` peut être utilisée pour ajouter des boutons aux formulaires ou à tout autre endroit de la page. Contrairement aux boutons réalisés avec la balise `<input>`, l'élément bouton possède une balise de fermeture et l'étiquette du bouton est son contenu interne :

```
<button>Submit form</button>
```

À l'intérieur d'un formulaire, l'action par défaut de l'élément `button` est de soumettre le formulaire. Comme pour les boutons d'entrée, l'attribut `type` du bouton peut être changé à `reset`.

Action et méthodes de formulaire

La dernière étape de l'écriture d'un formulaire HTML consiste à définir comment et où les données doivent être envoyées. Ces aspects dépendent des détails du client et du serveur.

Côté serveur, l'approche la plus courante consiste à disposer d'un fichier script qui analysera, validera et traitera les données du formulaire en fonction de l'objectif de l'application. Par exemple, le développeur backend pourrait écrire un script appelé `receive_form.php` pour recevoir les données envoyées par le formulaire. Du côté client, le script est indiqué dans l'attribut `action` de la balise de formulaire :

```
<form action="receive_form.php">
```

L'attribut `action` suit les mêmes conventions que toutes les adresses HTTP. Si le script se trouve au même niveau hiérarchique que la page contenant le formulaire, il peut être écrit sans son chemin. Sinon, le chemin absolu ou relatif doit être fourni. Le script doit également générer la réponse qui servira de page de destination, chargée par le navigateur après que le visiteur a soumis le formulaire.

HTTP fournit des méthodes distinctes pour envoyer les données d'un formulaire via une connexion avec le serveur. Les méthodes les plus courantes sont `get` et `post`, qui doivent être indiquées dans l'attribut `method` de la balise `form` :

```
<form action="receive_form.php" method="get">
```

Ou :

```
<form action="receive_form.php" method="post">
```

Dans la méthode `get`, les données du formulaire sont encodées directement dans l'URL de la requête. Lorsque le visiteur soumet le formulaire, le navigateur charge l'URL définie dans l'attribut `action` avec les champs du formulaire ajoutés.

La méthode `get` est préférable pour les petites quantités de données, comme les simples formulaires de contact. Cependant, l'URL ne peut pas dépasser quelques milliers de caractères. La méthode `post` doit donc être utilisée lorsque les formulaires contiennent des champs importants ou non textuels, comme des images.

La méthode `post` permet au navigateur d'envoyer les données du formulaire dans la section corps de la requête HTTP. Bien qu'elle soit nécessaire pour les données binaires qui dépassent la limite de taille d'une URL, la méthode `post` ajoute une surcharge inutile à la connexion lorsqu'elle est utilisée pour des formulaires textuels plus simples ; la méthode `get` est donc préférable dans ces cas.

La méthode choisie n'affecte pas la manière dont le visiteur interagit avec le formulaire. Les méthodes `get` et `post` sont traitées différemment par le script côté serveur qui reçoit le formulaire.

Lorsque vous utilisez la méthode `post`, il est également possible de modifier le type MIME du contenu du formulaire avec l'attribut de formulaire `enctype`. Cela affecte la façon dont les champs et les valeurs du formulaire seront empilés ensemble dans la communication HTTP avec le serveur. La valeur par défaut de `enctype` est `application/x-www-form-urlencoded`, ce qui est similaire au format utilisé par la méthode `get`. Si le formulaire contient des champs de saisie de type `file`, l'`enctype` `multipart/form-data` doit être utilisé à la place.

Exercices guidés

1. Quelle est la bonne approche pour associer une balise `<label>` à une balise `<input>` ?

2. Quel type d'élément d'entrée fournit une glissière pour choisir une valeur numérique ?

3. Quel est l'objectif de l'attribut de formulaire `placeholder` ?

4. Comment pouvez-vous faire en sorte que la deuxième option d'un élément `<select>` soit sélectionnée par défaut ?

Exercices d'exploration

1. Comment pouvez-vous modifier une entrée `file` pour qu'elle n'accepte que les fichiers PDF ?
2. Comment pouvez-vous informer le visiteur des champs obligatoires d'un formulaire ?
3. Réunissez trois extraits de code de cette leçon dans un seul formulaire. Veillez à ne pas utiliser le même attribut `name` ou `id` dans plusieurs champs de formulaire.

Résumé

Cette leçon explique comment créer des formulaires HTML simples pour renvoyer des données au serveur. Du côté client, les formulaires HTML sont constitués d'éléments HTML standard qui sont combinés pour construire des interfaces personnalisées. De plus, les formulaires doivent être configurés pour communiquer correctement avec le serveur. La leçon aborde les concepts et procédures suivants :

- La balise `<form>` et la structure de base du formulaire.
- Les éléments d'entrée de base et spéciaux.
- Le rôle des balises spéciales comme `<label>`, `<fieldset>` et `<legend>`.
- Les boutons et les actions du formulaire.

Réponses aux exercices guidés

1. Quelle est la bonne approche pour associer une balise `<label>` à une balise `<input>` ?

L'attribut `for` de la balise `<label>` doit contenir l'id de la balise `<input>` correspondante.

2. Quel type d'élément d'entrée fournit une glissière pour choisir une valeur numérique ?

Le type d'entrée `range`.

3. Quel est l'objectif de l'attribut de formulaire `placeholder` ?

L'attribut `placeholder` contient un exemple de saisie qui est visible lorsque le champ d'entrée correspondant est vide.

4. Comment pouvez-vous faire en sorte que la deuxième option d'un élément `<select>` soit sélectionnée par défaut ?

Le deuxième élément `option` doit avoir l'attribut `selected`.

Réponses aux exercices d'exploration

1. Comment pouvez-vous modifier une entrée `file` pour qu'elle n'accepte que les fichiers PDF ?

L'attribut `accept` de l'élément d'entrée doit prendre la valeur `application/pdf`.

2. Comment pouvez-vous informer le visiteur des champs obligatoires d'un formulaire ?

En général, les champs obligatoires sont marqués d'un astérisque (*), et une brève note du type "Les champs marqués d'un * sont obligatoires" est placée à proximité du formulaire.

3. Réunissez trois extraits de code de cette leçon dans un seul formulaire. Veillez à ne pas utiliser le même attribut `name` ou `id` dans plusieurs champs de formulaire.

```
<form action="receive_form.php" method="get">

<h2>Personal Information</h2>

<label for="fullname">Full name:</label>
<p><input type="text" name="fullname" id="fullname"></p>

<p>
  <label for="date">Date:</label>
  <input type="date" name="date" id="date">
</p>

<p>Favorite Browser:</p>

<p>
  <input type="checkbox" id="browser-firefox" name="browser" value="firefox"
checked>
  <label for="browser-firefox">Mozilla Firefox</label>
</p>

<p>
  <input type="checkbox" id="browser-chrome" name="browser" value="chrome"
checked>
  <label for="browser-chrome">Google Chrome</label>
</p>

<p>
  <input type="checkbox" id="browser-opera" name="browser" value="opera">
  <label for="browser-opera">Opera</label>
</p>
```

```
<p>  
  <input type="checkbox" id="browser-edge" name="browser" value="edge">  
  <label for="browser-edge">Microsoft Edge</label>  
</p>  
  
<p><input type="reset" value="Clear form"></p>  
<p><input type="submit" value="Submit form"></p>  
  
</form>
```



**Linux
Professional
Institute**

Thème 033: Styles de contenu CSS



033.1 Bases du CSS

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 033.1

Valeur

1

Domaines de connaissance les plus importants

- Intégrer le CSS dans un document HTML
- Comprendre la syntaxe CSS
- Ajouter des commentaires aux CSS
- Connaître les caractéristiques et les exigences en matière d'accessibilité

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- Attributs HTML `style` et `type (text/css)`
- `<style>`
- `<link>`, y compris les attributs `rel (stylesheet)`, `type (text/css)` et `src`
- `;`
- `/s/`



033.1 Leçon 1

Certification:	Web Development Essentials
Version:	1.0
Thème:	033 Styliser le contenu avec CSS
Objectif:	033.1 Bases du CSS
Leçon:	1 sur 1

Introduction

Tous les navigateurs web affichent les pages HTML en utilisant des règles de présentation par défaut qui sont pratiques et directes, mais pas attrayantes visuellement. Le HTML en lui-même offre peu de fonctionnalités pour écrire des pages élaborées basées sur des concepts modernes d'expérience utilisateur. Après avoir écrit des pages HTML simples, vous avez probablement réalisé qu'elles sont inesthétiques par rapport aux pages bien conçues que l'on trouve sur Internet. En effet, dans le langage HTML moderne, le code de balisage destiné à la structure et à la fonction des éléments du document (c'est-à-dire le *contenu sémantique*) est séparé des règles qui définissent l'aspect des éléments (la *présentation*). Les règles de présentation sont écrites dans un langage différent appelé *feuilles de style en cascade* (CSS : *Cascading Style Sheets*). Il vous permet de modifier presque tous les aspects visuels du document, tels que les polices, les couleurs et l'emplacement des éléments sur la page.

Les documents HTML ne sont pas, dans la plupart des cas, destinés à être affichés dans une mise en page fixe comme dans un fichier PDF. Au contraire, les pages web basées sur HTML seront probablement affichées sur une grande variété de tailles d'écran, voire imprimées. CSS fournit des options réglables pour garantir que la page web sera affichée comme prévu, en fonction du périphérique ou de l'application qui l'ouvre.

Appliquer des styles

Il existe plusieurs manières d'inclure le CSS dans un document HTML : l'écrire directement dans la balise de l'élément, dans une section distincte du code source de la page, ou dans un fichier externe qui sera référencé par la page HTML.

Attribut `style`

La méthode la plus simple pour modifier le style d'un élément spécifique est de l'écrire directement dans la balise de l'élément en utilisant l'attribut `style`. Tous les éléments HTML visibles autorisent un attribut `style`, dont la valeur peut être une ou plusieurs règles CSS, également appelées *propriétés*. Commençons par un exemple simple, un élément de type paragraphe :

```
<p>My stylized paragraph</p>
```

La syntaxe de base d'une propriété CSS personnalisée est `property : value`, où `property` est l'aspect particulier que vous voulez modifier dans l'élément et `value` définit le remplacement de l'option par défaut faite par le navigateur. Certaines propriétés s'appliquent à tous les éléments et d'autres ne s'appliquent qu'à des éléments spécifiques. De même, il existe des valeurs appropriées à utiliser dans chaque propriété.

Pour changer la couleur de notre paragraphe simple, par exemple, nous utilisons la propriété `color`. La propriété `color` fait référence à la couleur d'avant-plan, c'est-à-dire à la couleur des lettres du paragraphe. La couleur elle-même est placée dans la partie valeur de la règle et peut être spécifiée dans de nombreux formats différents, y compris des noms simples comme `red` (rouge), `green` (vert), `blue` (bleu), `yellow` (jaune), etc. Ainsi, pour rendre les lettres du paragraphe violettes (`purple`), ajoutez la propriété personnalisée `color : purple` à l'attribut `style` de l'élément :

```
<p style="color: purple">My first stylized paragraph</p>
```

D'autres propriétés personnalisées peuvent être incluses dans la même propriété `style`, mais elles doivent être séparées par des points-virgules. Si vous voulez que la taille de la police soit plus grande, par exemple, ajoutez `font-size : larger` à la propriété `style` :

```
<p style="color: purple; font-size: larger">My first stylized paragraph</p>
```

NOTE

Il n'est pas nécessaire d'ajouter des espaces autour des deux points et des points-virgules, mais ils peuvent faciliter la lecture du code CSS.

Pour voir le résultat de ces modifications, enregistrez le code HTML suivant dans un fichier, puis ouvrez-le dans un navigateur web (les résultats sont montrés dans la [Figure 32](#)) :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>
</head>
<body>

<p style="color: purple; font-size: larger">My first stylized paragraph</p>

<p style="color: green; font-size: larger">My second stylized paragraph</p>

</body>
</html>
```

My first stylized paragraph

My second stylized paragraph

Figure 32. Un changement visuel très simple à l'aide de CSS.

Vous pouvez imaginer chaque élément de la page comme un rectangle ou une boîte dont vous pouvez manipuler les propriétés géométriques et les décorations avec CSS. Le mécanisme de rendu utilise deux concepts standard de base pour le placement des éléments : le placement *bloc* et le placement *en ligne*. Les éléments de type bloc occupent tout l'espace horizontal de leur élément parent et sont placés de manière séquentielle, de haut en bas. Leur hauteur (leur *dimension verticale*, à ne pas confondre avec leur position *haute* dans la page) dépend généralement de la quantité de contenu qu'ils contiennent. Les éléments en ligne suivent la méthode de gauche à droite similaire à celle des langues écrites occidentales : les éléments sont placés horizontalement, de gauche à droite, jusqu'à ce qu'il n'y ait plus de place sur le côté droit, après quoi l'élément continue sur une nouvelle ligne juste en dessous de la ligne actuelle. Les éléments tels que `p`, `div`, et `section` sont placés en blocs par défaut, alors que les éléments tels que `span`, `em`, `a`, et les lettres simples sont placés en ligne. Ces méthodes de placement de base peuvent être fondamentalement modifiées par les règles CSS.

Le rectangle correspondant à l'élément `p` dans le corps du document HTML de l'exemple sera visible si vous ajoutez la propriété `background-color` à la règle ([Figure 33](#)) :

```
<p style="color: purple; font-size: larger; background-color: silver">My first  
stylized paragraph</p>
```

```
<p style="color: green; font-size: larger; background-color: silver">My second  
stylized paragraph</p>
```

My first stylized paragraph

My second stylized paragraph

Figure 33. Rectangles correspondant aux paragraphes.

Au fur et à mesure que vous ajoutez de nouvelles propriétés CSS personnalisées à l'attribut `style`, vous remarquerez que le code global commence à être encombré. L'écriture d'un trop grand nombre de règles CSS dans l'attribut `style` compromet la séparation de la structure (HTML) et de la présentation (CSS). En outre, vous vous rendrez vite compte que de nombreux styles sont partagés entre différents éléments et qu'il n'est pas judicieux de les répéter dans chaque élément.

Règles CSS

Plutôt que de modifier le style des éléments directement dans leurs attributs `style`, il est beaucoup plus pratique d'indiquer au navigateur l'ensemble des éléments auxquels s'applique le style personnalisé. Pour ce faire, nous ajoutons un *sélecteur* (*selector*) aux propriétés personnalisées, en faisant correspondre les éléments par type, classe, ID unique, position relative, etc. La combinaison d'un *sélecteur* et des propriétés personnalisées correspondantes—également appelées *déclarations*—s'appelle une *règle CSS*. La syntaxe de base d'une règle CSS est `selector { property: value }`. Comme dans l'attribut `style`, les propriétés séparées par des points-virgules peuvent être regroupées, comme dans `p { color : purple ; font-size : larger }`. Cette règle correspond à chaque élément `p` de la page et applique les propriétés personnalisées `color` et `font-size`.

Une règle CSS pour un élément parent correspondra automatiquement à tous ses éléments enfants. Cela signifie que nous pourrions appliquer les propriétés personnalisées à tout le texte de la page, qu'il soit à l'intérieur ou à l'extérieur d'une balise `<p>`, en utilisant le sélecteur `body` à la place : `body { color : purple ; font-size : larger }`. Cette stratégie nous évite de réécrire la même règle pour tous ses enfants, mais il peut être nécessaire d'écrire des règles supplémentaires pour "annuler" ou pour modifier les règles héritées.

Balise `style`

La balise `<style>` nous permet d'écrire des règles CSS à l'intérieur de la page HTML que nous voulons styliser. Elle permet au navigateur de différencier le code CSS du code HTML. La balise `<style>` se place dans la section `head` du document :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <style type="text/css">

    p { color: purple; font-size: larger }

  </style>

</head>
<body>

<p>My first stylized paragraph</p>

<p>My second stylized paragraph</p>

</body>
</html>
```

L'attribut `type` indique au navigateur quel type de contenu se trouve à l'intérieur de la balise `<style>`, c'est-à-dire son *type MIME*. Comme tous les navigateurs qui prennent en charge CSS supposent que le type de la balise `<style>` est `text/css` par défaut, l'inclusion de l'attribut `type` est facultative. Il existe également un attribut `media` qui indique le support—écrans d'ordinateur ou impression, par exemple—auquel les règles CSS de la balise `<style>` s'appliquent. Par défaut, les règles CSS s'appliquent à tout support sur lequel le document est rendu.

Comme dans le code HTML, les sauts de ligne et l'indentation du code ne modifient pas la manière dont les règles CSS sont interprétées par le navigateur. L'écriture :

```
<style type="text/css">

p { color: purple; font-size: larger }
```

```
</style>
```

a exactement le même résultat que l'écriture :

```
<style type="text/css">

p {
  color: purple;
  font-size: larger;
}

</style>
```

Les octets supplémentaires utilisés par les espaces et les sauts de ligne font peu de différence dans la taille finale du document et n'ont pas d'impact significatif sur le temps de chargement de la page, le choix de la mise en page est donc une question de goût. Notez le point-virgule après la dernière déclaration (`font-size : larger;`) de la règle CSS. Ce point-virgule n'est pas obligatoire, mais sa présence facilite l'ajout d'une autre déclaration dans la ligne suivante sans se soucier des points-virgules manquants.

Le fait d'avoir les déclarations sur des lignes séparées est également utile lorsque vous devez commenter une déclaration. Lorsque vous souhaitez désactiver temporairement une déclaration pour des raisons de débogage, par exemple, vous pouvez entourer la ligne correspondante avec `/*` et `*/` :

```
p {
  color: purple;
  /*
  font-size: larger;
  */
}
```

ou :

```
p {
  color: purple;
  /* font-size: larger; */
}
```

Si elle est écrite de cette façon, la déclaration `font-size : larger` sera ignorée par le navigateur.

Veillez à ouvrir et fermer correctement les commentaires, sinon le navigateur risque de ne pas être en mesure d'interpréter les règles.

Les commentaires sont également utiles pour écrire des rappels sur les règles :

```
/* Texts inside <p> are purple and larger */  
p {  
  color: purple;  
  font-size: larger;  
}
```

Les rappels comme celui de l'exemple sont inutiles dans les feuilles de style contenant un petit nombre de règles, mais ils sont essentiels pour aider à naviguer dans les feuilles de style contenant des centaines (ou plus) de règles.

Même si l'attribut `style` et la balise `<style>` sont adéquats pour tester des styles personnalisés et utiles dans des situations spécifiques, ils ne sont pas utilisés couramment. Au lieu de cela, les règles CSS sont généralement placées dans un fichier séparé qui peut être référencé à partir du document HTML.

CSS dans des fichiers externes

La méthode recommandée pour associer des définitions CSS à un document HTML consiste à stocker le CSS dans un fichier séparé. Cette méthode offre deux avantages principaux par rapport aux précédentes :

- Les mêmes règles de style peuvent être partagées entre plusieurs documents distincts.
- Le fichier CSS est généralement mis en cache par le navigateur, ce qui améliore les temps de chargement futurs.

Les fichiers CSS ont l'extension `.css` et, comme les fichiers HTML, ils peuvent être édités par n'importe quel éditeur de texte simple. Contrairement aux fichiers HTML, les fichiers CSS n'ont pas de structure de haut niveau construite avec des balises hiérarchiques telles que `<head>` ou `<body>`. Le fichier CSS est plutôt une simple liste de règles traitées dans un ordre séquentiel par le navigateur. Les mêmes règles écrites à l'intérieur d'une balise `<style>` pourraient plutôt aller dans un fichier CSS.

L'association entre le document HTML et les règles CSS stockées dans un fichier est définie uniquement dans le document HTML. Pour le fichier CSS, il importe peu que des éléments correspondant à ses règles existent, il n'est donc pas nécessaire d'énumérer dans le fichier CSS les documents HTML auxquels il est lié. Du côté HTML, chaque feuille de style liée sera appliquée au

document, comme si les règles étaient écrites dans une balise `<style>`.

La balise HTML `<link>` définit la feuille de style externe à utiliser dans le document actuel et doit être placée dans la section head du document HTML :

```
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>

  <link href="style.css" rel="stylesheet">

</head>
```

Vous pouvez maintenant placer la règle pour l'élément `p` que nous avons utilisée précédemment dans le fichier `style.css`, et les résultats vus par le visiteur de la page Web seront les mêmes. Si le fichier CSS ne se trouve pas dans le même répertoire que le document HTML, indiquez son chemin relatif ou complet dans l'attribut `href`. La balise `<link>` peut faire référence à différents types de ressources externes, il est donc important d'établir la relation entre la ressource externe et le document actuel. Pour les fichiers CSS externes, la relation est définie dans `rel="stylesheet"`.

L'attribut `media` peut être utilisé de la même manière que pour la balise `<style>` : il indique le support, tel que les écrans d'ordinateur ou l'impression, auquel les règles du fichier externe doivent s'appliquer.

CSS peut modifier complètement un élément, mais il est toujours important d'utiliser l'élément approprié pour les composants de la page. Sinon, les technologies d'assistance telles que les lecteurs d'écran risquent de ne pas pouvoir identifier les sections correctes de la page.

Exercices guidés

1. Quelles méthodes peuvent être utilisées pour modifier l'apparence des éléments HTML à l'aide de CSS ?

2. Pourquoi n'est-il pas recommandé d'utiliser l'attribut `style` de la balise `<p>` s'il y a des paragraphes similaires qui doivent avoir la même apparence ?

3. Quelle est la politique de placement par défaut pour placer un élément `div` ?

4. Quel attribut de la balise `<link>` indique l'emplacement d'un fichier CSS externe ?

5. Quelle est la section correcte pour insérer l'élément `link` dans un document HTML ?

Exercices d'exploration

1. Pourquoi n'est-il pas recommandé d'utiliser une balise `<div>` pour identifier un mot dans une phrase ordinaire ?

2. Que se passe-t-il si vous commencez un commentaire avec `/*` au milieu d'un fichier CSS, mais que vous oubliez de le fermer avec `*/` ?

3. Écrivez une règle CSS pour souligner tous les éléments `em` du document.

4. Comment indiquer qu'une feuille de style provenant d'une balise `<style>` ou `<link>` ne doit être utilisée que par les synthétiseurs vocaux ?

Résumé

Cette leçon couvre les concepts de base de CSS et la manière de l'intégrer à HTML. Les règles écrites dans les feuilles de style CSS constituent la méthode standard pour modifier l'apparence des documents HTML. CSS nous permet de garder le contenu sémantique séparé des règles de présentation. Cette leçon aborde les concepts et les procédures suivants :

- Comment modifier les propriétés CSS à l'aide de l'attribut `style`.
- La syntaxe de base des règles CSS.
- Utilisation de la balise `<style>` pour intégrer des règles CSS dans le document.
- Utilisation de la balise `<link>` pour intégrer des feuilles de style externes au document.

Réponses aux exercices guidés

1. Quelles méthodes peuvent être utilisées pour modifier l'apparence des éléments HTML à l'aide de CSS ?

Trois méthodes basiques : L'écrire directement dans la balise de l'élément, dans une section distincte du code source de la page, ou dans un fichier externe qui sera référencé par la page HTML.

2. Pourquoi n'est-il pas recommandé d'utiliser l'attribut `style` de la balise `<p>` s'il y a des paragraphes similaires qui doivent avoir la même apparence ?

La déclaration CSS devra être reproduite dans les autres balises `<p>`, ce qui prend du temps, augmente la taille du fichier et est sujet à des erreurs.

3. Quelle est la politique de placement par défaut pour placer un élément `div` ?

L'élément `div` est traité comme un élément de bloc par défaut, il occupera donc tout l'espace horizontal de son élément parent et sa hauteur dépendra de son contenu.

4. Quel attribut de la balise `<link>` indique l'emplacement d'un fichier CSS externe ?

L'attribut `href`.

5. Quelle est la section correcte pour insérer l'élément `link` dans un document HTML ?

L'élément `link` doit se trouver dans la section `head` du document HTML.

Réponses aux exercices d'exploration

1. Pourquoi n'est-il pas recommandé d'utiliser une balise `<div>` pour identifier un mot dans une phrase ordinaire ?

La balise `<div>` fournit une séparation sémantique entre deux sections distinctes du document et interfère avec les outils d'accessibilité lorsqu'elle est utilisée pour des éléments de texte en ligne.

2. Que se passe-t-il si vous commencez un commentaire avec `/*` au milieu d'un fichier CSS, mais que vous oubliez de le fermer avec `*/` ?

Toutes les règles qui suivent le commentaire seront ignorées par le navigateur.

3. Écrivez une règle CSS pour souligner tous les éléments `em` du document.

```
em { text-decoration: underline }
```

ou

```
em { text-decoration-line: underline }
```

4. Comment indiquer qu'une feuille de style provenant d'une balise `<style>` ou `<link>` ne doit être utilisée que par les synthétiseurs vocaux ?

La valeur de son attribut `media` doit être `speech`.



**Linux
Professional
Institute**

033.2 Sélecteurs CSS et application de style

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 033.2

Valeur

3

Domaines de connaissance les plus importants

- Utiliser des sélecteurs pour appliquer des règles CSS aux éléments
- Comprendre les pseudo-classes CSS
- Comprendre l'ordre et la préséance des règles en CSS
- Comprendre l'héritage en CSS

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `element; .class; #id;`
- `a, b; a.class; a[attr] a b;`
- `:hover, focus`
- `!important`



033.2 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	033 Styliser le contenu avec CSS
Objectif :	033.2 Sélecteurs CSS et application de style
Leçon :	1 sur 1

Introduction

Lorsque nous écrivons une règle CSS, nous devons indiquer au navigateur à quels éléments la règle s'applique. Pour ce faire, nous spécifions un *sélecteur* : un motif qui peut correspondre à un élément ou à un groupe d'éléments. Les sélecteurs se présentent sous de nombreuses formes différentes, qui peuvent être basées sur le nom de l'élément, ses attributs, son emplacement relatif dans la structure du document, ou une combinaison de ces caractéristiques.

Styles pour toute la page

L'un des avantages de l'utilisation de CSS est que vous n'avez pas besoin d'écrire des règles individuelles pour les éléments partageant le même style. Un astérisque applique la règle à tous les éléments de la page web, comme le montre l'exemple suivant :

```
* {  
  color: purple;  
  font-size: large;  
}
```

Le sélecteur `*` n'est pas la seule méthode permettant d'appliquer une règle de style à tous les éléments de la page. Un sélecteur qui correspond simplement à un élément par son nom de balise s'appelle un sélecteur de type. Ainsi, tous les noms de balises HTML tels que `body`, `p`, `table`, `em`, etc. peuvent être utilisés comme sélecteurs. En CSS, le style du parent est *hérité* par ses éléments enfants. Ainsi, en pratique, l'utilisation du sélecteur `*` a le même effet que l'application d'une règle à l'élément `body` :

```
body {  
  color: purple;  
  font-size: large;  
}
```

De plus, la structure en cascade de CSS vous permet d'affiner les propriétés héritées d'un élément. Vous pouvez écrire une règle CSS générale qui s'applique à tous les éléments de la page, puis écrire des règles pour des éléments ou des ensembles d'éléments spécifiques.

Si le même élément correspond à deux ou plusieurs règles contradictoires, le navigateur applique la règle du sélecteur le plus spécifique. Prenons l'exemple des règles CSS suivantes :

```
body {  
  color: purple;  
  font-size: large;  
}  
  
li {  
  font-size: small;  
}
```

Le navigateur appliquera les styles `color: purple` et `font-size: large` à tous les éléments contenus dans l'élément `body`. Toutefois, si la page contient des éléments `li`, le navigateur remplacera le style `font-size: large` par le style `font-size: small`, car le sélecteur `li` a une relation plus forte avec l'élément `li` que le sélecteur `body`.

CSS ne limite pas le nombre de sélecteurs équivalents dans une même feuille de style. Vous pouvez donc avoir deux règles ou plus utilisant le même sélecteur :

```
li {  
  font-size: small;  
}
```



```
li {  
  font-size: x-small;  
}
```

Dans ce cas, les deux règles sont également spécifiques à l'élément `li`. Le navigateur ne peut pas appliquer des règles contradictoires, il choisira donc la règle qui vient en dernier dans le fichier CSS. Pour éviter toute confusion, il est recommandé de regrouper toutes les propriétés qui utilisent le même sélecteur.

L'ordre dans lequel les règles apparaissent dans la feuille de style affecte la manière dont elles sont appliquées dans le document, mais vous pouvez outrepasser ce comportement en utilisant une règle `important`. Si, pour une raison quelconque, vous souhaitez conserver les deux règles `li` distinctes, mais forcer l'application de la première au lieu de la seconde, marquez la première règle comme importante :

```
li {  
  font-size: small !important;  
}  
  
li {  
  font-size: x-small;  
}
```

Les règles marquées avec `!important` doivent être utilisées avec précaution, car elles brisent la cascade naturelle de la feuille de style et rendent plus difficile la recherche et la correction des problèmes dans le fichier CSS.

Sélecteurs restrictifs

Nous avons vu que nous pouvons modifier certaines propriétés héritées en utilisant des sélecteurs correspondant à des balises spécifiques. Cependant, nous devons généralement utiliser des styles distincts pour les éléments du même type.

Les attributs des balises HTML peuvent être incorporés dans des sélecteurs pour restreindre l'ensemble des éléments auxquels ils font référence. Supposons que la page HTML sur laquelle vous travaillez comporte deux types de listes non ordonnées (``) : l'une est utilisée en haut de la page comme menu des sections du site et l'autre est utilisée pour les listes classiques dans le corps du texte :

```
<!DOCTYPE html>
```

```
<html>
<head>
  <meta charset="utf-8" />
  <title>CSS Basics</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>

<div id="content">

<p>The three rocky planets of the solar system are:</p>

<ul>
  <li>Mercury</li>
  <li>Venus</li>
  <li>Earth</li>
  <li>Mars</li>
</ul>

<p>The outer giant planets made most of gas are:</p>

<ul>
  <li>Jupiter</li>
  <li>Saturn</li>
  <li>Uranus</li>
  <li>Neptune</li>
</ul>

</div><!-- #content -->

<div id="footer">

<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>
```

```

</div><!-- #footer -->

</body>
</html>

```

Par défaut, chaque élément de liste comporte un cercle noir à sa gauche. Vous pouvez vouloir supprimer les cercles de la liste du menu supérieur tout en laissant les cercles dans l'autre liste. Cependant, vous ne pouvez pas simplement utiliser le sélecteur `li` car cela supprimerait également les cercles de la liste située dans le corps du texte. Vous devez trouver un moyen d'indiquer au navigateur de ne modifier que les éléments de liste utilisés dans une liste, mais pas dans l'autre.

Il existe plusieurs façons d'écrire des sélecteurs correspondant à des éléments spécifiques de la page. Comme nous l'avons mentionné précédemment, nous allons d'abord voir comment utiliser les attributs des éléments pour le faire. Pour cet exemple en particulier, nous pouvons utiliser l'attribut `id` pour spécifier la liste supérieure uniquement.

L'attribut `id` attribue un identifiant unique à l'élément correspondant, que nous pouvons utiliser comme sélecteur dans la règle CSS. Avant d'écrire la règle CSS, modifiez le fichier HTML d'exemple et ajoutez `id="topmenu"` à l'élément `ul` utilisé pour le menu supérieur :

```

<ul id="topmenu">
  <li>Home</li>
  <li>Articles</li>
  <li>About</li>
</ul>

```

Il existe déjà un élément `link` dans la section `head` du document HTML qui pointe vers le fichier de feuille de style `style.css` dans le même dossier, nous pouvons donc y ajouter les règles CSS suivantes :

```

ul#topmenu {
  list-style-type: none
}

```

Le caractère dièse est utilisé dans un sélecteur, à la suite d'un élément, pour désigner un identifiant (sans espace les séparant). Le nom de la balise à gauche du caractère dièse est facultatif, car il n'y aura pas d'autre élément avec le même identifiant. Par conséquent, le sélecteur pourrait être écrit simplement comme `#topmenu`.

Même si la propriété `list-style-type` n'est pas une propriété directe de l'élément `ul`, les propriétés

CSS de l'élément parent sont héritées par ses enfants, donc le style attribué à l'élément `ul` sera hérité par ses éléments `li` enfants.

Tous les éléments ne disposent pas d'un identifiant permettant de les sélectionner. En effet, seuls quelques éléments clés de la mise en page d'une page sont censés avoir un identifiant. Prenez par exemple les listes de planètes utilisées dans l'exemple de code. Bien qu'il soit possible d'attribuer des identifiants uniques à chacun de ces éléments répétés, cette méthode n'est pas pratique pour les pages plus longues comportant de nombreux contenus. Nous pouvons plutôt utiliser l'identifiant de l'élément parent `div` comme sélecteur pour modifier les propriétés de ses éléments internes.

Cependant, l'élément `div` n'est pas directement lié aux listes HTML, donc lui ajouter la propriété `list-style-type` n'aura aucun effet sur ses enfants. Ainsi, pour changer le cercle noir des listes à l'intérieur du contenu `div` en un cercle creux, nous devons utiliser un sélecteur *descendant* :

```
#topmenu {  
  list-style-type: none  
}  
  
#content ul {  
  list-style-type: circle  
}
```

Le sélecteur `#content ul` est appelé sélecteur descendant car il ne correspond qu'aux éléments `ul` qui sont enfants de l'élément dont l'identifiant est `content`. Nous pouvons utiliser autant de niveaux de descendance que nécessaire. Par exemple, l'utilisation de `#content ul li` ne correspondrait qu'aux éléments `li` qui sont des descendants des éléments `ul` qui sont des descendants de l'élément dont l'ID est `content`. Mais dans cet exemple, le sélecteur le plus long aura le même effet que l'utilisation de `#content ul`, car les éléments `li` héritent des propriétés CSS définies pour leur parent `ul`. Les sélecteurs descendants sont une technique essentielle lorsque la mise en page devient plus complexe.

Supposons que vous souhaitiez modifier la propriété `font-style` des éléments de la liste `topmenu` et de la liste de bas de page `footer div` pour leur donner un aspect oblique. Vous ne pouvez pas simplement écrire une règle CSS en utilisant `ul` comme sélecteur, car cela modifierait également les éléments de la liste de la division `content div`. Jusqu'à présent, nous avons modifié les propriétés CSS en utilisant un sélecteur à la fois, et cette méthode peut également être utilisée pour cette tâche :

```
#topmenu {  
  font-style: oblique  
}  
  
#footer ul {
```

```
font-style: oblique
}
```

Les sélecteurs distincts ne sont cependant pas la seule façon de procéder. Le CSS nous permet de regrouper les sélecteurs qui partagent un ou plusieurs styles, en utilisant une liste de sélecteurs séparés par des virgules :

```
#topmenu, #footer ul {
  font-style: oblique
}
```

Le regroupement des sélecteurs élimine le travail supplémentaire que représente l'écriture de styles en double. En outre, il se peut que vous souhaitiez modifier à nouveau la propriété à l'avenir et que vous ne vous souveniez pas de la modifier à tous les endroits différents.

Classes

Si vous ne voulez pas trop vous soucier de la hiérarchie des éléments, vous pouvez simplement ajouter une classe `class` à l'ensemble des éléments que vous voulez personnaliser. Les classes sont similaires aux identifiants, mais au lieu d'identifier un seul élément dans la page, elles peuvent identifier un groupe d'éléments partageant les mêmes caractéristiques.

Prenez l'exemple de la page HTML sur laquelle nous travaillons, par exemple. Il est peu probable que dans les pages du monde réel, nous trouvions des structures aussi simples que celle-ci. Il serait donc plus pratique de sélectionner un élément en utilisant uniquement les classes, ou une combinaison de classes et de descendance. Pour appliquer la propriété `font-style: oblique` aux listes de menus en utilisant les classes, nous devons d'abord ajouter la propriété `class` aux éléments du fichier HTML. Nous le ferons d'abord dans le menu supérieur :

```
<ul id="topmenu" class="menu">
  <li><a href="/">Home</a></li>
  <li><a href="/articles">Articles</a></li>
  <li><a href="/about">About</a></li>
</ul>
```

Et ensuite dans le menu du bas de page :

```
<div id="footer">

<ul class="menu">
```

```
<li><a href="/">Home</a></li>
<li><a href="/articles">Articles</a></li>
<li><a href="/about">About</a></li>
</ul>

</div><!-- #footer -->
```

Avec ça, nous pouvons remplacer le groupe de sélecteurs `#topmenu`, `#footer ul` par le sélecteur basé sur la classe `.menu` :

```
.menu {
  font-style: oblique
}
```

Comme pour les sélecteurs basés sur les identifiants, l'ajout du nom de la balise à gauche du point dans les sélecteurs basés sur les classes est facultatif. Toutefois, contrairement aux identifiants, la même classe est censée être utilisée dans plus d'un élément et il n'est même pas nécessaire qu'ils soient du même type. Par conséquent, si la classe `menu` est partagée entre différents types d'éléments, l'utilisation du sélecteur `ul.menu` ne correspondrait qu'aux éléments `ul` ayant la classe `menu`. Au contraire, l'utilisation du sélecteur `.menu` correspondra à tout élément ayant la classe `menu`, quel que soit son type.

De plus, les éléments peuvent être associés à plus d'une classe. Nous pourrions différencier le menu supérieur et le menu inférieur en ajoutant une classe supplémentaire à chacun d'eux :

```
<ul id="topmenu" class="menu top">
```

Et dans le menu du bas de page :

```
<ul class="menu footer">
```

Lorsque l'attribut `class` comporte plus d'une classe, elles doivent être séparées par des espaces. Maintenant, en plus de la règle CSS partagée entre les éléments de la classe `menu`, nous pouvons aborder les menus du haut et du bas de page en utilisant leurs classes correspondantes :

```
.menu {
  font-style: oblique
}
```

```
.menu.top {  
  font-size: large  
}  
  
.menu.footer {  
  font-size: small  
}
```

Faites attention au fait que l'écriture de `.menu.top` diffère de celle de `.menu .top` (avec un espace entre les mots). Le premier sélecteur correspondra aux éléments qui ont les deux classes `menu` et `top`, tandis que le second correspondra aux éléments qui ont la classe `top` et un élément parent avec la classe `menu`.

Sélecteurs spéciaux

Les sélecteurs CSS peuvent également correspondre aux états dynamiques des éléments. Ces sélecteurs sont particulièrement utiles pour les éléments interactifs, tels que les hyperliens. Vous pouvez souhaiter modifier l'apparence des hyperliens lorsque le pointeur de la souris passe dessus, afin d'attirer l'attention du visiteur.

Pour revenir à notre page d'exemple, nous pourrions supprimer le soulignement des liens dans le menu supérieur et afficher une ligne uniquement lorsque le pointeur de la souris passe sur le lien correspondant. Pour ce faire, nous écrivons d'abord une règle pour supprimer le soulignement des liens dans le menu supérieur :

```
.menu.top a {  
  text-decoration: none  
}
```

Ensuite, nous utilisons la pseudo-classe `:hover` sur ces mêmes éléments pour créer une règle CSS qui s'appliquera uniquement lorsque le pointeur de la souris se trouve sur l'élément correspondant :

```
.menu.top a:hover {  
  text-decoration: underline  
}
```

Le sélecteur de pseudo-classe `:hover` accepte toutes les propriétés CSS des règles CSS classiques. Les autres pseudo-classes comprennent `:visited`, qui correspond aux hyperliens qui ont déjà été visités, et `:focus`, qui correspond aux éléments de formulaire qui ont reçu le focus.

Exercices guidés

1. Supposons qu'une page HTML soit associée à une feuille de style contenant les deux règles suivantes :

```
p {  
  color: green;  
}  
  
p {  
  color: red;  
}
```

Quelle couleur le navigateur appliquera-t-il au texte contenu dans les éléments `p` ?

2. Quelle est la différence entre l'utilisation du sélecteur d'identification `div#main` et `#main` ?

3. Quel sélecteur correspond à tous les éléments `p` utilisés à l'intérieur d'une `div` avec l'attribut d'identification `#main` ?

4. Quelle est la différence entre l'utilisation du sélecteur de classe `p.highlight` et `.highlight` ?

Exercices d'exploration

1. Écrire une seule règle CSS qui change la propriété `font-style` en oblique. La règle doit correspondre uniquement aux éléments `a` qui se trouvent à l'intérieur de `<div id="sidebar"></div>` ou de `<ul class="links">`.

2. Supposons que vous souhaitiez modifier le style des éléments dont l'attribut `class` est défini sur `article reference`, comme dans `<p class="article reference">`. Cependant, le sélecteur `.article .reference` ne semble pas modifier leur apparence. Pourquoi le sélecteur ne correspond-il pas aux éléments comme prévu ?

3. Écrire une règle CSS pour changer la propriété `color` de tous les liens visités dans la page à `red`.

Résumé

Cette leçon explique comment utiliser les sélecteurs CSS et comment le navigateur décide des styles à appliquer à chaque élément. Séparé du balisage HTML, CSS fournit de nombreux sélecteurs permettant de faire correspondre des éléments individuels ou des groupes d'éléments dans la page. La leçon aborde les concepts et procédures suivants :

- Styles pour toute la page et héritage des styles.
- Stylisation des éléments par type.
- Utilisation de l'identifiant et de la classe de l'élément comme sélecteur.
- Sélecteurs composés.
- Utilisation de pseudo-classes pour styliser dynamiquement les éléments.

Réponses aux exercices guidés

1. Supposons qu'une page HTML soit associée à une feuille de style contenant les deux règles suivantes :

```
p {  
  color: green;  
}  
  
p {  
  color: red;  
}
```

Quelle couleur le navigateur appliquera-t-il au texte contenu dans les éléments `p` ?

La couleur `red`. Lorsque deux sélecteurs équivalents ou plus ont des propriétés contradictoires, le navigateur choisira le dernier.

2. Quelle est la différence entre l'utilisation du sélecteur d'identification `div#main` et `#main` ?

Le sélecteur `div#main` correspond à un élément `div` ayant pour identifiant `main`, tandis que le sélecteur `#main` correspond à l'élément ayant pour identifiant `main`, quel que soit son type.

3. Quel sélecteur correspond à tous les éléments `p` utilisés à l'intérieur d'une `div` avec l'attribut d'identification `#main` ?

Le sélecteur `#main p` ou `div#main p`.

4. Quelle est la différence entre l'utilisation du sélecteur de classe `p.highlight` et `.highlight` ?

Le sélecteur `p.highlight` correspond uniquement aux éléments de type `p` ayant la classe `highlight`, tandis que le sélecteur `.highlight` correspond à tous les éléments ayant la classe `highlight`, quel que soit leur type.

Réponses aux exercices d'exploration

1. Écrire une seule règle CSS qui change la propriété `font-style` en `oblique`. La règle doit correspondre uniquement aux éléments `a` qui se trouvent à l'intérieur de `<div id="sidebar"></div>` ou de `<ul class="links">`.

```
#sidebar a, ul.links a {  
    font-style: oblique  
}
```

2. Supposons que vous souhaitiez modifier le style des éléments dont l'attribut `class` est défini sur `article reference`, comme dans `<p class="article reference">`. Cependant, le sélecteur `.article .reference` ne semble pas modifier leur apparence. Pourquoi le sélecteur ne correspond-il pas aux éléments comme prévu ?

Le sélecteur `.article .reference` correspondra aux éléments ayant la classe `reference` qui sont des descendants des éléments ayant la classe `article`. Pour faire correspondre des éléments ayant à la fois les classes `article` et `reference`, le sélecteur doit être `.article.reference` (sans espace entre les deux).

3. Écrire une règle CSS pour changer la propriété `color` de tous les liens visités dans la page à `red`.

```
a:visited {  
    color: red;  
}
```



**Linux
Professional
Institute**

033.3 Stylistation CSS

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 033.3

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les propriétés CSS fondamentales
- Comprendre les unités couramment utilisées en CSS

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- px, %, em, rem, vw, vh
- color, background, background-*, font, font-*, text-*, list-style, line-height



033.3 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	033 Styliser le contenu avec CSS
Objectif :	033.3 Stylisation CSS
Leçon :	1 sur 1

Introduction

CSS fournit des centaines de propriétés qui peuvent être utilisées pour modifier l'apparence des éléments HTML. Seuls les concepteurs expérimentés parviennent à se souvenir de la plupart d'entre elles. Néanmoins, il est pratique de connaître les propriétés de base qui s'appliquent à tout élément, ainsi que certaines propriétés spécifiques aux éléments. Ce chapitre couvre certaines propriétés populaires que vous êtes susceptible d'utiliser.

Propriétés et valeurs communes de CSS

De nombreuses propriétés CSS ont le même type de valeur. Les couleurs, par exemple, ont le même format numérique, que vous changiez la couleur de la police ou celle de l'arrière-plan. De même, les unités disponibles pour modifier la taille de la police sont également utilisées pour modifier l'épaisseur d'une bordure. Toutefois, le format de la valeur n'est pas toujours unique. Les couleurs, par exemple, peuvent être saisies dans différents formats, comme nous allons le voir ci-après.

Couleurs

Changer la couleur d'un élément est probablement l'une des premières choses que les concepteurs

apprennent à faire avec CSS. Vous pouvez modifier la couleur du texte, la couleur d'arrière-plan, la couleur de la bordure des éléments, et bien plus encore.

La valeur d'une couleur en CSS peut être écrite sous la forme d'un *mot-clé de couleur* (c'est-à-dire un nom de couleur en anglais) ou d'une valeur numérique énumérant chaque composant de couleur. Tous les noms de couleur courants, tels que black, white, red, purple, green, yellow, et blue sont acceptés comme mots-clés de couleur. La liste complète des mots-clés de couleur acceptés par CSS est disponible à la [page web du W3C](#). Mais pour avoir un contrôle plus fin de la couleur, vous pouvez utiliser la valeur numérique.

Mots-clés de couleur

Nous utiliserons d'abord le mot-clé de couleur, car il est plus simple. La propriété `color` modifie la couleur du texte dans l'élément correspondant. Ainsi, pour mettre tout le texte de la page en violet, vous pourriez écrire la règle CSS suivante :

```
* {
  color: purple;
}
```

Valeurs numériques de couleur

Bien qu'intuitifs, les mots-clés de couleur n'offrent pas la gamme complète des couleurs possibles sur les écrans modernes. Les concepteurs de sites web développent généralement une palette de couleurs qui utilise des couleurs personnalisées, en attribuant des valeurs spécifiques aux composants rouge, vert et bleu.

Chaque composante de couleur est un nombre binaire de huit bits, allant de 0 à 255. Les trois composantes doivent être spécifiées dans le mélange de couleurs, et leur ordre est toujours rouge, vert, bleu. Par conséquent, pour changer la couleur de tout le texte de la page en rouge en utilisant la notation RVB (*RGB : red, green, blue*), utilisez `rgb(255, 0, 0)` :

```
* {
  color: rgb(255, 0, 0);
}
```

Une composante définie sur 0 signifie que la couleur de base correspondante n'est pas utilisée dans le mélange de couleurs. Des pourcentages peuvent également être utilisés à la place des chiffres :

```
* {
```

```
color: rgb(100%, 0%, 0%);  
}
```

La notation RVB est rarement utilisée si vous utilisez une application de dessin pour créer des mises en page ou simplement pour choisir ses couleurs. Il est plutôt courant de voir les couleurs dans les CSS exprimées sous forme de valeurs *hexadécimales*. Les composantes des couleurs en hexadécimal vont également de 0 à 255, mais de manière plus succincte, en commençant par un symbole dièse # et en utilisant une longueur fixe à deux chiffres pour toutes les composantes. La valeur minimale 0 est 00 et la valeur maximale 255 est FF, ainsi la couleur rouge red est #FF0000.

TIP

Si les chiffres de chaque composant d'une couleur hexadécimale se répètent, le deuxième chiffre peut être omis. La couleur red, par exemple, peut être écrite avec un seul chiffre pour chaque composant : #F00.

La liste suivante indique la notation RVB et hexadécimale de certaines couleurs de base :

Mot-clé de couleur	Notation RVB	Valeur hexadécimale
black	rgb(0, 0, 0)	#000000
white	rgb(255, 255, 255)	#FFFFFF
red	rgb(255, 0, 0)	#FF0000
purple	rgb(128, 0, 128)	#800080
green	rgb(0, 128, 0)	#008000
yellow	rgb(255, 255, 0)	#FFFF00
blue	rgb(0, 0, 255)	#0000FF

Les mots-clés de couleur et les lettres des valeurs de couleur hexadécimales sont insensibles à la casse.

Opacité de couleur

En plus des couleurs opaques, il est possible de remplir les éléments de la page avec des couleurs semi-transparentes. L'opacité d'une couleur peut être définie à l'aide d'une quatrième composante de la valeur de la couleur. Contrairement aux autres composantes de couleur, dont les valeurs sont des nombres entiers compris entre 0 et 255, l'opacité est une fraction comprise entre 0 et 1.

La valeur la plus basse, 0, donne une couleur complètement transparente, qui ne peut être distinguée d'aucune autre couleur complètement transparente. La valeur la plus élevée, 1, donne une couleur totalement opaque, qui est la même que la couleur originale sans aucune transparence.

Lorsque vous utilisez la notation RVB, spécifiez les couleurs avec une composante d'opacité par le préfixe `rgba` au lieu de `rgb`. Ainsi, pour rendre la couleur rouge semi-transparente, utilisez `rgba(255,0,0,0.5)`. Le caractère `a` représente le *canal alpha*, un terme couramment utilisé pour spécifier la composante d'opacité dans le jargon des graphistes numériques.

L'opacité peut également être définie en notation hexadécimale. Ici, comme pour les autres composantes de couleur, l'opacité va de `00` à `FF`. Par conséquent, pour rendre la couleur rouge semi-transparente en notation hexadécimale, utilisez `#FF000080`.

Arrière-plan

La couleur de l'arrière-plan des éléments individuels ou de la page entière est définie par la propriété `background-color`. Elle prend les mêmes valeurs que la propriété `color`, soit sous forme de mots-clés, soit en utilisant la notation RVB/hexadécimale.

L'arrière-plan des éléments HTML n'est cependant pas limité aux couleurs. Avec la propriété `background-image`, il est possible d'utiliser une image comme arrière-plan. Les formats d'image acceptés sont tous les formats classiques acceptés par le navigateur, tels que JPEG et PNG.

Le chemin d'accès à l'image doit être spécifié à l'aide d'un indicateur `url()`. Si l'image que vous souhaitez utiliser se trouve dans le même dossier que le fichier HTML, il suffit d'utiliser son nom de fichier :

```
body {  
  background-image: url("background.jpg");  
}
```

Dans cet exemple, le fichier image `background.jpg` sera utilisé comme image de fond pour l'ensemble du corps de la page. Par défaut, l'image d'arrière-plan est répétée si sa taille ne couvre pas toute la page, à partir du coin supérieur gauche de la zone correspondant au sélecteur de la règle. Ce comportement peut être modifié avec la propriété `background-repeat`. Si vous souhaitez que l'image d'arrière-plan soit placée dans la zone de l'élément sans être répétée, utilisez la valeur `no-repeat` :

```
body {  
  background-image: url("background.jpg");  
  background-repeat: no-repeat;  
}
```

Vous pouvez également faire en sorte que l'image soit répétée uniquement dans le sens horizontal (`background-repeat: repeat-x`) ou uniquement dans le sens vertical (`background-repeat: repeat-y`) :

y).

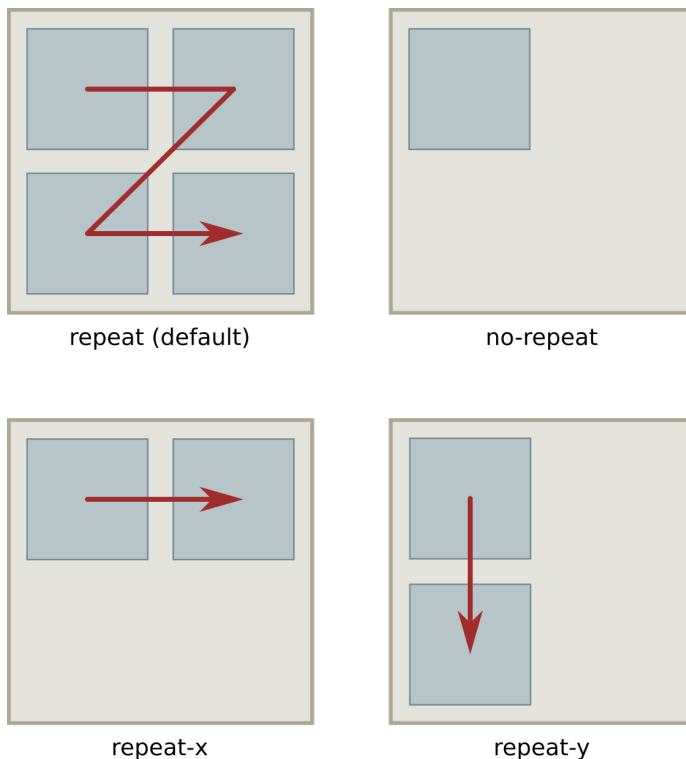


Figure 34. Placement de l'arrière-plan en utilisant la propriété `background-repeat`.

TIP

Deux ou plusieurs propriétés CSS peuvent être combinées en une seule propriété, appelée propriété d'arrière-plan *raccourcie*. Les propriétés `background-image` et `background-repeat`, par exemple, peuvent être combinées en une seule propriété d'arrière-plan avec `background: no-repeat url("background.jpg")`.

Une image d'arrière-plan peut également être placée à un endroit précis de la zone de l'élément à l'aide de la propriété `background-position`. Les cinq positions de base sont `top`, `bottom`, `left`, `right` et `center`, mais la position haut-gauche de l'image peut aussi être ajustée avec des pourcentages :

```
body {  
  background-image: url("background.jpg");  
  background-repeat: no-repeat;  
  background-position: 30% 10%;  
}
```

Le pourcentage pour chaque position est relatif à la taille correspondante de l'élément. Dans l'exemple, le côté gauche de l'image d'arrière-plan se situera à 30 % de la largeur du corps (le corps est généralement l'ensemble du document visible) et le côté supérieur de l'image se situera à 10 % de

la hauteur du corps.

Bordures

La modification de la bordure d'un élément est également une personnalisation courante de la mise en page réalisée à l'aide de CSS. La bordure fait référence à la ligne formant un rectangle autour de l'élément et possède trois propriétés de base : `color`, `style` et `width`.

La couleur de la bordure, définie avec `border-color`, suit le même format que nous avons vu pour les autres propriétés de couleur.

Les bordures peuvent être tracées dans un style autre qu'une ligne pleine. Vous pouvez, par exemple, utiliser des tirets pour la bordure avec la propriété `border-style: dashed`. Les autres valeurs de style possibles sont :

dotted

Une séquence de points arrondis

double

Deux lignes droites

groove

Une ligne ayant un aspect sculpté

ridge

Une ligne avec un aspect extrudé

inset

Un élément qui apparaît encastré

outset

Un élément qui semble en relief

La propriété `border-width` définit l'épaisseur de la bordure. Sa valeur peut être un mot-clé (`thin`, `medium` ou `thick`) ou une valeur numérique spécifique. Si vous préférez utiliser une valeur numérique, vous devrez également spécifier l'unité correspondante. Ceci sera décrit ci-dessous.

Valeurs d'unité

Les tailles et les distances en CSS peuvent être définies de différentes manières. Les unités absolues sont basées sur un nombre fixe de pixels d'écran, elles ne sont donc pas si différentes des tailles et

dimensions fixes utilisées dans les médias imprimés. Il existe également des unités relatives, qui sont calculées dynamiquement à partir d'une mesure donnée par le média où la page est rendue, comme l'espace disponible ou une autre taille écrite en unités absolues.

Unités absolues

Les unités absolues sont équivalentes à leurs homologues physiques, comme les centimètres ou les pouces. Sur les écrans d'ordinateur classiques, un pouce correspond à une largeur de 96 pixels. Les unités absolues suivantes sont couramment utilisées :

in (pouce)

1 pouce (*inch*) est équivalent à 2,54 cm ou 96 px.

cm (centimètre)

1 cm est équivalent à 96 px / 2,54.

mm (millimètre)

1 mm est équivalent à 1 cm / 10.

px (pixel)

1 px est équivalent à 1 / 96ème de pouce.

pt (point)

1pt est équivalent à 1 / 72ème de pouce.

Gardez à l'esprit que le rapport entre les pixels et les pouces peut varier. Sur les écrans haute résolution, où les pixels sont plus denses, un pouce correspondra à plus de 96 pixels.

Unités relatives

Les unités relatives varient en fonction d'autres mesures ou des dimensions de la fenêtre d'affichage. La fenêtre d'affichage est la zone du document actuellement visible dans sa fenêtre. En mode plein écran, la fenêtre d'affichage correspond à l'écran du périphérique. Les unités relatives suivantes sont couramment utilisées :

%

Pourcentage — il est relatif à l'élément parent.

em

La taille de la police utilisée dans l'élément.

rem

La taille de la police utilisée dans l'élément racine.

vw

1% de la largeur de la fenêtre d'affichage.

vh

1% de la hauteur de la fenêtre d'affichage.

L'avantage de l'utilisation d'unités relatives est que vous pouvez créer des mises en page qui sont ajustables en modifiant seulement quelques tailles clés. Par exemple, vous pouvez utiliser l'unité `pt` pour définir la taille de la police dans l'élément `body` et l'unité `rem` pour les polices des autres éléments. Une fois que vous aurez modifié la taille de la police de l'élément `body`, toutes les autres tailles de police s'adapteront en conséquence. Par ailleurs, l'utilisation de `vw` et `vh` pour définir les dimensions des sections de page permet de les adapter à des écrans de tailles différentes.

Propriétés des polices et du texte

La typographie, ou l'étude des types de police, est un sujet de conception très vaste, et la typographie CSS n'est pas en reste. Toutefois, il existe quelques propriétés de base des polices qui répondront aux besoins de la plupart des utilisateurs apprenant le CSS.

La propriété `font-family` définit le nom de la police à utiliser. Il n'y a aucune garantie que la police choisie sera disponible dans le système où la page sera affichée, de sorte que cette propriété peut n'avoir aucun effet dans le document. Bien qu'il soit possible de faire en sorte que le navigateur télécharge et utilise le fichier de police spécifié, la plupart des concepteurs de sites web se contentent d'utiliser une famille de polices générique dans leurs documents.

Les trois familles de polices génériques les plus courantes sont `serif`, `sans-serif` et `monospace`. La famille de polices utilisée par défaut dans la plupart des navigateurs est `serif`. Si vous préférez utiliser `sans-serif` pour l'ensemble de la page, ajoutez la règle suivante à votre feuille de style :

```
* {
  font-family: sans-serif;
}
```

Optionnellement, vous pouvez d'abord définir un nom de famille de polices spécifique, suivi du nom de famille générique :

```
* {
```

```
font-family: "DejaVu Sans", sans-serif;  
}
```

Si le périphérique qui affiche la page possède cette famille de polices spécifique, le navigateur l'utilisera. Sinon, il utilisera sa police par défaut correspondant au nom de famille générique. Les navigateurs recherchent les polices dans l'ordre où elles sont spécifiées dans la propriété.

Toute personne qui a déjà utilisé une application de traitement de texte connaît également trois autres réglages de police : la taille, le poids et le style. Ces trois réglages ont des propriétés CSS équivalentes : `font-size`, `font-weight` et `font-style`.

La propriété `font-size` accepte des mots-clés tels que `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, `xxx-large`. Ces mots-clés sont relatifs à la taille de police par défaut utilisée par le navigateur. Les mots-clés `larger` et `smaller` modifient la taille de la police par rapport à celle de l'élément parent. Vous pouvez également exprimer la taille de la police par des valeurs numériques, en incluant l'unité après la valeur, ou par des pourcentages.

Si vous ne voulez pas modifier la taille de la police, mais la distance entre les lignes, utilisez la propriété de hauteur de ligne `line-height`. Avec `line-height` à 1, la hauteur de la ligne sera égale à la taille de la police de l'élément, ce qui peut rendre les lignes du texte trop rapprochées. Par conséquent, une valeur supérieure à 1 est plus appropriée pour les textes. Comme pour la propriété `font-size`, d'autres unités peuvent être utilisées avec la valeur numérique.

Le paramètre `font-weight` définit l'épaisseur de la police avec les mots-clés familiers `normal` ou `bold`. Les mots-clés `lighter` et `bolder` modifient le poids de la police de l'élément par rapport au poids de la police de son élément parent.

La propriété `font-style` peut être définie sur `italic` pour sélectionner la version italique de la famille de polices actuelle. La valeur `oblique` sélectionne la version oblique de la police. Ces deux options sont presque identiques, mais la version italique d'une police est généralement un peu plus étroite que la version oblique. Si aucune version italique ou oblique de la police n'existe, la police sera artificiellement inclinée par le navigateur.

Il existe d'autres propriétés qui modifient la façon dont le texte est affiché dans le document. Vous pouvez, par exemple, ajouter un soulignement à certaines parties du texte que vous souhaitez mettre en valeur. Tout d'abord, utilisez une balise `` pour délimiter le texte :

```
<p>CSS is the <span class="under">proper mechanism</span> to style HTML  
documents.</p>
```

Vous pouvez ensuite utiliser le sélecteur `.under` pour modifier la propriété `text-decoration` :

```
.under {  
  text-decoration: underline;  
}
```

Par défaut, tous les éléments `a` (liens) sont soulignés. Si vous voulez le supprimer, utilisez la valeur `none` pour la propriété `text-decoration` de tous les éléments `a` :

```
a {  
  text-decoration: none;  
}
```

Lors de la révision du contenu, certains auteurs aiment barrer les parties incorrectes ou obsolètes du texte, afin que le lecteur sache que le texte a été mis à jour et ce qui a été supprimé. Pour ce faire, utilisez la valeur `line-through` de la propriété `text-decoration` :

```
.disregard {  
  text-decoration: line-through;  
}
```

Là encore, une balise `` peut être utilisée pour appliquer le style :

```
<p>Netscape Navigator is was one of the most popular  
Web browsers.</p>
```

D'autres décorations peuvent être spécifiques à un élément. Les cercles des listes à puces peuvent être personnalisés à l'aide de la propriété `list-style-type`. Pour les remplacer par des carrés, par exemple, utilisez la propriété `list-style-type: square`. Pour les supprimer tout simplement, mettez la valeur de `list-style-type` à `none`.

Exercices guidés

1. Comment ajouter une demi-transparence à la couleur bleue (notation RVB `rgb(0,0,255)`) afin de l'utiliser dans la propriété `color` de CSS ?

2. Quelle couleur correspond à la valeur hexadécimale `#000` ?

3. Étant donné que Times New Roman est une police serif et qu'elle ne sera pas disponible sur tous les appareils, comment pouvez-vous écrire une règle CSS pour demander Times New Roman tout en définissant la famille de polices génériques `serif` comme solution de remplacement ?

4. Comment utiliser un mot-clé de taille relative pour définir la taille de la police de l'élément `<p class="disclaimer">` plus petite par rapport à son élément parent ?

Exercices d'exploration

1. La propriété `background` est un raccourci pour définir plus d'une propriété `background-*` à la fois. Réécrire la règle CSS suivante comme une seule propriété raccourcie `background`.

```
body {  
  background-image: url("background.jpg");  
  background-repeat: repeat-x;  
}
```

2. Écrire une règle CSS pour l'élément `<div id="header"></div>` qui change *seulement* la largeur de sa bordure inférieure à 4px.

3. Écrire une propriété `font-size` qui double la taille de la police utilisée dans l'élément racine de la page.

4. Le *double interligne* est une fonction de formatage de texte courante dans les traitements de texte. Comment pourriez-vous définir un format similaire à l'aide de CSS ?

Résumé

Cette leçon couvre l'application de styles simples aux éléments d'un document HTML. CSS fournit des centaines de propriétés, et la plupart des concepteurs de sites web devront recourir à des manuels de référence pour se les rappeler toutes. Toutefois, un ensemble relativement restreint de propriétés et de valeurs est utilisé la plupart du temps, et il est important de les connaître par cœur. Cette leçon passe en revue les concepts et procédures suivants :

- Les propriétés fondamentales de CSS traitant des couleurs, des arrière-plans et des polices.
- Les unités absolues et relatives que CSS peut utiliser pour définir les tailles et les distances, telles que px, em, rem, vw, vh, etc.

Réponses aux exercices guidés

1. Comment ajouter une demi-transparence à la couleur bleue (notation RVB `rgb(0,0,255)`) afin de l'utiliser dans la propriété `color` de CSS ?

Utiliser le préfixe `rgba` et ajouter `0.5` comme quatrième valeur : `rgba(0,0,0,0.5)`.

2. Quelle couleur correspond à la valeur hexadécimale `#000` ?

La couleur noire `black`. La valeur hexadécimale `#000` est une abréviation de `#000000`.

3. Étant donné que `Times New Roman` est une police `serif` et qu'elle ne sera pas disponible sur tous les appareils, comment pouvez-vous écrire une règle CSS pour demander `Times New Roman` tout en définissant la famille de polices génériques `serif` comme solution de remplacement ?

```
* {  
  font-family: "Times New Roman", serif;  
}
```

4. Comment utiliser un mot-clé de taille relative pour définir la taille de la police de l'élément `<p class="disclaimer">` plus petite par rapport à son élément parent ?

En utilisant le mot-clé `smaller` :

```
p.disclaimer {  
  font-size: smaller;  
}
```

Réponses aux exercices d'exploration

1. La propriété `background` est un raccourci pour définir plus d'une propriété `background-*` à la fois. Réécrire la règle CSS suivante comme une seule propriété raccourcie `background`.

```
body {  
  background-image: url("background.jpg");  
  background-repeat: repeat-x;  
}
```

```
body {  
  background: repeat-x url("background.jpg");  
}
```

2. Écrire une règle CSS pour l'élément `<div id="header"></div>` qui change *seulement* la largeur de sa bordure inférieure à 4px.

```
#header {  
  border-bottom-width: 4px;  
}
```

3. Écrire une propriété `font-size` qui double la taille de la police utilisée dans l'élément racine de la page.

L'unité `rem` correspond à la taille de la police utilisée dans l'élément racine, donc la propriété doit être `font-size: 2rem`.

4. Le *double interligne* est une fonction de formatage de texte courante dans les traitements de texte. Comment pourriez-vous définir un format similaire à l'aide de CSS ?

Définir la propriété `line-height` à la valeur `2em`, car l'unité `em` correspond à la taille de la police de l'élément courant.



Linux
Professional
Institute

033.4 Modèle de boîte CSS et mise en page

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 033.4

Valeur

2

Domaines de connaissance les plus importants

- Définir la dimension, la position et l'alignement des éléments dans une mise en page CSS.
- Spécifier comment le texte circule autour des autres éléments
- Comprendre le flux du document
- Connaissance de la grille CSS
- Connaissance du responsive web design
- Connaissance des media queries CSS

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `width`, `height`, `padding`, `padding-*`, `margin`, `margin-*`, `border`, `border-*`
- `top`, `left`, `right`, `bottom`
- `display`: `block` | `inline` | `flex` | `inline-flex` | `none`
- `position`: `static` | `relative` | `absolute` | `fixed` | `sticky`
- `float`: `left` | `right` | `none`
- `clear`: `left` | `right` | `both` | `none`



033.4 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	033 Styliser le contenu avec CSS
Objectif :	033.4 Modèle de boîte CSS et mise en page
Leçon :	1 sur 1

Introduction

Chaque élément visible d'un document HTML est affiché sous la forme d'une boîte rectangulaire. Ainsi, le terme *modèle de boîte* (*box model*) décrit l'approche adoptée par CSS pour modifier les propriétés visuelles des éléments. Comme des boîtes de différentes tailles, les éléments HTML peuvent être imbriqués à l'intérieur d'éléments *conteneurs* (*container*)—généralement l'élément `div`—afin d'être séparés en sections.

Nous pouvons utiliser le CSS pour modifier la position des boîtes, qu'il s'agisse d'ajustements mineurs ou de changements radicaux dans la disposition des éléments de la page. Outre le flux normal, la position de chaque boîte peut être basée sur les éléments qui l'entourent, soit sa relation avec son conteneur parent, soit sa relation avec fenêtre d'affichage (*viewport*), qui est la zone de la page visible par l'utilisateur. Aucun mécanisme ne répond à toutes les exigences possibles en matière de mise en page ; vous pouvez donc avoir besoin d'une combinaison de ces mécanismes.

Flux normal

La façon dont le navigateur affiche par défaut l'arborescence du document est appelée *flux normal* (*normal flow*). Les rectangles correspondant aux éléments sont placés plus ou moins dans le même

ordre qu'ils apparaissent dans l'arbre du document, par rapport à leurs éléments parents. Néanmoins, selon le type d'élément, le rectangle correspondant peut suivre des règles de positionnement distinctes.

Une bonne méthode pour comprendre la logique du flux normal est de rendre les boîtes visibles. Nous pouvons commencer avec une page très basique, ayant seulement trois éléments `div` séparés, chacun ayant un paragraphe avec du texte aléatoire :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>CSS Box Model and Layout</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

<div id="first">
  <h2>First div</h2>
  <p><span>Sed</span> <span>eget</span> <span>velit</span>
  <span>id</span> <span>ante</span> <span>tempus</span>
  <span>porta</span> <span>pulvinar</span> <span>et</span>
  <span>ex.</span></p>
</div><!-- #first -->

<div id="second">
  <h2>Second div</h2>
  <p><span>Fusce</span> <span>vitae</span> <span>vehicula</span>
  <span>neque.</span> <span>Etiam</span> <span>maximus</span>
  <span>vulputate</span> <span>neque</span> <span>eu</span>
  <span>lobortis.</span> <span>Phasellus</span> <span>condimentum,</span>
  <span>felis</span> <span>eget</span> <span>eleifend</span>
  <span>aliquam,</span> <span>dui</span> <span>dolor</span>
  <span>bibendum</span> <span>leo.</span></p>
</div><!-- #second -->

<div id="third">
  <h2>Third div</h2>
  <p><span>Pellentesque</span> <span>ornare</span> <span>ultrices</span>
  <span>elementum.</span> <span>Morbi</span> <span>vulputate</span>
  <span>pretium</span> <span>arcu,</span> <span>sed</span>
  <span>faucibus.</span></p>
</div><!-- #third -->
```

```
</body>
</html>
```

Chaque mot se trouve dans un élément `span`, nous pouvons donc donner un style aux mots et voir comment ils sont traités comme des boîtes. Pour rendre les boîtes visibles, nous devons modifier le fichier de feuille de style `style.css` référencé par le document HTML. Les règles suivantes feront ce dont nous avons besoin :

```
* {
  font-family: sans;
  font-size: 14pt;
}

div {
  border: 2px solid #00000044;
}

#first {
  background-color: #c4a000ff;
}

#second {
  background-color: #4e9a06ff;
}

#third {
  background-color: #5c3566da;
}

h2 {
  background-color: #ffffff66;
}

p {
  background-color: #ffffff66;
}

span {
  background-color: #ffffffaa;
}
```

Le résultat apparaît dans la [Figure 35](#).

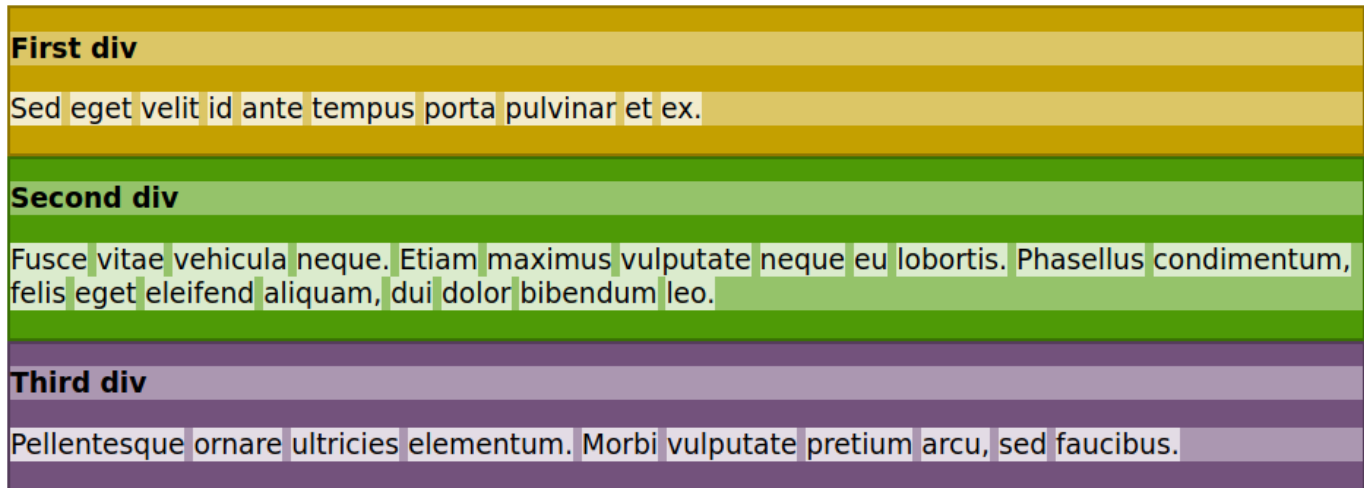


Figure 35. Le flux des éléments basiques se fait de haut en bas et de gauche à droite.

La [Figure 35](#) montre que chaque balise HTML a une case correspondante dans la mise en page. Les éléments `div`, `h2` et `p` s'étendent à la largeur de leur élément parent. Par exemple, l'élément parent des éléments `div` est l'élément `body`, ils s'étendent donc sur la largeur du corps `body`, tandis que le parent de chaque élément `h2` et `p` est l'élément `div` correspondant. Les boîtes qui s'étendent sur la largeur de leur élément parent sont appelées des éléments de *bloc*. Parmi les balises HTML les plus courantes rendues sous forme de blocs figurent les éléments `h1`, `h2`, `h3`, `p`, `ul`, `ol`, `table`, `li`, `div`, `section`, `form` et `aside`. Les éléments de bloc frères—éléments de bloc partageant le même élément parent immédiat—sont empilés dans leur parent de haut en bas.

NOTE

Certains éléments de bloc ne sont pas destinés à être utilisés comme conteneurs pour d'autres éléments de bloc. Il est possible, par exemple, d'insérer un élément de bloc à l'intérieur d'un élément `h1` ou `p`, mais ce n'est pas considéré comme une bonne pratique. Le concepteur doit plutôt utiliser une balise appropriée comme conteneur. Les balises de conteneur courantes sont `div`, `section` et `aside`.

Mis à part le texte lui-même, des éléments tels que `h1`, `p` et `li` n'attendent que des éléments *en ligne* (*inline*) comme enfants. Comme la plupart des scripts occidentaux, les éléments en ligne suivent le flux de gauche à droite du texte. Lorsqu'il ne reste plus de place dans la partie droite, le flux des éléments en ligne continue sur la ligne suivante, tout comme le texte. Certaines balises HTML courantes traitées comme des éléments en ligne sont `span`, `a`, `em`, `strong`, `img`, `input`, et `label`.

Dans notre exemple de page HTML, chaque mot à l'intérieur des paragraphes était entouré d'une balise `span`, afin qu'ils puissent être mis en évidence avec une règle CSS correspondante. Comme le montre l'image, chaque élément `span` est placé horizontalement, de gauche à droite, jusqu'à ce qu'il n'y ait plus de place dans l'élément parent.

La hauteur de l'élément dépend de son contenu, de sorte que le navigateur ajuste la hauteur d'un

élément conteneur pour s'adapter à ses éléments de bloc imbriqués ou à ses lignes d'éléments en ligne. Cependant, certaines propriétés CSS affectent la forme d'une boîte, sa position et le placement de ses éléments internes.

Les propriétés `margin` et `padding` affectent tous les types de boîtes. Si vous ne définissez pas ces propriétés explicitement, le navigateur en définit certaines en utilisant des valeurs standard. Comme on le voit dans la [Figure 35](#), les éléments `h2` et `p` ont été affichés avec un espace entre eux. Ces écarts sont les marges supérieure et inférieure que le navigateur ajoute par défaut à ces éléments. Nous pouvons les supprimer en modifiant les règles CSS pour les sélecteurs `h2` et `p` :

```
h2 {  
  background-color: #ffffff66;  
  margin: 0;  
}  
  
p {  
  background-color: #ffffff66;  
  margin: 0;  
}
```

Le résultat apparaît dans la [Figure 36](#).

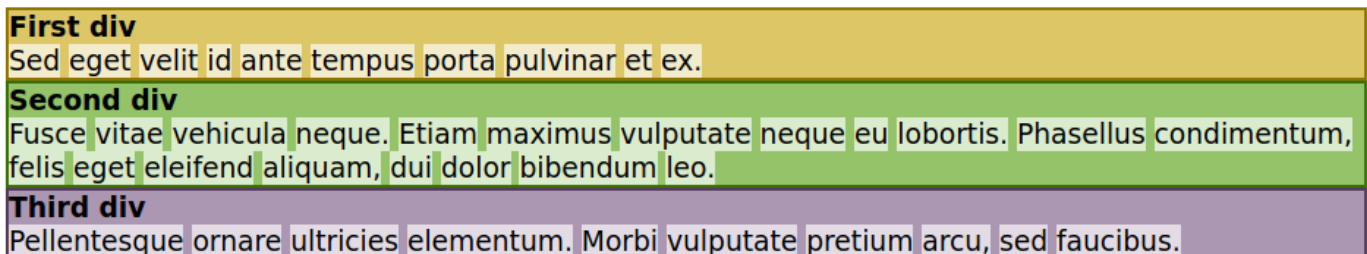


Figure 36. La propriété `margin` permet de modifier ou de supprimer les marges des éléments.

L'élément `body` a aussi, par défaut, une petite marge qui crée un espace tout autour. Cet espace peut également être supprimé à l'aide de la propriété `margin`.

Alors que la propriété `margin` définit l'écart entre l'élément et son environnement, la propriété `padding` de l'élément définit l'écart interne entre les limites du conteneur et ses éléments enfants. Prenons par exemple les éléments `h2` et `p` à l'intérieur de chaque `div` dans notre exemple de code. Nous pourrions utiliser leur propriété `margin` pour créer un écart par rapport aux limites de la `div` correspondante, mais il est plus simple de modifier la propriété `padding` du conteneur :

```
#second {
```

```
background-color: #4e9a06ff;
padding: 1em;
}
```

Seule la règle pour le deuxième `div` a été modifiée, de sorte que les résultats (Figure 37) montrent la différence entre le deuxième `div` et les autres conteneurs `div`.

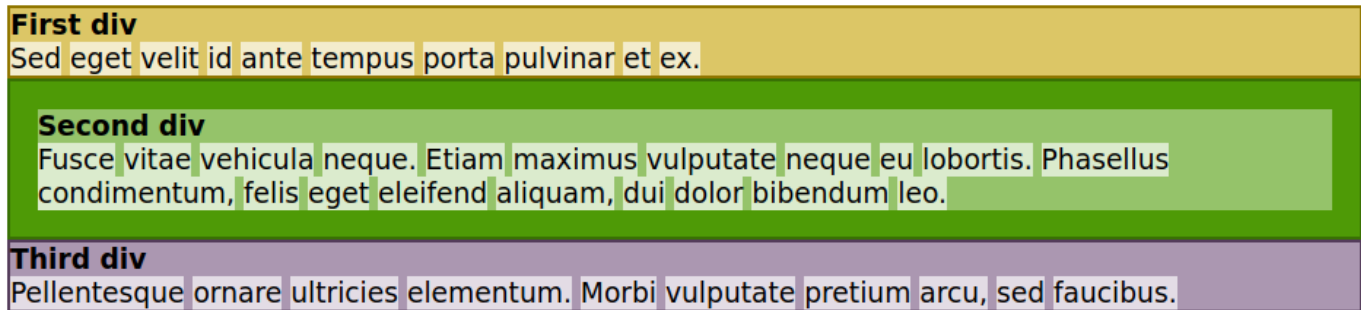


Figure 37. Des conteneurs `div` différents peuvent avoir des padding différents.

La propriété `margin` est un raccourci pour quatre propriétés contrôlant les quatre côtés de la boîte : `margin-top`, `margin-right`, `margin-bottom`, et `margin-left`. Lorsque l'on attribue une seule valeur à `margin`, comme dans les exemples précédents, les quatre marges de la boîte l'utilisent. Lorsque deux valeurs sont inscrites, la première définit les marges supérieure et inférieure, tandis que la seconde définit les marges droite et gauche. En utilisant `margin: 1em 2em`, par exemple, on définit un espace de 1 em pour les marges supérieure et inférieure et un espace de 2 em pour les marges droite et gauche. L'écriture de quatre valeurs définit les marges des quatre côtés dans le sens des aiguilles d'une montre, en commençant par le haut. Les différentes valeurs de la propriété raccourcie ne sont pas tenues d'utiliser les mêmes unités.

La propriété `padding` est également un raccourci, suivant les mêmes principes que la propriété `margin`.

Dans leur comportement par défaut, les éléments de bloc s'étirent pour s'adapter à la largeur disponible. Mais ce n'est pas obligatoire. La propriété `width` permet de définir une taille horizontale fixe pour le bloc :

```
#first {
  background-color: #c4a000ff;
  width: 6em;
}
```

L'ajout de `width: 6em` à la règle CSS rétrécit le premier `div` horizontalement, laissant un espace vide

sur son côté droit ([Figure 38](#)).

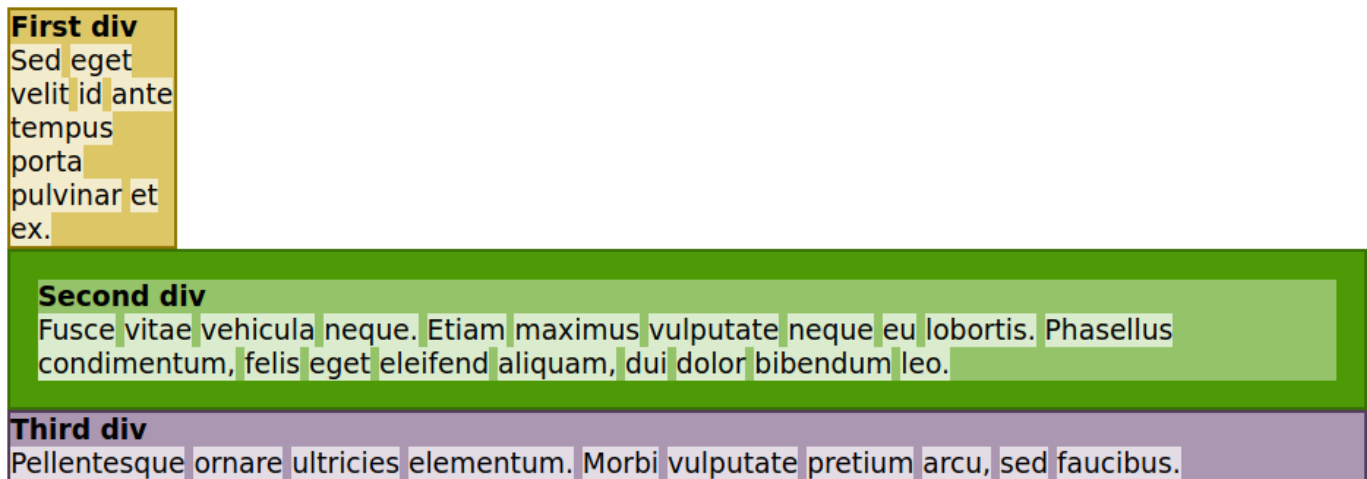


Figure 38. La propriété `width` modifie la largeur horizontale du premier `div`.

Au lieu de laisser le premier `div` aligné sur la gauche, nous pouvons souhaiter le centrer. Centrer une boîte revient à définir des marges de même taille des deux côtés, nous pouvons donc utiliser la propriété `margin` pour la centrer. La taille de l'espace disponible pouvant varier, nous utilisons la valeur `auto` pour les marges gauche et droite :

```
#first {
  background-color: #c4a000ff;
  width: 6em;
  margin: 0 auto;
}
```

Les marges gauche et droite sont automatiquement calculées par le navigateur et la boîte sera centrée ([Figure 39](#)).

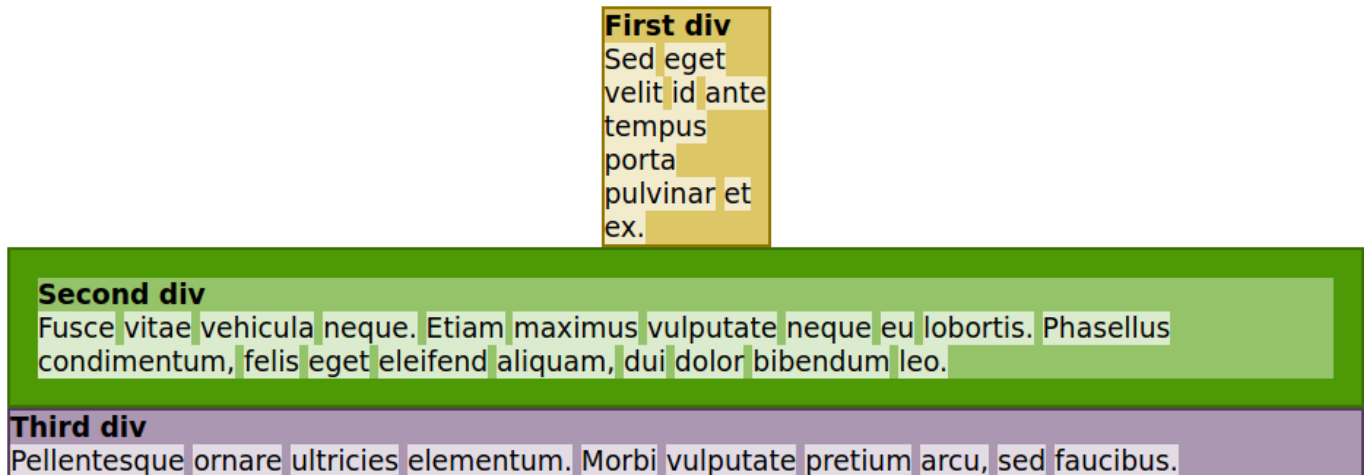


Figure 39. La propriété `margin` est utilisée pour centrer le premier `div`.

Comme vous pouvez le constater, le fait de rendre un élément de bloc plus étroit ne rend pas l'espace restant disponible pour l'élément suivant. Le flux normal est toujours préservé, comme si l'élément plus étroit occupait toujours toute la largeur disponible.

Personnalisation du flux normal

Le flux normal est simple et séquentiel. CSS vous permet également de rompre le flux normal et de positionner les éléments de manière très spécifique, voire même d'ignorer le défilement de la page si vous le souhaitez. Nous allons examiner dans cette section plusieurs façons de contrôler le positionnement des éléments.

Éléments flottants

Il est possible de faire en sorte que des éléments de bloc frères partagent le même espace horizontal. L'un des moyens d'y parvenir est d'utiliser la propriété `float`, qui retire l'élément du flux normal. Comme son nom l'indique, la propriété `float` fait flotter la boîte au-dessus des éléments de bloc qui suivent, de sorte qu'ils seront affichés comme s'ils étaient sous la boîte flottante. Pour faire flotter le premier `div` vers la droite, ajoutez `float: right` à la règle CSS correspondante :

```
#first {
  background-color: #c4a000ff;
  width: 6em;
  float: right;
}
```

Les marges automatiques sont ignorées dans une boîte flottante, donc la propriété `margin` peut être supprimée. La [Figure 40](#) montre le résultat obtenu en faisant flotter le premier `div` vers la droite.

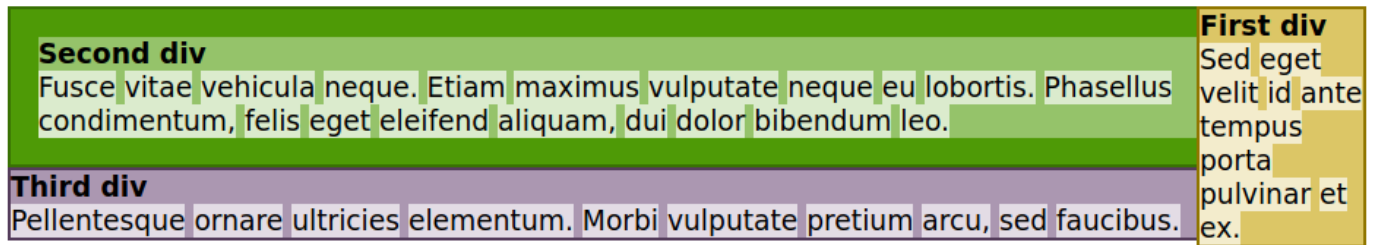


Figure 40. Le premier `div` est flottant et ne fait pas partie du flux normal.

Par défaut, tous les éléments de bloc venant après l'élément flottant passeront sous celui-ci. Par conséquent, si la hauteur est suffisante, la boîte flottante couvrira tous les éléments de bloc restants.

Bien qu'un élément flottant se place au-dessus des autres éléments du bloc, le contenu en ligne à l'intérieur du conteneur de l'élément flottant s'enroule autour de l'élément flottant. Ce principe s'inspire de la mise en page des magazines et des journaux, qui enroulent souvent le texte autour d'une image, par exemple.

L'image précédente montre comment le premier `div` couvre le deuxième `div` et une partie du troisième `div`. Supposons que nous voulions que le premier `div` flotte au-dessus du deuxième `div`, mais pas du troisième. La solution consiste à inclure la propriété `clear` dans la règle CSS correspondant au troisième `div` :

```
#third {  
  background-color: #5c3566da;  
  clear: right;  
}
```

En définissant la propriété `clear` à `right`, l'élément correspondant saute tous les éléments flottant à droite précédents, reprenant le flux normal (Figure 41).

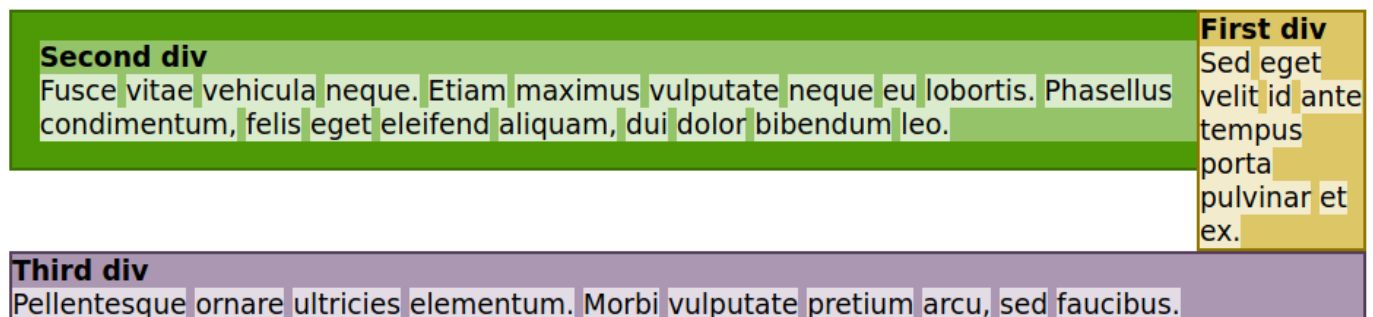


Figure 41. La propriété `clear` retourne au flux normal.

De même, si un élément précédent flottait sur la gauche, vous pouvez utiliser `clear: left` pour

reprendre le flux normal. Si vous devez ignorer des éléments flottants à gauche et à droite, utilisez `clear: both`.

Positionnement des boîtes

Dans un flux normal, chaque boîte va après les boîtes qui la précèdent dans l'arbre du document. Les éléments frères précédents "poussent" ("push") les éléments qui les suivent, les déplaçant vers la droite et vers le bas à l'intérieur de leur élément parent. L'élément parent peut avoir ses propres frères qui lui font la même chose. C'est comme placer des tuiles côte à côte dans un mur, en commençant par le haut.

Cette méthode de positionnement des boîtes est appelée *statique* et constitue la valeur par défaut de la propriété CSS `position`. En dehors de la définition des marges et du remplissage, il n'existe aucun moyen de repositionner une boîte statique dans la page.

Comme pour l'analogie des tuiles dans le mur, le placement statique n'est pas obligatoire. Comme pour les tuiles, vous pouvez placer les boîtes où vous voulez, même en couvrant d'autres boîtes. Pour ce faire, attribuez à la propriété `position` l'une des valeurs suivantes :

relative

L'élément suit le flux normal du document, mais il peut utiliser les propriétés `top`, `right`, `bottom` et `left` pour définir des décalages par rapport à sa position statique initiale. Ces décalages peuvent également être négatifs. Les autres éléments restent à leur place initiale, comme si l'élément relatif était toujours statique.

absolute

L'élément ignore le flux normal des autres éléments et se positionne sur la page grâce aux propriétés `top`, `right`, `bottom`, et `left`. Leurs valeurs sont relatives au corps du document ou à un conteneur parent non statique.

fixed

L'élément ignore le flux normal des autres éléments et se positionne grâce aux propriétés `top`, `right`, `bottom` et `left`. Leurs valeurs sont relatives à la fenêtre d'affichage (c'est-à-dire la zone de l'écran où le document est affiché). Les éléments fixes ne se déplacent pas lorsque le visiteur fait défiler le document, mais ressemblent à un autocollant fixé à l'écran.

sticky

L'élément suit le flux normal du document. Cependant, au lieu de sortir de la fenêtre d'affichage lorsque le document défile, il s'arrête à la position définie par les propriétés `top`, `right`, `bottom` et `left`. Si la valeur `top` est `10px`, par exemple, l'élément arrêtera de défiler sous la partie supérieure de la fenêtre lorsqu'il atteindra 10 pixels de la limite supérieure de la fenêtre. Lorsque cela se

produit, le reste de la page continue de défiler, mais l'élément collant se comporte comme un élément fixe dans cette position. Il reprendra sa position initiale lorsque le document défilera de nouveau jusqu'à sa position dans la fenêtre d'affichage. Les éléments collants sont couramment utilisés de nos jours pour créer des menus supérieurs qui restent toujours visibles.

Les positions qui peuvent utiliser les propriétés `top`, `right`, `bottom` et `left` ne sont pas obligées de les utiliser toutes. Si vous définissez à la fois les propriétés `top` et `height` d'un élément absolu, par exemple, le navigateur calcule implicitement sa propriété `bottom` (`top + height = bottom`).

Propriétés display

Si l'ordre donné par le flux normal n'est pas un problème dans votre conception, mais que vous voulez changer la façon dont les boîtes s'alignent dans la page, modifiez la propriété `display` de l'élément. La propriété `display` peut même faire disparaître complètement l'élément du document rendu, en définissant `display: none`. Ceci peut être utile lorsque vous souhaitez afficher l'élément plus tard en utilisant JavaScript.

La propriété `display` peut aussi, par exemple, faire en sorte qu'un élément en bloc se comporte comme un élément en ligne (`display: inline`). Ce n'est toutefois pas une bonne pratique. Il existe de meilleures méthodes pour placer les éléments conteneurs côte à côte, comme le *modèle de boîte flexible* (*flexbox model*).

Le modèle de boîte flexible a été inventé pour surmonter les limites des éléments flottants et pour éliminer l'utilisation inappropriée des tableaux pour structurer la mise en page. Lorsque vous définissez la propriété `display` d'un élément conteneur sur `flex` pour le transformer en conteneur *flexbox*, ses enfants immédiats se comporteront plus ou moins comme les cellules d'une rangée de tableau.

TIP

Si vous voulez encore mieux contrôler le placement des éléments sur la page, jetez un coup d'œil à la fonctionnalité de grille *CSS grid*. La grille est un système puissant basé sur des lignes et des colonnes permettant de créer des mises en page élaborées.

Pour tester l'affichage flexible, ajoutez un nouvel élément `div` à la page d'exemple et faites-en le conteneur des trois éléments `div` existants :

```
<div id="container">

<div id="first">
  <h2>First div</h2>
  <p><span>Sed</span> <span>eget</span> <span>velit</span>
  <span>id</span> <span>ante</span> <span>tempus</span>
  <span>porta</span> <span>pulvinar</span> <span>et</span>
```



```

    <span>ex.</span></p>
</div><!-- #first -->

<div id="second">
  <h2>Second div</h2>
  <p><span>Fusce</span> <span>vitae</span> <span>vehicula</span>
  <span>neque.</span> <span>Etiam</span> <span>maximus</span>
  <span>vulputate</span> <span>neque</span> <span>eu</span>
  <span>lobortis.</span> <span>Phasellus</span> <span>condimentum,</span>
  <span>felis</span> <span>eget</span> <span>eleifend</span>
  <span>aliquam,</span> <span>dui</span> <span>dolor</span>
  <span>bibendum</span> <span>leo.</span></p>
</div><!-- #second -->

<div id="third">
  <h2>Third div</h2>
  <p><span>Pellentesque</span> <span>ornare</span> <span>ultrices</span>
  <span>elementum.</span> <span>Morbi</span> <span>vulputate</span>
  <span>pretium</span> <span>arcu,</span> <span>sed</span>
  <span>faucibus.</span></p>
</div><!-- #third -->

</div><!-- #container -->

```

Ajoutez la règle CSS suivante à la feuille de style pour transformer le conteneur `div` en conteneur flexible *flexbox* :

```

#container {
  display: flex;
}

```

Le résultat est un affichage côte à côte des trois éléments internes `div` ([Figure 42](#)).

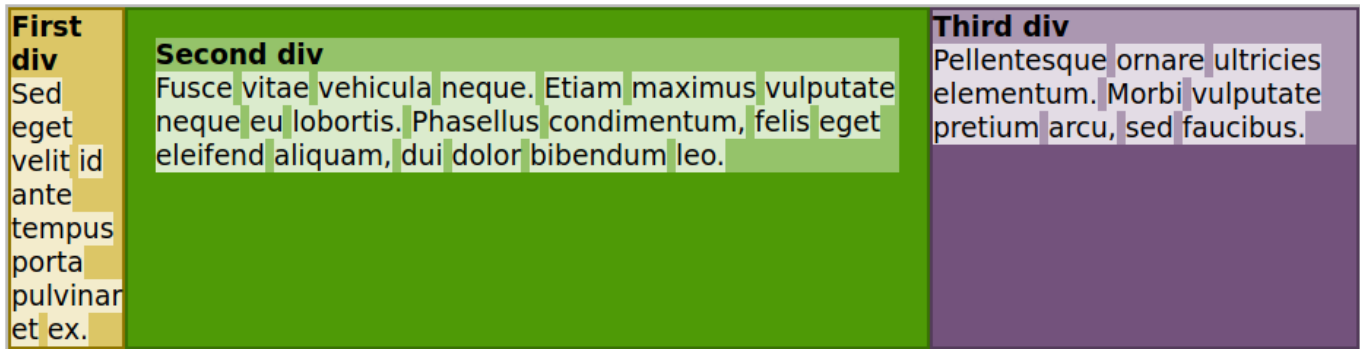


Figure 42. Le modèle de boîte flexible crée une grille.

L'utilisation de la valeur `inline-flex` au lieu de `flex` a fondamentalement le même résultat, mais fait que les enfants se comportent plus comme des éléments en ligne.

Conception réactive

Nous savons que le CSS fournit des propriétés qui permettent d'ajuster les tailles des éléments et des polices par rapport à la surface d'écran disponible. Cependant, vous pouvez aller plus loin et utiliser un design différent selon les appareils : par exemple, les systèmes de bureau par rapport aux appareils dont les dimensions de l'écran sont inférieures à une certaine taille. Cette approche est appelée *conception web réactive* (*responsive web design*), et CSS fournit des méthodes appelées *requêtes média* pour la rendre possible.

Dans l'exemple précédent, nous avons modifié la mise en page pour placer les éléments `div` côte à côte dans des colonnes. Cette mise en page convient aux grands écrans, mais elle sera trop encombrante sur les petits écrans. Pour résoudre ce problème, nous pouvons ajouter une requête média à la feuille de style qui ne correspond qu'aux écrans d'une largeur d'au moins 600px :

```
@media (min-width: 600px){
  #container {
    display: flex;
  }
}
```

Les règles CSS contenues dans la directive `@media` ne seront utilisées que si le critère entre parenthèses est satisfait. Dans cet exemple, si la largeur de la fenêtre d'affichage est inférieure à 600px, la règle ne sera pas appliquée au conteneur `div` et ses enfants seront rendus comme des éléments `div` classiques. Le navigateur réévalue les requêtes média chaque fois que la dimension de la fenêtre d'affichage change, de sorte que la mise en page peut être modifiée en temps réel lors du redimensionnement de la fenêtre du navigateur ou de la rotation du smartphone.

Exercices guidés

1. Si la propriété `position` n'est pas modifiée, quelle méthode de positionnement sera utilisée par le navigateur ?

2. Comment s'assurer que la boîte d'un élément sera affichée après les éléments flottants précédents ?

3. Comment utiliser la propriété raccourcie `margin` pour définir les marges supérieure et inférieure à 4px et les marges droite et gauche à 6em ?

4. Comment centrer horizontalement un élément conteneur statique à largeur fixe sur la page ?

Exercices d'exploration

1. Écrire une règle CSS correspondant à l'élément `<div class="picture">` pour que le texte à l'intérieur de ses éléments de bloc suivants se dépose vers son côté droit.

2. Comment la propriété `top` affecte-t-elle un élément statique par rapport à son élément parent ?

3. Comment le fait de changer la propriété `display` d'un élément en `flex` affecte-t-il son placement dans le flux normal ?

4. Quelle fonctionnalité CSS vous permet d'utiliser un ensemble de règles distinct en fonction des dimensions de l'écran ?

Résumé

Cette leçon traite du modèle de boîte CSS et de la façon dont nous pouvons le personnaliser. En plus du flux normal du document, le concepteur peut utiliser différents mécanismes de positionnement pour mettre en œuvre une mise en page personnalisée. La leçon aborde les concepts et procédures suivants :

- Le flux normal du document.
- Ajustements de la marge et du padding de la boîte d'un élément.
- Utilisation des propriétés `float` et `clear`.
- Les mécanismes de positionnement : statique, relatif, absolu, fixe et collant.
- Valeurs alternatives pour la propriété `display`.
- Les bases du design réactif.

Réponses aux exercices guidés

1. Si la propriété `position` n'est pas modifiée, quelle méthode de positionnement sera utilisée par le navigateur ?

La méthode `static`.

2. Comment s'assurer que la boîte d'un élément sera affichée après les éléments flottants précédents ?

La propriété `clear` de l'élément doit être définie à `both`.

3. Comment utiliser la propriété raccourcie `margin` pour définir les marges supérieure et inférieure à `4px` et les marges droite et gauche à `6em` ?

Elle peut être soit `margin: 4px 6em` ou `margin: 4px 6em 4px 6em`.

4. Comment centrer horizontalement un élément conteneur statique à largeur fixe sur la page ?

En utilisant la valeur `auto` dans ses propriétés `margin-left` et `margin-right`.

Réponses aux exercices d'exploration

1. Écrire une règle CSS correspondant à l'élément `<div class="picture">` pour que le texte à l'intérieur de ses éléments de bloc suivants se dépose vers son côté droit.

```
.picture { float: left; }
```

2. Comment la propriété `top` affecte-t-elle un élément statique par rapport à son élément parent ?

La propriété `top` ne s'applique pas aux éléments statiques.

3. Comment le fait de changer la propriété `display` d'un élément en `flex` affecte-t-il son placement dans le flux normal ?

Le placement de l'élément lui-même ne change pas, mais ses éléments enfants immédiats seront affichés côte à côte horizontalement.

4. Quelle fonctionnalité CSS vous permet d'utiliser un ensemble de règles distinct en fonction des dimensions de l'écran ?

Les *requêtes média* permettent au navigateur de vérifier les dimensions de la fenêtre d'affichage avant d'appliquer une règle CSS.



Thème 034: Programmation JavaScript



**Linux
Professional
Institute**

034.1 Exécution et syntaxe JavaScript

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 034.1

Valeur

1

Domaines de connaissance les plus importants

- Exécuter JavaScript dans un document HTML
- Comprendre la syntaxe JavaScript
- Ajouter des commentaires au code JavaScript
- Accéder à la console JavaScript
- Écrire dans la console JavaScript

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `<script>`, y compris les attributs `type (text/javascript)` et `src`
- `;`
- `//`, `/* */`
- `console.log`



034.1 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	034 Programmation JavaScript
Objectif :	034.1 Exécution et syntaxe JavaScript
Leçon :	1 sur 1

Introduction

Les pages web sont développées à l'aide de trois technologies standard : HTML, CSS et JavaScript. JavaScript est un langage de programmation qui permet au navigateur de mettre à jour dynamiquement le contenu du site web. Le JavaScript est généralement exécuté par le même navigateur que celui utilisé pour afficher une page web. Cela signifie que, comme pour CSS et HTML, le comportement exact du code que vous écrivez peut varier d'un navigateur à l'autre. Mais les navigateurs les plus courants adhèrent à la norme ECMAScript. Il s'agit d'une norme qui unifie l'utilisation de JavaScript sur le web, et qui servira de base à la leçon, avec la spécification HTML5, qui spécifie comment JavaScript doit être placé dans une page Web pour qu'un navigateur l'exécute.

Exécution de JavaScript dans le navigateur

Pour exécuter JavaScript, le navigateur doit obtenir le code soit directement, comme une partie du code HTML qui compose la page web, soit sous la forme d'une URL qui indique l'emplacement d'un script à exécuter.

L'exemple suivant montre comment inclure du code directement dans le fichier HTML :

```
<html>
  <head>
  </head>
  <body>
    <h1>Website Headline</h1>
    <p>Content</p>

    <script>
      console.log('test');
    </script>

  </body>
</html>
```

Le code est enveloppé entre les balises `<script>` et `</script>`. Tout ce qui est inclus dans ces balises sera exécuté par le navigateur directement lors du chargement de la page.

La position de l'élément `<script>` dans la page détermine le moment où il sera exécuté. Un document HTML est analysé de haut en bas, le navigateur décidant quand afficher les éléments à l'écran. Dans l'exemple qui vient d'être montré, les balises `<h1>` et `<p>` du site web sont analysées, et probablement affichées, avant l'exécution du script. Si le code JavaScript contenu dans la balise `<script>` prenait beaucoup de temps à s'exécuter, la page s'afficherait quand même sans problème. En revanche, si le script avait été placé au-dessus des autres balises, le visiteur de la page web devrait attendre la fin de l'exécution du script pour voir la page. Pour cette raison, les balises `<script>` sont généralement placées à l'un des deux endroits suivants :

- À la toute fin du corps du HTML, de sorte que le script soit la dernière chose à être exécutée. Faites-le lorsque le code ajoute quelque chose à la page qui ne serait pas utile sans le contenu. Un exemple serait d'ajouter une fonctionnalité à un bouton, car le bouton doit exister pour que la fonctionnalité ait un sens.
- A l'intérieur de l'élément `<head>` du HTML. Cela garantit que le script sera exécuté avant que le corps du HTML ne soit analysé. Si vous voulez modifier le comportement de chargement de la page, ou si vous avez quelque chose qui doit être exécuté alors que la page n'est pas encore complètement chargée, vous pouvez placer le script ici. De même, si vous avez plusieurs scripts qui dépendent d'un script particulier, vous pouvez placer ce script partagé dans l'en-tête pour vous assurer qu'il est exécuté avant les autres scripts.

Pour diverses raisons, notamment la facilité de gestion, il est utile de placer votre code JavaScript dans des fichiers distincts qui existent en dehors de votre code HTML. Les fichiers JavaScript externes sont inclus en utilisant une balise `<script>` avec un attribut `src`, comme suit :

```
<html>
  <head>
    <script src="/button-interaction.js"></script>
  </head>
  <body>
  </body>
</html>
```

La balise `src` indique au navigateur l'emplacement de la source, c'est-à-dire le fichier contenant le code JavaScript. L'emplacement peut être un fichier sur le même serveur, comme dans l'exemple ci-dessus, ou toute URL accessible sur le Web, comme <https://www.lpi.org/example.js>. La valeur de l'attribut `src` suit la même convention que l'importation de fichiers CSS ou d'images, en ce sens qu'elle peut être relative ou absolue. Lorsqu'il rencontre une balise de `script` avec l'attribut `src`, le navigateur tente alors d'obtenir le fichier source en utilisant une requête HTTP GET, les fichiers externes doivent donc être accessibles.

Lorsque vous utilisez l'attribut `src`, tout code ou texte placé entre les balises `<script>...</script>` est ignoré, conformément à la spécification HTML.

```
<html>
  <head>
    <script src="/button-interaction.js">
      console.log("test"); // <-- This is ignored
    </script>
  </head>
  <body>
  </body>
</html>
```

Il existe d'autres attributs que vous pouvez ajouter à la balise `script` pour spécifier davantage comment le navigateur doit obtenir les fichiers, et comment il doit les traiter ensuite. La liste suivante détaille les attributs les plus importants :

async

Peut être utilisé sur les balises `script` et donne l'instruction au navigateur d'aller chercher le script en arrière-plan, afin de ne pas bloquer le processus de chargement de la page. Le chargement de la page sera toujours interrompu après que le navigateur ait obtenu le script, parce que le navigateur doit l'analyser, ce qui est fait immédiatement après que le script ait été récupéré complètement. Cet attribut est booléen, donc il suffit d'écrire la balise comme `<script async src="/script.js"></script>` et aucune valeur ne doit être fournie.

defer

Comme `async`, cet attribut indique au navigateur de ne pas bloquer le processus de chargement de la page pendant la récupération du script. Au contraire, le navigateur va différer l'analyse du script. Le navigateur attendra que l'ensemble du document HTML ait été analysé et ce n'est qu'alors qu'il analysera le script, avant d'annoncer que le document a été complètement chargé. Comme `async`, `defer` est un attribut booléen et s'utilise de la même manière. Comme `defer` implique `async`, il n'est pas utile de spécifier les deux balises ensemble.

NOTE

Lorsqu'une page est complètement analysée, le navigateur indique qu'elle est prête à être affichée en déclenchant un événement `DOMContentLoaded`, lorsque le visiteur pourra voir le document. Ainsi, le JavaScript inclus dans un événement `<head>` a toujours une chance d'agir sur la page avant son affichage, même si vous incluez l'attribut `defer`.

type

Indique le type de script que le navigateur doit attendre dans la balise. La valeur par défaut est JavaScript (`type="application/javascript"`). Cet attribut n'est donc pas nécessaire lorsque vous incluez du code JavaScript ou que vous pointez vers une ressource JavaScript avec la balise `src`. En général, tous les types MIME peuvent être spécifiés, mais seuls les scripts désignés comme JavaScript seront exécutés par le navigateur. Il existe deux cas d'utilisation réalistes pour cet attribut : indiquer au navigateur de ne pas exécuter le script en définissant `type` à une valeur arbitraire comme `template` ou `other`, ou indiquer au navigateur que le script est un module ES6. Nous ne couvrirons pas les modules ES6 dans cette leçon.

WARNING

Lorsque plusieurs scripts ont l'attribut `async`, ils seront exécutés dans l'ordre de leur téléchargement, *non pas* dans l'ordre des balises `script` dans le document. L'attribut `defer`, quant à lui, maintient l'ordre des balises `script`.

Console du navigateur

Bien qu'il soit généralement exécuté dans le cadre d'un site web, il existe un autre moyen d'exécuter JavaScript : via la console du navigateur. Tous les navigateurs de bureau modernes proposent un menu permettant d'exécuter du code JavaScript dans leur moteur. Cette opération sert généralement à tester un nouveau code ou à déboguer des sites web existants.

Il existe plusieurs façons d'accéder à la console du navigateur, en fonction du navigateur. La méthode la plus simple consiste à utiliser les raccourcis clavier. Vous trouverez ci-dessous les raccourcis clavier pour certains des navigateurs actuellement utilisés :

Chrome

Ctrl + Shift + J (Cmd + Option + J sur Mac)

Firefox

Ctrl + Shift + K (Cmd + Option + K sur Mac)

Safari

Ctrl + Shift + ? (Cmd + Option + ? sur Mac)

Vous pouvez également cliquer avec le bouton droit de la souris sur une page web et sélectionner l'option "Inspector" ou "Inspecter l'élément" pour ouvrir l'inspecteur, qui est un autre outil du navigateur. Lorsque l'inspecteur s'ouvre, un nouveau panneau apparaît. Dans ce panneau, vous pouvez sélectionner l'onglet "Console", qui fera apparaître la console du navigateur.

Une fois la console affichée, vous pouvez exécuter du JavaScript sur la page en le saisissant directement dans le champ de saisie. Le résultat de tout code exécuté s'affichera sur une ligne distincte.

Instructions JavaScript

Maintenant que nous savons comment lancer l'exécution d'un script, nous allons aborder les bases de l'exécution d'un script. Un script JavaScript est un ensemble d'instructions et de blocs. Un exemple d'instruction est `console.log('test')`. Cette instruction demande au navigateur d'afficher le mot `test` dans la console du navigateur.

Chaque instruction en JavaScript se termine par un point-virgule (;). Cela indique au navigateur que cette instruction est terminée et qu'une nouvelle peut être lancée. Considérons le script suivant :

```
var message = "test"; console.log(message);
```

Nous avons écrit deux instructions. Chaque instruction est terminée soit par un point-virgule, soit par la fin du script. Pour des raisons de lisibilité, nous pouvons placer les instructions sur des lignes séparées. De cette façon, le script pourrait aussi être écrit comme suit :

```
var message = "test";  
console.log(message);
```

Cela est possible parce que tous les caractères blancs entre les instructions, tels qu'un espace, une nouvelle ligne ou une tabulation, sont ignorés. Les caractères blancs peuvent aussi souvent être placés entre les mots-clés individuels dans les instructions, mais cela sera expliqué plus en détail dans une prochaine leçon. Les instructions peuvent également être vides, ou composées uniquement d'espaces.

Si une instruction est invalide parce qu'elle n'a pas été terminée par un point-virgule, ECMAScript tente d'insérer automatiquement les points-virgules appropriés, en se basant sur un ensemble complexe de règles. La règle la plus importante est la suivante : Si une instruction invalide est composée de deux instructions valides séparées par une nouvelle ligne, insérez un point-virgule à la nouvelle ligne. Par exemple, le code suivant ne forme pas une déclaration valide :

```
console.log("hel lo")
console.log("wor ld")
```

Mais un navigateur moderne l'exécutera automatiquement comme s'il avait été écrit avec les points-virgules appropriés :

```
console.log("hel lo");
console.log("wor ld");
```

Il est donc possible d'omettre les points-virgules dans certains cas. Toutefois, comme les règles d'insertion automatique des points-virgules sont complexes, nous vous conseillons de toujours terminer correctement vos instructions pour éviter les erreurs indésirables.

Commentaires JavaScript

Les scripts volumineux peuvent devenir assez complexes. Vous voudrez peut-être commenter ce que vous écrivez, afin de rendre le script plus facile à lire pour d'autres personnes, ou pour vous-même à l'avenir. Vous pouvez également inclure des méta-informations dans le script, telles que des informations sur les droits d'auteur ou sur la date et la raison de la rédaction du script.

Pour permettre d'inclure de telles méta-informations, JavaScript prend en charge les *commentaires*. Un développeur peut inclure des caractères spéciaux dans un script afin de désigner certaines parties de celui-ci comme des commentaires, qui seront ignorés lors de l'exécution. Le texte suivant est une version fortement commentée du script que nous avons vu précédemment.

```
/*
  This script was written by the author of this lesson in May, 2020.
  It has exactly the same effect as the previous script, but includes comments.
*/

// First, we define a message.
var message = "test";
```

```
console.log(message); // Then, we output the message to the console.
```

Les commentaires ne sont pas des instructions et n'ont pas besoin d'être terminés par un point-virgule. Au lieu de cela, ils suivent leurs propres règles de fin, en fonction de la manière dont le commentaire est écrit. Il existe deux façons d'écrire des commentaires en JavaScript :

Commentaire sur plusieurs lignes

Utilisez `/*` et `*/` pour signaler le début et la fin d'un commentaire sur plusieurs lignes. Tout ce qui suit `/*`, jusqu'à la première occurrence de `*/` est ignoré. Ce type de commentaire est généralement utilisé pour couvrir plusieurs lignes, mais il peut aussi être utilisé pour des lignes simples, ou même à l'intérieur d'une ligne, comme ceci :

```
console.log(/* what we want to log: */ "hello world")
```

Comme le but des commentaires est généralement d'augmenter la lisibilité d'un script, vous devriez éviter d'utiliser ce style de commentaire à l'intérieur d'une ligne.

Commentaire sur une seule ligne

Utilisez `//` (deux barres obliques) pour *commenter* une ligne. Tout ce qui suit la double barre oblique sur la même ligne est ignoré. Dans l'exemple montré précédemment, ce motif est d'abord utilisé pour commenter une ligne entière. Après l'instruction `console.log(message);`, il est utilisé pour écrire un commentaire sur le reste de la ligne.

En général, les commentaires d'une seule ligne doivent être utilisés pour les lignes simples, et les commentaires de plusieurs lignes pour les lignes multiples, même s'il est possible de les utiliser d'une autre manière. Les commentaires à l'intérieur d'une instruction doivent être évités.

Les commentaires peuvent également être utilisés pour supprimer temporairement des lignes de code existantes, comme suit :

```
// We temporarily want to use a different message
// var message = "test";
var message = "something else";
```


Exercices guidés

1. Créez une variable appelée `ColorName` et affectez la valeur `RED` à celle-ci.

2. Lesquels des scripts suivants sont valides ?

<code>console.log("hello") console.log("world");</code>	
<code>console.log("hello"); console.log("world");`</code>	
<code>// console.log("hello") console.log("world");</code>	
<code>console.log("hello"); console.log("world") //;</code>	
<code>console.log("hello"); /* console.log("world") */</code>	

Exercices d'exploration

1. Combien d'instructions JavaScript peuvent être écrites sur une seule ligne sans utiliser de point-virgule ?

2. Créez deux variables nommées `x` et `y`, puis affichez leur somme sur la console.

Résumé

Dans cette leçon, nous avons appris différentes manières d'exécuter JavaScript, et comment modifier le comportement de chargement des scripts. Nous avons également appris les concepts de base de la composition et du commentaire de scripts, et avons appris à utiliser la commande `console.log()`.

HTML utilisé dans cette leçon :

`<script>`

La balise `script` peut être utilisée pour inclure JavaScript directement ou en spécifiant un fichier avec l'attribut `src`. Modifiez la façon dont le script est chargé avec les attributs `async` et `defer`.

Les concepts JavaScript présentés dans cette leçon :

`;`

Le point-virgule est utilisé pour séparer les instructions. Les points-virgules peuvent parfois — mais ne doivent pas — être omis.

`//, /*...*/`

Les commentaires peuvent être utilisés pour ajouter des explications ou des méta-informations à un fichier de script, ou pour empêcher l'exécution d'instructions.

`console.log("text")`

L'instruction `console.log()` peut être utilisée pour afficher du texte dans la console du navigateur.

Réponses aux exercices guidés

1. Créez une variable appelée `ColorName` et affectez la valeur `RED` à celle-ci.

```
var ColorName = "RED";
```

2. Lesquels des scripts suivants sont valides ?

<pre>console.log("hello") console.log("world");</pre>	Invalide : La première instruction <code>console.log()</code> n'est pas correctement terminée, et la ligne dans son ensemble ne forme pas une instruction valide.
<pre>console.log("hello"); console.log("world");</pre>	Valide : Chaque instruction est correctement terminée.
<pre>// console.log("hello") console.log("world");</pre>	Valide : Le code entier est ignoré car il s'agit d'un commentaire.
<pre>console.log("hello"); console.log("world") //;</pre>	Invalide : Il manque un point-virgule dans la dernière instruction. Le point-virgule à la toute fin est ignoré car il est commenté.
<pre>console.log("hello"); /* console.log("world") */</pre>	Valide : Une instruction valide est suivie d'un code commenté, qui est ignoré.

Réponses aux exercices d'exploration

1. Combien d'instructions JavaScript peuvent être écrites sur une seule ligne sans utiliser de point-virgule ?

Si nous sommes à la fin d'un script, nous pouvons écrire une instruction et elle sera terminée par la fin du fichier. Sinon, vous ne pouvez pas écrire une instruction sans point-virgule avec la syntaxe que vous avez apprise jusqu'à présent.

2. Créez deux variables nommées `x` et `y`, puis affichez leur somme sur la console.

```
var x = 5;  
var y = 10;  
console.log(x+y);
```



**Linux
Professional
Institute**

034.2 Structures de données JavaScript

Referencia al objetivo del LPI

Web Development Essentials version 1.0, Exam 030, Objective 034.2

Valeur

3

Domaines de connaissance les plus importants

- Définir et utiliser les variables et les constantes
- Comprendre les types de données
- Comprendre la conversion/coercition de types
- Comprendre les tableaux et les objets
- Connaissance de la portée des variables

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `=`, `+`, `-`, `*`, `/`, `%`, `--`, `++`, `+=`, `-=`, `*=`, `/=`
- `var`, `let`, `const`
- `boolean`, `number`, `string`, `symbol`
- `array`, `object`
- `undefined`, `null`, `NaN`



034.2 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	034 Programmation JavaScript
Objectif :	034.2 Structures de données JavaScript
Leçon:	1 sur 1

Introduction

Les langages de programmation, comme les langages naturels, représentent la réalité par des symboles qui sont combinés pour former des énoncés significatifs. La réalité représentée par un langage de programmation est constituée par les ressources de la machine, telles que les opérations du processeur, les périphériques et la mémoire.

Chacun des innombrables langages de programmation adopte un paradigme pour représenter les informations. JavaScript adopte les conventions typiques des langages de *haut niveau*, où la plupart des détails tels que l'allocation de mémoire sont implicites, ce qui permet au programmeur de se concentrer sur l'objectif du script dans le contexte de l'application.

Langages de haut niveau

Les langages de haut niveau fournissent des règles abstraites afin que le programmeur n'ait besoin que d'écrire le minimum de code pour exprimer une idée. JavaScript offre des moyens pratiques pour utiliser la mémoire de l'ordinateur, en utilisant des concepts de programmation qui simplifient l'écriture de routines récurrentes et qui sont généralement suffisants pour l'objectif du développeur web.

NOTE

Bien qu'il soit possible d'utiliser des mécanismes spécialisés pour un accès méticuleux à la mémoire, les types de données plus simples que nous allons examiner sont d'un usage plus général.

Les opérations typiques d'une application web consistent à demander des données par le biais d'une instruction JavaScript et à les stocker pour qu'elles soient traitées et finalement présentées à l'utilisateur. Ce stockage est assez flexible en JavaScript, avec des formats de stockage appropriés pour chaque objectif.

Déclaration des constantes et des variables

La déclaration des constantes et des variables pour contenir des données est la pierre angulaire de tout langage de programmation. JavaScript adopte la convention de la plupart des langages de programmation, en assignant des valeurs aux constantes ou aux variables avec la syntaxe `name = value`. La constante ou la variable de gauche prend la valeur de droite. Le nom de la constante ou de la variable doit commencer par une lettre ou un trait de soulignement.

Le type de données stockées dans la variable n'a pas besoin d'être indiqué, car JavaScript est un langage à typage dynamique. Le type de la variable est déduit de la valeur qui lui est attribuée. Cependant, il est pratique de désigner certains attributs dans la déclaration pour garantir le résultat souhaité.

NOTE

TypeScript est un langage inspiré de JavaScript qui, comme les langages de bas niveau, vous permet de déclarer des variables pour des types de données spécifiques.

Constantes

Une constante est un symbole qui est attribué une fois au démarrage du programme et qui ne change jamais. Les constantes sont utiles pour spécifier des valeurs fixes, comme définir la constante `PI` comme étant 3.14159265, ou `COMPANY_NAME` pour contenir le nom de votre entreprise.

Dans une application Web, par exemple, prenons un client qui reçoit des informations météorologiques d'un serveur distant. Le programmeur peut décider que l'adresse du serveur doit être constante, car elle ne changera pas pendant l'exécution de l'application. En revanche, les informations sur la température peuvent changer à chaque nouvelle réception de données du serveur.

L'intervalle entre les requêtes faites au serveur peut également être défini comme une constante, qui peut être demandée à partir de n'importe quelle partie du programme :


```
const update_interval = 10;

function setup_app(){
  console.log("Update every " + update_interval + "minutes");
}
```

Lorsqu'elle est appelée, la fonction `setup_app()` affiche le message `Update every 10 minutes` sur la console. Le terme `const` placé devant le nom `update_interval` permet de s'assurer que sa valeur restera la même tout au long de l'exécution du script. Si l'on tente de modifier la valeur d'une constante, une erreur `TypeError: Assignment to constant variable` est générée.

Variables

Sans le terme `const`, JavaScript suppose automatiquement que `update_interval` est une variable et que sa valeur peut être modifiée. Ceci est équivalent à déclarer la variable explicitement avec `var` :

```
var update_interval;
update_interval = 10;

function setup_app(){
  console.log("Update every " + update_interval + "minutes");
}
```

Notez que bien que la variable `update_interval` ait été définie en dehors de la fonction, elle a été accédée depuis l'intérieur de la fonction. Toute constante ou variable déclarée en dehors des fonctions ou des blocs de code définis par des accolades (`{}`) a une portée *globale*, c'est-à-dire qu'on peut y accéder depuis n'importe quelle partie du code. L'inverse n'est pas vrai : une constante ou une variable déclarée à l'intérieur d'une fonction a une portée *locale*, c'est-à-dire qu'elle n'est accessible qu'à l'intérieur de la fonction elle-même. Les blocs de code délimités par des accolades, comme ceux placés dans les structures de décision `if` ou les boucles `for`, délimitent la portée des constantes, mais pas des variables déclarées comme `var`. Le code suivant, par exemple, est valide :

```
var success = true;
if ( success == true )
{
  var message = "Transaction succeeded";
  var retry = 0;
}
else
{
```

```
var message = "Transaction failed";
var retry = 1;
}

console.log(message);
```

L'instruction `console.log(message)` est capable d'accéder à la variable `message`, même si elle a été déclarée dans le bloc de code de la structure `if`. La même chose ne se produirait pas si `message` était constante, comme le montre l'exemple suivant :

```
var success = true;
if ( success == true )
{
    const message = "Transaction succeeded";
    var retry = 0;
}
else
{
    const message = "Transaction failed";
    var retry = 1;
}

console.log(message);
```

Dans ce cas, un message d'erreur du type `ReferenceError` : `message is not defined` serait émis et le script serait arrêté. Bien que cela puisse sembler une limitation, restreindre la portée des variables et des constantes permet d'éviter toute confusion entre les informations traitées dans le corps du script et dans ses différents blocs de code. Pour cette raison, les variables déclarées avec `let` au lieu de `var` ont également une portée limitée par les blocs délimités par des accolades. Il existe d'autres différences subtiles entre la déclaration d'une variable avec `var` ou avec `let`, mais la plus importante d'entre elles concerne la portée de la variable, comme nous le verrons ici.

Types de valeurs

La plupart du temps, le programmeur n'a pas à se soucier du type de données stockées dans une variable, car JavaScript l'identifie automatiquement avec l'un de ses types *primitifs* lors de la première affectation d'une valeur à la variable. Certaines opérations, cependant, peuvent être spécifiques à un type de données ou à un autre et peuvent entraîner des erreurs si elles sont utilisées sans précautions. En outre, JavaScript propose des types *structurés* qui vous permettent de combiner plusieurs types primitifs en un seul objet.

Types primitifs

Les instances de type primitif correspondent aux variables traditionnelles, qui ne stockent qu'une seule valeur. Les types sont définis implicitement, de sorte que l'opérateur `typeof` peut être utilisé pour identifier quel type de valeur est stocké dans une variable :

```
console.log("Undefined variables are of type", typeof variable);

{
  let variable = true;
  console.log("Value `true` is of type " + typeof variable);
}

{
  let variable = 3.14159265;
  console.log("Value `3.14159265` is of type " + typeof variable);
}

{
  let variable = "Text content";
  console.log("Value `Text content` is of type " + typeof variable);
}

{
  let variable = Symbol();
  console.log("A symbol is of type " + typeof variable);
}
```

Ce script affichera sur la console quel type de variable a été utilisé dans chaque cas :

```
undefined variables are of type undefined
Value `true` is of type boolean
Value `3.14159265` is of type number
Value `Text content` is of type string
A symbol is of type symbol
```

Remarquez que la première ligne essaie de trouver le type d'une variable non déclarée. Cela fait que la variable donnée est identifiée comme `undefined`. Le type `symbol` est le type primitif le moins intuitif. Son but est de fournir un nom d'attribut unique dans un objet lorsqu'il n'est pas nécessaire de définir un nom d'attribut spécifique. Un objet est l'une des structures de données que nous examinerons ensuite.

Types structurés

Si les types primitifs sont suffisants pour écrire des routines simples, leur utilisation exclusive dans des applications plus complexes présente des inconvénients. Une application de commerce électronique, par exemple, serait beaucoup plus difficile à écrire, car le programmeur devrait trouver des moyens de stocker des listes d'articles et les valeurs correspondantes en utilisant uniquement des variables de types primitifs.

Les types structurés simplifient la tâche consistant à regrouper des informations de même nature dans une seule variable. La liste des articles d'un panier d'achat, par exemple, peut être stockée dans une seule variable de type *array* :

```
let cart = ['Mi lk', 'Bread', 'Eggs'];
```

Comme le montre l'exemple, un tableau d'éléments est désigné par des crochets. L'exemple a rempli le tableau avec trois valeurs de chaîne littérales, d'où l'utilisation de guillemets simples. Les variables peuvent également être utilisées comme éléments d'un tableau, mais dans ce cas, elles doivent être désignées sans guillemets. Le nombre d'éléments d'un tableau peut être interrogé avec la propriété `length` :

```
let cart = ['Mi lk', 'Bread', 'Eggs'];  
console.log(cart.length);
```

Le nombre 3 sera affiché dans la sortie console. De nouveaux éléments peuvent être ajoutés au tableau avec la méthode `push()` :

```
cart.push('Candy');  
console.log(cart.length);
```

Cette fois, le nombre affiché sera 4. On peut accéder à chaque élément de la liste par son indice numérique, en commençant par 0 :

```
console.log(cart[0]);  
console.log(cart[3]);
```

La sortie affichée sur la console sera :

```
Mi lk
```

Candy

Tout comme vous pouvez utiliser `push()` pour ajouter un élément, vous pouvez utiliser `pop()` pour retirer le dernier élément d'un tableau.

Les valeurs stockées dans un tableau ne doivent pas nécessairement être du même type. Il est possible, par exemple, de stocker la quantité de chaque article à côté de celui-ci. Une liste de courses comme celle de l'exemple précédent pourrait être construite comme suit :

```
let cart = ['Milk', 1, 'Bread', 4, 'Eggs', 12, 'Candy', 2];

// Item indexes are even
let item = 2;

// Quantities indexes are odd
let quantity = 3;

console.log("Item: " + cart[item]);
console.log("Quantity: " + cart[quantity]);
```

La sortie affichée sur la console après l'exécution de ce code est la suivante :

```
Item: Bread
Quantity: 4
```

Comme vous l'avez peut-être déjà remarqué, combiner les noms des articles avec leurs quantités respectives dans un seul tableau n'est pas une bonne idée, car la relation entre eux n'est pas explicite dans la structure de données et elle est très sensible aux erreurs (humaines). Même si l'on utilisait un tableau pour les noms et un autre pour les quantités, le maintien de l'intégrité de la liste exigerait le même soin et ne serait pas très productif. Dans ces situations, la meilleure alternative est d'utiliser une structure de données plus appropriée : un *objet*.

En JavaScript, une structure de données de type objet vous permet de lier des propriétés à une variable. De plus, contrairement à un tableau, les éléments qui composent un objet n'ont pas un ordre fixe. Un article de liste de courses, par exemple, peut être un objet avec les propriétés `name` et `quantity` :

```
let item = { name: 'Milk', quantity: 1 };
console.log("Item: " + item.name);
```

```
console.log("Quantity: " + item.quantity);
```

Cet exemple montre qu'un objet peut être défini en utilisant des accolades (`{}`), où chaque paire propriété/valeur est séparée par un deux-points et les propriétés sont séparées par des virgules. La propriété est accessible sous le format *variable.propriété*, comme dans `item.name`, à la fois pour la lecture et pour l'attribution de nouvelles valeurs. La sortie affichée sur la console après l'exécution de ce code est :

```
Item: Milk  
Quantity: 1
```

Enfin, chaque objet représentant un article peut être inclus dans le tableau de la liste de courses. Cela peut être fait directement lors de la création de la liste :

```
let cart = [{ name: 'Milk', quantity: 1 }, { name: 'Bread', quantity: 4 }];
```

Comme précédemment, un nouvel objet représentant un élément peut être ajouté ultérieurement au tableau :

```
cart.push({ name: 'Eggs', quantity: 12 });
```

Les éléments de la liste sont désormais accessibles par leur index et leur nom de propriété :

```
console.log("Third item: " + cart[2].name);  
console.log(cart[2].name + " quantity: " + cart[2].quantity);
```

La sortie affichée sur la console après l'exécution de ce code est la suivante :

```
third item: eggs  
Eggs quantity: 12
```

Les structures de données permettent au programmeur de garder son code beaucoup plus organisé et plus facile à maintenir, que ce soit par l'auteur original ou par d'autres programmeurs de l'équipe. En outre, de nombreux résultats des fonctions JavaScript sont des types structurés, qui doivent être traités correctement par le programmeur.

Opérateurs

Jusqu'à présent, nous n'avons vu que la manière d'attribuer des valeurs aux variables nouvellement créées. Aussi simple que cela soit, tout programme effectuera plusieurs autres manipulations sur les valeurs des variables. JavaScript propose plusieurs types d'*opérateurs* qui peuvent agir directement sur la valeur d'une variable ou stocker le résultat de l'opération dans une nouvelle variable.

La plupart des opérateurs sont orientés vers les opérations arithmétiques. Pour augmenter la quantité d'un article dans la liste des courses, par exemple, il suffit d'utiliser l'opérateur d'addition `+` :

```
item.quantity = item.quantity + 1;
```

Le bout de code suivant imprime la valeur de `item.quantity` avant et après l'ajout. Ne confondez pas les rôles du signe plus dans le bout de code. Les instructions `console.log` utilisent un signe plus pour combiner deux chaînes de caractères.

```
let item = { name: 'Milk', quantity: 1 };  
console.log("Item: " + item.name);  
console.log("Quantity: " + item.quantity);  
  
item.quantity = item.quantity + 1;  
console.log("New quantity: " + item.quantity);
```

La sortie affichée sur la console après l'exécution de ce code est la suivante :

```
Item: Milk  
Quantity: 1  
New quantity: 2
```

Notez que la valeur précédemment stockée dans `item.quantity` est utilisée comme opérande de l'opération d'addition : `item.quantity = item.quantity + 1`. Ce n'est qu'une fois l'opération terminée que la valeur de `item.quantity` est mise à jour avec le résultat de l'opération. Ce type d'opération arithmétique impliquant la valeur courante de la variable cible est assez courant, c'est pourquoi il existe des opérateurs raccourcis qui vous permettent d'écrire la même opération dans un format réduit :

```
item.quantity += 1;
```

Les autres opérations de base ont également des opérateurs raccourcis équivalents :

- `a = a - b` est équivalente à `a -= b`.
- `a = a * b` est équivalente à `a *= b`.
- `a = a / b` est équivalente à `a /= b`.

Pour l'addition et la soustraction, il existe un troisième format disponible lorsque le second opérande n'est que d'une unité :

- `a = a + 1` est équivalente à `a++`.
- `a = a - 1` est équivalente à `a--`.

Plusieurs opérateurs peuvent être combinés dans la même opération et le résultat peut être stocké dans une nouvelle variable. Par exemple, l'instruction suivante calcule le prix total d'un article plus les frais d'expédition :

```
let total = item.quantity * 9.99 + 3.15;
```

L'ordre d'exécution des opérations suit l'ordre de priorité traditionnel : on exécute d'abord les opérations de multiplication et de division, puis seulement les opérations d'addition et de soustraction. Les opérateurs ayant la même priorité sont exécutés dans l'ordre où ils apparaissent dans l'expression, de gauche à droite. Pour remplacer l'ordre de priorité par défaut, vous pouvez utiliser des parenthèses, comme dans `a * (b + c)`.

Dans certaines situations, le résultat d'une opération n'a même pas besoin d'être stocké dans une variable. C'est le cas lorsque vous voulez évaluer le résultat d'une expression dans une instruction `if` :

```
if ( item.quantity % 2 == 0 )
{
    console.log("Quantity for the item is even");
}
else
{
    console.log("Quantity for the item is odd");
}
```

L'opérateur `%` (modulo) renvoie le reste de la division du premier opérande par le second opérande. Dans l'exemple, l'instruction `if` vérifie si le reste de la division de `item.quantity` par 2 est égal à zéro, c'est-à-dire si `item.quantity` est un multiple de 2.

Lorsque l'un des opérandes de l'opérateur `+` est une chaîne de caractères, les autres opérateurs sont *constraints* en chaînes de caractères et le résultat est une concaténation de chaînes de caractères. Dans les exemples précédents, ce type d'opération a été utilisé pour concaténer des chaînes de caractères et des variables dans l'argument de l'instruction `console.log`.

Cette transformation automatique peut ne pas être le comportement souhaité. Une valeur fournie par l'utilisateur dans un champ de formulaire, par exemple, peut être identifiée comme une chaîne de caractères, mais il s'agit en fait d'une valeur numérique. Dans ce cas, la variable doit d'abord être convertie en un nombre avec la fonction `Number()` :

```
sum = Number(value1) + value2;
```

De plus, il est important de vérifier que l'utilisateur a fourni une valeur valide avant de procéder à l'opération. En JavaScript, une variable sans valeur assignée contient la valeur `null`. Cela permet au programmeur d'utiliser une instruction de décision telle que `if (value1 == null)` pour vérifier si une valeur a été attribuée à une variable, quel que soit le type de la valeur attribuée à la variable.

Exercices Guidés

1. Un tableau est une structure de données présente dans plusieurs langages de programmation, dont certains ne permettent que des tableaux avec des éléments de même type. Dans le cas de JavaScript, est-il possible de définir un tableau avec des éléments de types différents ?

2. D'après l'exemple `let item = { name: 'Milk', quantity: 1 }` pour un objet dans une liste de courses, comment pourrait-on déclarer cet objet pour inclure le prix de l'article ?

3. Dans une seule ligne de code, quelles sont les façons de mettre à jour la valeur d'une variable à la moitié de sa valeur actuelle ?

Exercices d'exploration

1. Dans le code suivant, quelle valeur sera affichée dans la sortie de la console ?

```
var value = "Global";  
  
{  
  value = "Location";  
}  
  
console.log(value);
```

2. Que se passe-t-il lorsque l'un ou plusieurs des opérandes impliqués dans une opération de multiplication est une chaîne de caractères ?

3. Comment est-il possible de supprimer l'élément Eggs du tableau cart déclaré avec `let cart = ['Milk', 'Bread', 'Eggs']` ?

Résumé

Cette leçon couvre l'utilisation de base des constantes et des variables en JavaScript. JavaScript est un langage à typage dynamique, le programmeur n'a donc pas besoin de spécifier le type de la variable avant de la définir. Cependant, il est important de connaître les types primitifs du langage pour garantir le résultat correct des opérations de base. En outre, les structures de données telles que les tableaux et les objets combinent les types primitifs et permettent au programmeur de construire des variables composites plus complexes. Cette leçon aborde les concepts et procédures suivants :

- Comprendre les constantes et les variables
- Portée des variables
- Déclarer des variables avec ``var`` et ``let``.
- Les types primitifs
- Opérateurs arithmétiques
- Tableaux et objets
- Contrainte et conversion de types

Réponses aux exercices guidés

1. Un tableau est une structure de données présente dans plusieurs langages de programmation, dont certains ne permettent que des tableaux avec des éléments de même type. Dans le cas de JavaScript, est-il possible de définir un tableau avec des éléments de types différents ?

Oui, en JavaScript, il est possible de définir des tableaux avec des éléments de différents types primitifs, tels que des chaînes de caractères et des nombres.

2. D'après l'exemple `let item = { name: 'Milk', quantity: 1 }` pour un objet dans une liste de courses, comment pourrait-on déclarer cet objet pour inclure le prix de l'article ?

```
let item = { name: 'Milk', quantity: 1, price: 4.99 };
```

3. Dans une seule ligne de code, quelles sont les façons de mettre à jour la valeur d'une variable à la moitié de sa valeur actuelle ?

On peut utiliser la variable elle-même comme opérande, `valeur = valeur / 2`, ou l'opérateur raccourci `/=` : `valeur /= 2`.

Réponses aux exercices d'exploration

1. Dans le code suivant, quelle valeur sera affichée dans la sortie de la console ?

```
var value = "Global";

{
  value = "Location";
}

console.log(value);
```

Location

2. Que se passe-t-il lorsque l'un ou plusieurs des opérandes impliqués dans une opération de multiplication est une chaîne de caractères ?

JavaScript attribuera la valeur `NaN` (*Not a Number* : n'est pas un nombre) au résultat, indiquant que l'opération n'est pas valide.

3. Comment est-il possible de supprimer l'élément `Eggs` du tableau `cart` déclaré avec `let cart = ['Milk', 'Bread', 'Eggs']` ?

Les tableaux en javascript ont la méthode `pop()`, qui supprime le dernier élément de la liste : `cart.pop()`.



034.3 Structures de contrôle et fonctions JavaScript

Referencia al objetivo del LPI

Web Development Essentials version 1.0, Exam 030, Objective 034.3

Valeur

4

Domaines de connaissance les plus importants

- Comprendre les valeurs vraies et fausses
- Comprendre les opérateurs de comparaison
- Comprendre la différence entre une comparaison libre et une comparaison stricte
- Utiliser les instructions conditionnelles
- Utiliser les boucles
- Définir des fonctions personnalisées

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `if`, `else if`, `else`
- `switch`, `case`, `break`
- `for`, `while`, `break`, `continue`
- `function`, `return`
- `==`, `!=`, `<`, `>`, `<=`, `>=`
- `===`, `!==`



034.3 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	034 Programmation JavaScript
Objectif :	034.3 Structures de contrôle et fonctions JavaScript
Leçon :	1 sur 2

Introduction

Comme tout autre langage de programmation, le code JavaScript est une collection d'instructions qui indique à un interpréteur d'instructions ce qu'il faut faire dans un ordre séquentiel. Cependant, cela ne signifie pas que chaque instruction ne doit être exécutée qu'une seule fois ou qu'elle ne doit pas être exécutée du tout. La plupart des instructions ne doivent être exécutées que lorsque des conditions spécifiques sont remplies. Même lorsqu'un script est déclenché de manière asynchrone par des événements indépendants, il doit souvent vérifier un certain nombre de variables de contrôle pour trouver la bonne portion de code à exécuter.

Instructions If

La structure de contrôle la plus simple est donnée par l'instruction `if`, qui exécutera l'instruction immédiatement après si la condition spécifiée est vraie. JavaScript considère que les conditions sont vraies si la valeur évaluée est différente de zéro. Tout ce qui se trouve à l'intérieur des parenthèses après le mot `if` (les espaces sont ignorés) sera interprété comme une condition. Dans l'exemple suivant, le nombre littéral `1` est la condition :


```
if ( 1 ) console.log("1 is always true");
```

Le nombre 1 est explicitement écrit dans cet exemple de condition, il est donc traité comme une valeur constante (il reste le même tout au long de l'exécution du script) et il donnera toujours vrai lorsqu'il est utilisé comme expression conditionnelle. Le mot `true` (sans les guillemets) pourrait également être utilisé à la place de 1, car il est également traité comme une valeur littérale vraie par le langage. L'instruction `console.log` imprime ses arguments dans la *fenêtre console* du navigateur.

TIP

La console du navigateur affiche les erreurs, les avertissements et les messages d'information envoyés avec l'instruction JavaScript `console.log`. Sur Chrome, la combinaison de touches `Ctrl + Shift + J` (`Cmd + Option + J` sur Mac) ouvre la console. Sur Firefox, la combinaison de touches `Ctrl + Shift + K` (`Cmd + Option + K` sur Mac) ouvre l'onglet console dans les outils du développeur.

Bien que syntaxiquement correcte, l'utilisation d'expressions constantes dans les conditions n'est pas très utile. Dans une application réelle, vous voudrez probablement tester la véracité d'une variable :

```
let my_number = 3;
if ( my_number ) console.log("The value of my_number is", my_number, "and it yields true");
```

La valeur assignée à la variable `my_number` (3) est non nulle, donc elle donne vrai. Mais cet exemple n'est pas d'usage courant, car vous avez rarement besoin de tester si un nombre est égal à zéro. Il est beaucoup plus courant de comparer une valeur à une autre et de tester si le résultat est vrai :

```
let my_number = 3;
if ( my_number == 3 ) console.log("The value of my_number is", my_number, "indeed");
```

L'opérateur de comparaison double égal est utilisé car l'opérateur simple égal est déjà défini comme l'opérateur d'affectation. La valeur de chaque côté de l'opérateur est appelée un *opérande*. L'ordre des opérands n'a pas d'importance et toute expression qui renvoie une valeur peut être un opérande. Voici une liste d'autres opérateurs de comparaison disponibles :

`value1 == value2`

Vrai si `value1` est égale à `value2`.

value1 != value2

Vrai si value1 n'est pas égale à value2.

value1 < value2

Vrai si value1 est inférieur à value2.

value1 > value2

Vrai si value1 est plus grand que value2.

value1 <= value2

Vrai si value1 est inférieur ou égal à value2.

value1 >= value2

Vrai si value1 est supérieur ou égal à value2.

En général, il importe peu que l'opérande à gauche de l'opérateur soit une chaîne de caractères et que l'opérande à droite soit un nombre, tant que JavaScript est capable de convertir l'expression en une comparaison significative. Ainsi, la chaîne contenant le caractère 1 sera traitée comme le nombre 1 lorsqu'elle sera comparée à une variable numérique. Pour s'assurer que l'expression ne donne un résultat vrai que si les deux opérandes sont exactement du même type et de la même valeur, l'opérateur d'identité stricte `===` doit être utilisé à la place de `==`. De même, l'opérateur de non-identité stricte `!==` est évalué comme vrai si le premier opérande n'est pas exactement du même type et de la même valeur que le second opérateur.

Optionnellement, la structure de contrôle `if` peut exécuter une autre instruction lorsque l'expression est évaluée comme fausse :

```
let my_number = 4;
if ( my_number == 3 ) console.log("The value of my_number is 3");
else console.log("The value of my_number is not 3");
```

L'instruction `else` doit suivre immédiatement l'instruction `if`. Jusqu'à présent, nous n'avons exécuté qu'une seule instruction lorsque la condition est vérifiée. Pour exécuter plus d'une instruction, vous devez les entourer d'accolades :

```
let my_number = 4;
if ( my_number == 3 )
{
  console.log("The value of my_number is 3");
  console.log("and this is the second statement in the block");
}
```

```

}
else
{
  console.log("The value of my_number is not 3");
  console.log("and this is the second statement in the block");
}

```

Un groupe d'une ou de plusieurs instructions délimitées par une paire d'accolades est appelé *bloc*. Il est courant d'utiliser des instructions de type bloc même lorsqu'il n'y a qu'une seule instruction à exécuter, afin de rendre le style de codage cohérent dans tout le script. En outre, JavaScript n'exige pas que les accolades ou les instructions soient sur des lignes séparées, mais cela améliore la lisibilité et facilite la maintenance du code.

Les structures de contrôle peuvent être imbriquées les unes dans les autres, mais il est important de ne pas confondre les accolades ouvrantes et fermantes de chaque bloc d'instructions :

```

let my_number = 4;

if ( my_number > 0 )
{
  console.log("The value of my_number is positive");

  if ( my_number % 2 == 0 )
  {
    console.log("and it is an even number");
  }
  else
  {
    console.log("and it is an odd number");
  }
} // end of if ( my_number > 0 )
else
{
  console.log("The value of my_number is less than or equal to 0");
  console.log("and I decided to ignore it");
}

```

Les expressions évaluées par l'instruction `if` peuvent être plus élaborées que de simples comparaisons. C'est le cas dans l'exemple précédent, où l'expression arithmétique `my_number % 2` était employée à l'intérieur des parenthèses dans le `if` imbriqué. L'opérateur `%` renvoie le reste après avoir divisé le nombre à sa gauche par le nombre à sa droite. Les opérateurs arithmétiques comme `%` sont prioritaires sur les opérateurs de comparaison comme `==`, la comparaison utilisera donc le

résultat de l'expression arithmétique comme opérande gauche.

Dans de nombreuses situations, les structures conditionnelles imbriquées peuvent être combinées en une seule structure en utilisant des *opérateurs logiques*. Si nous ne nous intéressons qu'aux nombres pairs positifs, par exemple, une seule structure `if` pourrait être utilisée :

```
let my_number = 4;

if ( my_number > 0 && my_number % 2 == 0 )
{
  console.log("The value of my_number is positive");
  console.log("and it is an even number");
}
else
{
  console.log("The value of my_number either 0, negative");
  console.log("or it is a negative number");
}
```

L'opérateur double esperluette `&&` dans l'expression évaluée est l'opérateur logique *AND*. Il n'est évalué comme vrai que si l'expression à sa gauche et l'expression à sa droite sont évaluées comme vraies. Si vous voulez faire correspondre des nombres qui sont soit positifs, soit pairs, il faut utiliser l'opérateur `||` à la place, qui représente l'opérateur logique *OR* :

```
let my_number = -4;

if ( my_number > 0 || my_number % 2 == 0 )
{
  console.log("The value of my_number is positive");
  console.log("or it is a even negative number");
}
```

Dans cet exemple, seuls les nombres impairs négatifs ne correspondront pas aux critères imposés par l'expression composite. Si vous avez l'intention contraire, c'est-à-dire de ne faire correspondre que les nombres impairs négatifs, ajoutez l'opérateur logique *NOT* ! au début de l'expression :

```
let my_number = -5;

if ( ! ( my_number > 0 || my_number % 2 == 0 ) )
{
  console.log("The value of my_number is an odd negative number");
}
```

```
}
```

L'ajout des parenthèses dans l'expression composite force l'expression qu'elles encadrent à être évaluée en premier. Sans ces parenthèses, l'opérateur NOT ne s'appliquerait qu'à `my_number > 0` et ensuite l'expression OR serait évaluée. Les opérateurs `&&` et `||` sont appelés opérateurs logiques *binaires*, car ils nécessitent deux opérandes. L'opérateur `!` est appelé opérateur logique *unaire*, car il ne nécessite qu'un seul opérande.

Structures switch

Bien que la structure `if` soit assez polyvalente et suffisante pour contrôler le flux du programme, la structure de contrôle `switch` peut être plus appropriée lorsque des résultats autres que vrai ou faux doivent être évalués. Par exemple, si nous voulons prendre une action distincte pour chaque élément choisi dans une liste, il sera nécessaire d'écrire une structure `if` pour chaque évaluation :

```
// Available languages: en (English), es (Spanish), pt (Portuguese)
let language = "pt";

// Variable to register whether the language was found in the list
let found = 0;

if ( language == "en" )
{
    found = 1;
    console.log("English");
}

if ( found == 0 && language == "es" )
{
    found = 1;
    console.log("Spanish");
}

if ( found == 0 && language == "pt" )
{
    found = 1;
    console.log("Portuguese");
}

if ( found == 0 )
{
    console.log(language, " is unknown to me");
}
```

```
}
```

Dans cet exemple, une variable auxiliaire `found` est utilisée par toutes les structures `if` pour savoir si une correspondance s'est produite. Dans un cas comme celui-ci, la structure `switch` effectuera la même tâche, mais de manière plus succincte :

```
switch ( language )
{
  case "en":
    console.log("English");
    break;
  case "es":
    console.log("Spanish");
    break;
  case "pt":
    console.log("Portuguese");
    break;
  default:
    console.log(language, " not found");
}
```

Chaque `case` imbriqué est appelé une *clause*. Lorsqu'une clause correspond à l'expression évaluée, elle exécute les instructions qui suivent les deux points jusqu'à l'instruction `break`. La dernière clause n'a pas besoin d'instruction `break` et elle est souvent utilisée pour définir l'action par défaut lorsqu'aucune autre correspondance ne se produit. Comme on le voit dans l'exemple, la variable auxiliaire n'est pas nécessaire dans la structure `switch`.

WARNING

`switch` utilise la comparaison stricte pour faire correspondre les expressions à ses clauses `case`.

Si plusieurs clauses déclenchent la même action, vous pouvez combiner deux ou plusieurs conditions `case` :

```
switch ( language )
{
  case "en":
  case "en_US":
  case "en_GB":
    console.log("English");
    break;
  case "es":
```

```

    console.log("Spanish");
    break;
  case "pt":
  case "pt_BR":
    console.log("Portuguese");
    break;
  default:
    console.log(language, " not found");
}

```

Boucles

Dans les exemples précédents, les structures `if` et `switch` étaient bien adaptées aux tâches qui doivent s'exécuter une seule fois après avoir passé un ou plusieurs tests conditionnels. Cependant, il y a des situations où une tâche doit s'exécuter de façon répétée—dans ce qu'on appelle une *boucle*—tant que son expression conditionnelle continue à être testée comme vraie. Si vous avez besoin de savoir si un nombre est premier, par exemple, vous devrez vérifier si la division de ce nombre par tout entier supérieur à 1 et inférieur à lui-même a un reste égal à 0. Si c'est le cas, le nombre a un facteur entier et il n'est pas premier. (Il ne s'agit pas d'une méthode rigoureuse ou efficace pour trouver des nombres premiers, mais elle fonctionne comme un exemple simple). Les structures de contrôle en boucle sont plus adaptées à de tels cas, en particulier l'instruction `while` :

```

// A naive prime number tester

// The number we want to evaluate
let candidate = 231;

// Auxiliary variable
let is_prime = true;

// The first factor to try
let factor = 2;

// Execute the block statement if factor is
// less than candidate and keep doing it
// while factor is less than candidate
while ( factor < candidate )
{
    if ( candidate % factor == 0 )
    {
        // The remainder is zero, so the candidate is not prime
        is_prime = false;
    }
}

```

```
    break;
  }

  // The next factor to try. Simply
  // increment the current factor by one
  factor++;
}

// Display the result in the console window.
// If candidate has no integer factor, then
// the auxiliary variable is_prime still true
if ( is_prime )
{
  console.log(candidate, "is prime");
}
else
{
  console.log(candidate, "is not prime");
}
```

Le bloc d'instructions qui suit l'instruction `while` s'exécutera de manière répétée tant que la condition `factor < candidate` est vraie. Elle s'exécutera au moins une fois, tant que nous initialiserons la variable `factor` avec une valeur inférieure à `candidate`. La structure `if` imbriquée dans la structure `while` va évaluer si le reste de `candidate` divisé par `factor` est égal à zéro. Si c'est le cas, le nombre candidat n'est pas premier et la boucle peut se terminer. L'instruction `break` terminera la boucle et l'exécution sautera à la première instruction après le bloc `while`.

Notez que le résultat de la condition utilisée par l'instruction `while` doit changer à chaque boucle, sinon l'instruction de bloc bouclera "à l'infini". Dans l'exemple, nous incrémentons la variable `factor` – le prochain diviseur que nous voulons essayer – et cela garantit que la boucle se terminera à un moment donné.

Cette simple implémentation d'un testeur de nombres premiers fonctionne comme prévu. Cependant, nous savons qu'un nombre qui n'est pas divisible par deux ne sera pas divisible par un autre nombre pair. Par conséquent, nous pourrions simplement sauter les nombres pairs en ajoutant une autre instruction `if` :

```
while ( factor < candidate )
{

  // Skip even factors bigger than two
  if ( factor > 2 && factor % 2 == 0 )
```



```

{
    factor++;
    continue;
}

if ( candidate % factor == 0 )
{
    // The remainder is zero, so the candidate is not prime
    is_prime = false;
    break;
}

// The next number that will divide the candidate
factor++;
}

```

L'instruction `continue` est similaire à l'instruction `break`, mais au lieu de terminer cette itération de la boucle, elle va ignorer le reste du bloc de la boucle et commencer une nouvelle itération. Notez que la variable `factor` a été modifiée avant l'instruction `continue`, sinon la boucle aurait le même résultat à l'itération suivante. Cet exemple est trop simple et sauter une partie de la boucle n'améliorera pas vraiment ses performances, mais sauter les instructions redondantes est très important pour écrire des applications efficaces.

Les boucles sont si couramment utilisées qu'elles existent dans de nombreuses variantes différentes. La boucle `for` est particulièrement adaptée à l'itération de valeurs séquentielles, car elle nous permet de définir les règles de la boucle en une seule ligne :

```

for ( let factor = 2; factor < candidate; factor++ )
{
    // Skip even factors bigger than two
    if ( factor > 2 && factor % 2 == 0 )
    {
        continue;
    }

    if ( candidate % factor == 0 )
    {
        // The remainder is zero, so the candidate is not prime
        is_prime = false;
        break;
    }
}

```

Cet exemple produit exactement le même résultat que l'exemple précédent avec `while`, mais son expression entre parenthèses comprend trois parties, séparées par des points-virgules : l'initialisation (`let factor = 2`), la condition de la boucle (`factor < candidate`), et l'expression finale à évaluer à la fin de chaque itération de la boucle (`factor++`). Les instructions `continue` et `break` s'appliquent également aux boucles `for`. L'expression finale entre parenthèses (`factor++`) sera évaluée après l'instruction `continue`, elle ne doit donc pas se trouver à l'intérieur du bloc d'instructions, sinon elle sera incrémentée deux fois avant la prochaine itération.

JavaScript dispose de types spéciaux de boucles `for` pour travailler avec des objets de type tableau. Nous pourrions, par exemple, vérifier un tableau de variables candidates au lieu d'une seule :

```
// A naive prime number tester

// The array of numbers we want to evaluate
let candidates = [111, 139, 293, 327];

// Evaluates every candidate in the array
for (candidate of candidates)
{
    // Auxiliary variable
    let is_prime = true;

    for ( let factor = 2; factor < candidate; factor++ )
    {
        // Skip even factors bigger than two
        if ( factor > 2 && factor % 2 == 0 )
        {
            continue;
        }

        if ( candidate % factor == 0 )
        {
            // The remainder is zero, so the candidate is not prime
            is_prime = false;
            break;
        }
    }

    // Display the result in the console window
    if ( is_prime )
    {
        console.log(candidate, "is prime");
    }
    else

```

```
{  
  console.log(candidate, "is not prime");  
}  
}
```

L'instruction `for (candidat des candidats)` attribue un élément du tableau `candidates` à la variable `candidate` et l'utilise dans le bloc d'instructions, en répétant le processus pour chaque élément du tableau. Vous n'avez pas besoin de déclarer la variable `candidate` séparément, car la boucle `for` la définit. Enfin, le même code que celui de l'exemple précédent est imbriqué dans ce nouveau bloc d'instructions, mais cette fois-ci en testant chaque candidat du tableau.

Exercices guidés

1. Quelles valeurs de la variable `my_var` correspondent à la condition `my_var > 0 && my_var < 9` ?

2. Quelles valeurs de la variable `my_var` correspondent à la condition `my_var > 0 || my_var < 9` ?

3. Combien de fois la boucle `while` suivante exécutera-t-elle son bloc d'instructions ?

```
let i = 0;
while ( 1 )
{
    if ( i == 10 )
    {
        continue;
    }
    i++;
}
```

Exercices d'exploration

1. Que se passe-t-il si l'opérateur d'affectation égale `=` est utilisé au lieu de l'opérateur de comparaison double égale `==` ?

2. Écrivez un fragment de code utilisant la structure de contrôle `if` où une comparaison d'égalité ordinaire retournera vrai, mais pas une comparaison d'égalité stricte.

3. Réécrivez l'instruction `for` suivante en utilisant l'opérateur logique unaire `NOT` dans la condition de la boucle. Le résultat de la condition doit être le même.

```
for ( let factor = 2; factor < candidate; factor++ )
```

4. En vous basant sur les exemples de cette leçon, écrivez une structure de contrôle en boucle qui affiche tous les facteurs entiers d'un nombre donné.

Résumé

Cette leçon explique comment utiliser les structures de contrôle dans le code JavaScript. Les structures conditionnelles et les boucles sont des éléments essentiels de tout paradigme de programmation, et le développement web avec JavaScript ne fait pas exception. La leçon aborde les concepts et procédures suivants :

- L'instruction `if` et les opérateurs de comparaison.
- Comment utiliser la structure `switch` avec `case`, `default`, et `break`.
- La différence entre une comparaison ordinaire et une comparaison stricte.
- Les structures de contrôle des boucles : `while` et `for`.

Réponses aux exercices guidés

1. Quelles valeurs de la variable `my_var` correspondent à la condition `my_var > 0 && my_var < 9` ?

Seuls les nombres qui sont à la fois supérieurs à 0 et inférieurs à 9. L'opérateur logique `&&` (AND) exige que les deux comparaisons soient respectées.

2. Quelles valeurs de la variable `my_var` correspondent à la condition `my_var > 0 || my_var < 9` ?

L'utilisation de l'opérateur logique `||` (OR) permet de faire correspondre n'importe quel nombre, car tout nombre sera soit supérieur à 0, soit inférieur à 9.

3. Combien de fois la boucle `while` suivante exécutera-t-elle son bloc d'instructions ?

```
let i = 0;
while ( 1 )
{
    if ( i == 10 )
    {
        continue;
    }
    i++;
}
```

Le bloc se répétera à l'infini, car aucune condition d'arrêt n'a été fournie.

Réponses aux exercices d'exploration

1. Que se passe-t-il si l'opérateur d'affectation égale `=` est utilisé au lieu de l'opérateur de comparaison double égale `==` ?

La valeur située à droite de l'opérateur est affectée à la variable située à gauche et le résultat est transmis à la comparaison, ce qui peut ne pas être le comportement souhaité.

2. Écrivez un fragment de code utilisant la structure de contrôle `if` où une comparaison d'égalité ordinaire retournera vrai, mais pas une comparaison d'égalité stricte.

```
let a = "1";
let b = 1;

if ( a == b )
{
  console.log("An ordinary comparison will match.");
}

if ( a === b )
{
  console.log("A strict comparison will not match.");
}
```

3. Réécrivez l'instruction `for` suivante en utilisant l'opérateur logique unaire NOT dans la condition de la boucle. Le résultat de la condition doit être le même.

```
for ( let factor = 2; factor < candidate; factor++ )
```

Réponse :

```
for ( let factor = 2; ! (factor >= candidate); factor++ )
```

4. En vous basant sur les exemples de cette leçon, écrivez une structure de contrôle en boucle qui affiche tous les facteurs entiers d'un nombre donné.

```
for ( let factor = 2; factor <= my_number; factor++ )
{
  if ( my_number % factor == 0 )
  {
```



```
    console.log(factor, " is an integer factor of ", my_number);  
  }  
}
```



034.3 Leçon 2

Certification :	Web Development Essentials
Version :	1.0
Thème :	034 Programmation JavaScript
Objectif :	034.3 Structures de contrôle et fonctions JavaScript
Leçon :	2 sur 2

Introduction

En plus de l'ensemble standard de fonctions intégrées fournies par le langage JavaScript, les développeurs peuvent écrire leurs propres fonctions personnalisées pour faire correspondre une entrée à une sortie adaptée aux besoins de l'application. Les fonctions personnalisées sont essentiellement un ensemble d'instructions encapsulées pour être utilisées ailleurs dans le cadre d'une expression.

L'utilisation de fonctions est un bon moyen d'éviter la duplication du code, car elles peuvent être appelées à partir de différents endroits du programme. De plus, le regroupement des instructions dans des fonctions facilite la liaison des actions personnalisées aux événements, ce qui est un aspect central de la programmation JavaScript.

Définir une fonction

À fur et à mesure qu'un programme évolue, il devient plus difficile d'organiser ce qu'il fait sans utiliser de fonctions. Chaque fonction possède sa propre portée de variable privée, de sorte que les variables définies dans une fonction ne seront disponibles qu'à l'intérieur de cette même fonction.

Ainsi, elles ne seront pas mélangées avec les variables d'autres fonctions. Les variables globales sont toujours accessibles à l'intérieur des fonctions, mais la meilleure façon d'envoyer des valeurs d'entrée à une fonction est d'utiliser les *paramètres de fonction*. À titre d'exemple, nous allons nous appuyer sur le testeur de nombres premiers de la leçon précédente :

```
// A naive prime number tester

// The number we want to evaluate
let candidate = 231;

// Auxiliary variable
let is_prime = true;

// Start with the lowest prime number after 1
let factor = 2;

// Keeps evaluating while factor is less than the candidate
while ( factor < candidate )
{
    if ( candidate % factor == 0 )
    {
        // The remainder is zero, so the candidate is not prime
        is_prime = false;
        break;
    }

    // The next number that will divide the candidate
    factor++;
}

// Display the result in the console window
if ( is_prime )
{
    console.log(candidate, "is prime");
}
else
{
    console.log(candidate, "is not prime");
}
```

Si plus tard dans le code vous avez besoin de vérifier si un nombre est premier, il serait nécessaire de répéter le code déjà écrit. Cette pratique n'est pas recommandée, car toute correction ou

amélioration du code original devrait être reproduite manuellement partout où le code a été copié. De plus, la répétition du code fait peser une charge sur le navigateur et le réseau, ce qui peut ralentir l’affichage de la page web. Au lieu de procéder ainsi, placez les instructions appropriées dans une fonction :

```
// A naive prime number tester function
function test_prime(candidate)
{
    // Auxiliary variable
    let is_prime = true;

    // Start with the lowest prime number after 1
    let factor = 2;

    // Keeps evaluating while factor is less than the candidate
    while ( factor < candidate )
    {

        if ( candidate % factor == 0 )
        {
            // The remainder is zero, so the candidate is not prime
            is_prime = false;
            break;
        }

        // The next number that will divide the candidate
        factor++;
    }

    // Send the answer back
    return is_prime;
}
```

La déclaration de fonction commence par une instruction `function`, suivie du nom de la fonction et de ses paramètres. Le nom de la fonction doit suivre les mêmes règles que les noms des variables. Les paramètres de la fonction, également appelés *arguments* de la fonction, sont séparés par des virgules et encadrés par des parenthèses.

TIP

Il n’est pas obligatoire de lister les arguments dans la déclaration de la fonction. Les arguments passés à une fonction peuvent être récupérés dans un objet `arguments` de type tableau à l’intérieur de cette fonction. L’index des arguments commence à 0, donc le premier argument est `arguments[0]`, le deuxième argument est `arguments[1]`,

et ainsi de suite.

Dans l'exemple, la fonction `test_prime` n'a qu'un seul argument : l'argument `candidate`, qui est le nombre premier candidat à tester. Les arguments de fonction se comportent comme des variables, mais leurs valeurs sont attribuées par l'instruction qui appelle la fonction. Par exemple, l'instruction `test_prime(231)` appellera la fonction `test_prime` et assignera la valeur 231 à l'argument `candidate`, qui sera alors disponible dans le corps de la fonction comme une variable ordinaire.

Si l'instruction appelante utilise des variables simples pour les paramètres de la fonction, leurs valeurs seront copiées dans les arguments de la fonction. Cette méthode—copier les valeurs des paramètres utilisés dans l'instruction d'appel vers les paramètres utilisés dans la fonction—est appelée *passage des arguments par valeur*. Toute modification apportée à l'argument par la fonction n'affecte pas la variable originale utilisée dans l'instruction appelante. Toutefois, si l'instruction appelante utilise des objets complexes comme arguments (c'est-à-dire un objet auquel sont attachées des propriétés et des méthodes) pour les paramètres de la fonction, ils seront *passés comme référence* et la fonction pourra modifier l'objet original utilisé dans l'instruction d'appel.

Les arguments qui sont passés par valeur, ainsi que les variables déclarées dans la fonction, ne sont pas visibles en dehors de celle-ci. En d'autres termes, leur portée est limitée au corps de la fonction où elles ont été déclarées. Néanmoins, les fonctions sont généralement utilisées pour créer des résultats visibles en dehors de la fonction. Pour partager une valeur avec sa fonction appelante, une fonction définit une instruction `return`.

La fonction `test_prime` de l'exemple précédent renvoie par exemple la valeur de la variable `is_prime`. Par conséquent, la fonction peut remplacer la variable partout où elle serait utilisée dans l'exemple original :

```
// The number we want to evaluate
let candidate = 231;

// Display the result in the console window
if ( test_prime(candidate) )
{
    console.log(candidate, "is prime");
}
else
{
    console.log(candidate, "is not prime");
}
```

L'instruction `return`, comme son nom l'indique, renvoie le contrôle à la fonction appelante. Par

conséquent, où que soit placée l'instruction `return` dans la fonction, rien de ce qui la suit n'est exécuté. Une fonction peut contenir plusieurs instructions `return`. Cette pratique peut être utile si certaines d'entre elles se trouvent dans des blocs d'instructions conditionnelles, de sorte que la fonction peut ou non exécuter une instruction `return` particulière à chaque exécution.

Certaines fonctions peuvent ne pas renvoyer de valeur, l'instruction `return` n'est donc pas obligatoire. Les instructions internes de la fonction sont exécutées indépendamment de sa présence, de sorte que les fonctions peuvent également être utilisées, par exemple, pour modifier les valeurs des variables globales ou le contenu des objets passés par référence. Néanmoins, si la fonction n'a pas d'instruction `return`, sa valeur de retour par défaut est définie comme `undefined` : une variable réservée qui n'a pas de valeur et ne peut pas être modifiée.

Expressions de fonction

En JavaScript, les fonctions ne sont qu'un autre type d'*objet*. Ainsi, les fonctions peuvent être employées dans le script comme des variables. Cette caractéristique devient explicite lorsque la fonction est déclarée à l'aide d'une syntaxe alternative, appelée *expressions de fonction* :

```
let test_prime = function(candidate)
{
    // Auxiliary variable
    let is_prime = true;

    // Start with the lowest prime number after 1
    let factor = 2;

    // Keeps evaluating while factor is less than the candidate
    while ( factor < candidate )
    {
        if ( candidate % factor == 0 )
        {
            // The remainder is zero, so the candidate is not prime
            is_prime = false;
            break;
        }

        // The next number that will divide the candidate
        factor++;
    }

    // Send the answer back
    return is_prime;
}
```

```
}
```

La seule différence entre cet exemple et la déclaration de fonction de l'exemple précédent réside dans la première ligne : `let test_prime = function(candidate)` au lieu de `function test_prime(candidate)`. Dans une expression de fonction, le nom `test_prime` est utilisé pour l'objet contenant la fonction et non pour nommer la fonction elle-même. Les fonctions définies dans les expressions de fonctions sont appelées de la même manière que les fonctions définies à l'aide de la syntaxe de déclaration. Toutefois, alors que les fonctions déclarées peuvent être appelées avant ou après leur déclaration, les expressions de fonction ne peuvent être appelées qu'après leur initialisation. Comme pour les variables, l'appel d'une fonction définie dans une expression avant son initialisation entraîne une erreur de référence.

Fonctions récursives

Outre l'exécution d'instructions et l'appel de fonctions intégrées, les fonctions personnalisées peuvent également appeler d'autres fonctions personnalisées, y compris elles-mêmes. L'appel d'une fonction à partir d'elle-même est appelé *récursivité de la fonction*. Selon le type de problème que vous essayez de résoudre, l'utilisation de fonctions récursives peut être plus simple que l'utilisation de boucles imbriquées pour effectuer des tâches répétitives.

Jusqu'à présent, nous savons comment utiliser une fonction pour vérifier si un nombre donné est premier. Supposons maintenant que vous vouliez trouver le prochain nombre premier suivant un nombre donné. Vous pourriez utiliser une boucle `while` pour incrémenter le nombre candidat et écrire une boucle imbriquée qui cherchera des facteurs entiers pour ce candidat :

```
// This function returns the next prime number
// after the number given as its only argument
function next_prime(from)
{
    // We are only interested in the positive primes,
    // so we will consider the number 2 as the next
    // prime after any number less than two.
    if ( from < 2 )
    {
        return 2;
    }

    // The number 2 is the only even positive prime,
    // so it will be easier to treat it separately.
    if ( from == 2 )
    {
        return 3;
    }
}
```

```
}

// Decrement "from" if it is an even number
if ( from % 2 == 0 )
{
    from--;
}

// Start searching for primes greater then 3.

// The prime candidate is the next odd number
let candidate = from + 2;

// "true" keeps the loop going until a prime is found
while ( true )
{
    // Auxiliary control variable
    let is_prime = true;

    // "candidate" is an odd number, so the loop will
    // try only the odd factors, starting with 3
    for ( let factor = 3; factor < candidate; factor = factor + 2 )
    {
        if ( candidate % factor == 0 )
        {
            // The remainder is zero, so the candidate is not prime.
            // Test the next candidate
            is_prime = false;
            break;
        }
    }
    // End loop and return candidate if it is prime
    if ( is_prime )
    {
        return candidate;
    }
    // If prime not found yet, try the next odd number
    candidate = candidate + 2;
}

let from = 1024;
console.log("The next prime after", from, "is", next_prime(from));
```


Notez que nous devons utiliser une condition constante pour la boucle `while` (l'expression `true` entre parenthèses) et la variable auxiliaire `is_prime` pour savoir quand arrêter la boucle. Bien que cette solution soit correcte, l'utilisation de boucles imbriquées n'est pas aussi élégante que l'utilisation de la récursivité pour effectuer la même tâche :

```
// This function returns the next prime number
// after the number given as its only argument
function next_prime(from)
{
    // We are only interested in the positive primes,
    // so we will consider the number 2 as the next
    // prime after any number less than two.
    if ( from < 2 )
    {
        return 2;
    }

    // The number 2 is the only even positive prime,
    // so it will be easier to treat it separately.
    if ( from == 2 )
    {
        return 3;
    }

    // Decrement "from" if it is an even number
    if ( from % 2 == 0 )
    {
        from--;
    }

    // Start searching for primes greater than 3.

    // The prime candidate is the next odd number
    let candidate = from + 2;

    // "candidate" is an odd number, so the loop will
    // try only the odd factors, starting with 3
    for ( let factor = 3; factor < candidate; factor = factor + 2 )
    {
        if ( candidate % factor == 0 )
        {
            // The remainder is zero, so the candidate is not prime.
            // Call the next_prime function recursively, this time
            // using the failed candidate as the argument.
        }
    }
}
```

```
        return next_prime(candidate);
    }
}

// "candidate" is not divisible by any integer factor other
// than 1 and itself, therefore it is a prime number.
return candidate;
}

let from = 1024;
console.log("The next prime after", from, "is", next_prime(from));
```

Les deux versions de `next_prime` retournent le nombre premier suivant le nombre donné comme seul argument (`from`). La version récursive, comme la version précédente, commence par vérifier les cas particuliers (c'est-à-dire les nombres inférieurs ou égaux à deux). Ensuite, elle incrémente le candidat et commence à chercher tous les facteurs entiers avec la boucle `for` (remarquez que la boucle `while` n'est plus là). À ce stade, le seul nombre premier pair a déjà été testé, donc le candidat et ses facteurs possibles sont incrémentés de deux. (Un nombre impair plus deux est le prochain nombre impair).

Il n'y a que deux façons de sortir de la boucle `for` dans l'exemple. Si tous les facteurs possibles sont testés et qu'aucun d'entre eux n'a un reste égal à zéro lors de la division du candidat, la boucle `for` se termine et la fonction renvoie le candidat comme le prochain nombre premier après `from`. Sinon, si facteur est un facteur entier de candidat (`candidat % facteur == 0`), la valeur retournée provient de la fonction `next_prime` appelée récursivement, cette fois avec le candidat incrémenté comme paramètre `from`. Les appels à `next_prime` seront empilés les uns sur les autres, jusqu'à ce qu'un candidat ne trouve finalement aucun facteur entier. Ensuite, la dernière instance de `next_prime` contenant le nombre premier le renverra à l'instance précédente de `next_prime`, et ainsi successivement jusqu'à la première instance de `next_prime`. Même si chaque appel de la fonction utilise les mêmes noms pour les variables, les appels sont isolés les uns des autres, de sorte que leurs variables sont conservées séparément en mémoire.

Exercices guidés

1. Quel type de surcharge les développeurs peuvent-ils atténuer en utilisant des fonctions ?

2. Quelle est la différence entre les arguments de fonction passés par valeur et les arguments de fonction passés par référence ?

3. Quelle valeur sera utilisée comme sortie d'une fonction personnalisée si elle n'a pas d'instruction de retour ?

Exercices d'exploration

1. Quelle est la cause probable d'un *Uncaught Reference Error* émis lors de l'appel d'une fonction déclarée avec la syntaxe *expression* ?

2. Ecrivez une fonction appelée `multiples_of` qui reçoit trois arguments : `factor`, `from` et `to`. À l'intérieur de la fonction, utilisez l'instruction `console.log()` pour afficher tous les multiples de `factor` compris entre `from` et `to`.

Résumé

Cette leçon explique comment écrire des fonctions personnalisées dans le code JavaScript. Les fonctions personnalisées permettent au développeur de diviser l'application en “morceaux” de code réutilisable, ce qui facilite l'écriture et la maintenance de programmes plus importants. La leçon aborde les concepts et procédures suivants :

- Comment définir une fonction personnalisée : déclarations de fonction et expressions de fonction.
- Utilisation des paramètres comme entrée de la fonction.
- Utilisation de l'instruction `return` pour définir la sortie de la fonction.
- Fonctions récursives.

Réponses aux exercices guidés

1. Quel type de surcharge les développeurs peuvent-ils atténuer en utilisant des fonctions ?

Les fonctions nous permettent de réutiliser le code, ce qui facilite sa maintenance. Un fichier de script plus petit permet également d'économiser de la mémoire et du temps de téléchargement.

2. Quelle est la différence entre les arguments de fonction passés par valeur et les arguments de fonction passés par référence ?

Lorsqu'il est passé par valeur, l'argument est copié dans la fonction et la fonction n'est pas en mesure de modifier la variable originale dans l'instruction appelante. Lorsqu'il est transmis par référence, la fonction est en mesure de manipuler la variable originale utilisée dans l'instruction appelante.

3. Quelle valeur sera utilisée comme sortie d'une fonction personnalisée si elle n'a pas d'instruction de retour ?

La valeur retournée sera définie comme `undefined`.

Réponses aux exercices d'exploration

1. Quelle est la cause probable d'un *Uncaught Reference Error* émis lors de l'appel d'une fonction déclarée avec la syntaxe *expression* ?

La fonction a été appelée avant sa déclaration dans le fichier de script.

2. Ecrivez une fonction appelée `multiples_of` qui reçoit trois arguments : `factor`, `from` et `to`. À l'intérieur de la fonction, utilisez l'instruction `console.log()` pour afficher tous les multiples de `factor` compris entre `from` et `to`.

```
function multiples_of(factor, from, to)
{
  for ( let number = from; number <= to; number++ )
  {
    if ( number % factor == 0 )
    {
      console.log(factor, "×", number / factor, "=", number);
    }
  }
}
```



**Linux
Professional
Institute**

034.4 Manipulation en JavaScript du contenu et du style des sites web

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 034.4

Valeur

4

Domaines de connaissance les plus importants

- Comprendre le concept et la structure de l'interface DOM
- Modifier le contenu et les propriétés des éléments HTML par le biais de l'interface DOM.
- Modifier le style CSS des éléments HTML par le biais de l'interface DOM.
- Déclencher des fonctions JavaScript à partir d'éléments HTML

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `document.getElementById()`, `document.getElementsByClassName()`,
`document.getElementsByTagName()`, `document.querySelector()`,
`document.querySelectorAll()`
- Propriétés et méthodes `innerHTML`, `setAttribute()`, `removeAttribute()` des éléments de l'interface DOM
- Propriétés et méthodes `classList`, `classList.add()`, `classList.remove()`, `classList.toggle()` des éléments de l'interface DOM
- Attributs des éléments HTML `onClick`, `onmouseover`, `onmouseout`



034.4 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	034 Programmation JavaScript
Objectif :	034.4 Manipulation en JavaScript du contenu et du style des sites web
Leçon :	1 sur 1

Introduction

HTML, CSS et JavaScript sont trois technologies distinctes qui se rejoignent sur le web. Pour créer des pages véritablement dynamiques et interactives, le programmeur JavaScript doit combiner des composants HTML et CSS au moment de l'exécution, une tâche qui est grandement facilitée par l'utilisation du *modèle objet de document* (DOM : *Document Object Model*).

Interagir avec le DOM

Le DOM est une structure de données qui fonctionne comme une interface de programmation pour le document, où chaque aspect du document est représenté comme un nœud dans le DOM et chaque changement apporté au DOM se répercute immédiatement dans le document. Pour montrer comment utiliser le DOM en JavaScript, enregistrez le code HTML suivant dans un fichier appelé `exemple.html` :

```
<!DOCTYPE html>
<html>
<head>
```

```
<meta charset="utf-8" />
<title>HTML Manipulation with JavaScript</title>
</head>
<body>

<div class="content" id="content_first">
<p>The dynamic content goes here</p>
</div><!-- #content_first -->

<div class="content" id="content_second" hidden>
<p>Second section</p>
</div><!-- #content_second -->

</body>
</html>
```

Le DOM ne sera disponible qu'après le chargement du HTML, donc écrivez le code JavaScript suivant à la fin du corps de la page (avant la balise de fin `</body>`) :

```
<script>
let body = document.getElementsByTagName("body")[0];
console.log(body.innerHTML);
</script>
```

L'objet `document` est l'élément supérieur du DOM, tous les autres éléments en sont dérivés. La méthode `getElementsByTagName()` liste tous les éléments descendant de `document` qui ont le nom de balise donné. Même si la balise `body` n'est utilisée qu'une seule fois dans le document, la méthode `getElementsByTagName()` renvoie toujours un tableau des éléments trouvés, d'où l'utilisation de l'index `[0]` pour renvoyer le premier (et seul) élément trouvé.

Contenu HTML

Comme le montre l'exemple précédent, l'élément DOM renvoyé par la commande `document.getElementsByTagName("body")[0]` a été affecté à la variable `body`. La variable `body` peut alors être utilisée pour manipuler l'élément `'<body>'` de la page, car elle hérite de toutes les méthodes et attributs DOM de cet élément. Par exemple, la propriété `innerHTML` contient l'intégralité du code de balisage HTML écrit à l'intérieur de l'élément correspondant, elle peut donc être utilisée pour lire le balisage interne. Notre appel `console.log(body.innerHTML)` affiche le contenu de `<body></body>` dans la console web. La variable peut également être utilisée pour remplacer ce contenu, comme dans `body.innerHTML = "<p>Content erased</p>"`.

Plutôt que de modifier des portions entières du balisage HTML, il est plus pratique de ne pas modifier la structure du document et de n'interagir qu'avec ses éléments. Une fois le document rendu par le navigateur, tous les éléments sont accessibles par les méthodes DOM. Il est possible, par exemple, de lister et d'accéder à tous les éléments HTML en utilisant la chaîne spéciale `*` dans la méthode `getElementsByName()` de l'objet `document` :

```
let elements = document.getElementsByTagName("*");
for ( element of elements )
{
    if ( element.id == "content_first" )
    {
        element.innerHTML = "<p>New content</p>";
    }
}
```

Ce code va placer tous les éléments trouvés dans `document` dans la variable `elements`. La variable `elements` est un objet de type tableau, nous pouvons donc itérer à travers chacun de ses éléments avec une boucle `for`. Si la page HTML sur laquelle s'exécute ce code contient un élément dont l'attribut `id` est défini sur `content_first` (voir l'exemple de page HTML présenté au début de la leçon), l'instruction `if` correspond à cet élément et son contenu sera modifié en `<p>New content</p>`. Notez que les attributs d'un élément HTML dans le DOM sont accessibles en utilisant la notation *point* des propriétés des objets JavaScript : ainsi, `element.id` fait référence à l'attribut `id` de l'élément courant de la boucle `for`. La méthode `getAttribute()` peut également être utilisée, comme dans `element.getAttribute("id")`.

Il n'est pas nécessaire d'itérer à travers tous les éléments si vous voulez inspecter seulement un sous-ensemble d'entre eux. Par exemple, la méthode `document.getElementsByClassName()` limite les éléments trouvés à ceux qui ont une classe spécifique :

```
let elements = document.getElementsByClassName("content");
for ( element of elements )
{
    if ( element.id == "content_first" )
    {
        element.innerHTML = "<p>New content</p>";
    }
}
```

Cependant, l'itération à travers de nombreux éléments du document à l'aide d'une boucle n'est pas la meilleure stratégie lorsque vous devez modifier un élément spécifique de la page.

Sélection d'éléments spécifiques

JavaScript fournit des méthodes optimisées pour sélectionner l'élément exact sur lequel vous voulez travailler. La boucle précédente pourrait être entièrement remplacée par la méthode `document.getElementById()` :

```
let element = document.getElementById("content_first");
element.innerHTML = "<p>New content</p>";
```

Chaque attribut `id` du document doit être unique, donc la méthode `document.getElementById()` ne retourne qu'un seul objet DOM. Même la déclaration de la variable `element` peut être omise, car JavaScript nous permet d'enchaîner directement les méthodes :

```
document.getElementById("content_first").innerHTML = "<p>New content</p>";
```

La méthode `getElementById()` est la méthode préférable pour localiser des éléments dans le DOM, car ses performances sont bien meilleures que les méthodes itératives lorsqu'on travaille avec des documents complexes. Cependant, tous les éléments n'ont pas un identifiant explicite, et la méthode renvoie une valeur *null* si aucun élément ne correspond à l'identifiant fourni (cela empêche également l'utilisation d'attributs ou de fonctions enchaînés, comme le `innerHTML` utilisé dans l'exemple ci-dessus). De plus, il est plus pratique d'attribuer des attributs `id` uniquement aux principaux composants de la page, puis d'utiliser des sélecteurs CSS pour localiser leurs éléments enfants.

Les sélecteurs, présentés dans une leçon précédente sur les CSS, sont des motifs qui correspondent à des éléments dans le DOM. La méthode `querySelector()` renvoie le premier élément correspondant dans l'arbre du DOM, tandis que `querySelectorAll()` renvoie tous les éléments qui correspondent au sélecteur spécifié.

Dans l'exemple précédent, la méthode `getElementById()` récupère l'élément portant l'identifiant `content_first`. La méthode `querySelector()` peut effectuer la même tâche :

```
document.querySelector("#content_first").innerHTML = "<p>New content</p>";
```

Comme la méthode `querySelector()` utilise la syntaxe des sélecteurs, l'identifiant fourni doit commencer par un caractère dièse. Si aucun élément correspondant n'est trouvé, la méthode `querySelector()` renvoie *null*.

Dans l'exemple précédent, le contenu entier de la section `content_first` est remplacé par la chaîne

de texte fournie. Cette chaîne contient du code HTML, ce qui n'est pas considéré comme une bonne pratique. Vous devez être prudent lorsque vous ajoutez des balises HTML codées en dur au code JavaScript, car le suivi des éléments peut devenir difficile lorsque des modifications de la structure globale du document sont nécessaires.

Les sélecteurs ne sont pas limités à l'identifiant de l'élément. L'élément interne `p` peut être adressé directement :

```
document.querySelector("#content_first p").innerHTML = "New content";
```

Le sélecteur `#content_first p` ne correspondra qu'au premier élément `p` à l'intérieur de la section `#content_first`. Cela fonctionne bien si nous voulons manipuler le premier élément. Cependant, nous pouvons vouloir modifier le deuxième paragraphe :

```
<div class="content" id="content_first">
<p>Don't change this paragraph.</p>
<p>The dynamic content goes here.</p>
</div><!-- #content_first -->
```

Dans ce cas, nous pouvons utiliser la pseudo-classe `:nth-child(2)` pour faire correspondre le deuxième élément `p` :

```
document.querySelector("#content_first p:nth-child(2)").innerHTML = "New content";
```

Le chiffre 2 dans `p:nth-child(2)` indique le deuxième paragraphe qui correspond au sélecteur. Consultez la leçon sur les sélecteurs CSS pour en savoir plus sur les sélecteurs et leur utilisation.

Travailler avec les attributs

La capacité de JavaScript à interagir avec le DOM ne se limite pas à la manipulation du contenu. En effet, l'utilisation la plus répandue de JavaScript dans le navigateur consiste à modifier les attributs des éléments HTML existants.

Disons que notre page d'exemple HTML originale comporte maintenant trois sections de contenu :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
```

```
<title>HTML Manipulation with JavaScript</title>
</head>
<body>

<div class="content" id="content_first" hidden>
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second" hidden>
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third" hidden>
<p>Third section.</p>
</div><!-- #content_third -->

</body>
</html>
```

Vous pouvez vouloir n'en rendre qu'une seule visible à la fois, d'où l'attribut `hidden` dans toutes les balises `div`. C'est utile, par exemple, pour ne montrer qu'une seule image d'une galerie d'images. Pour rendre l'une d'entre elles visible au chargement de la page, ajoutez le code JavaScript suivant à la page :

```
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
    break;
  case 1:
    content_visible = "#content_second";
    break;
  case 2:
    content_visible = "#content_third";
    break;
}

document.querySelector(content_visible).removeAttribute("hidden");
```

L'expression évaluée par l'instruction `switch` renvoie aléatoirement le nombre 0, 1 ou 2. L'identifiant

du sélecteur correspondant est alors assigné à la variable `content_visible`, qui est utilisée par la méthode `querySelector(content_visible)`. L'appel enchaîné `removeAttribute("hidden")` supprime l'attribut `hidden` de l'élément.

L'approche inverse est également possible : Toutes les sections pourraient être initialement visibles (sans l'attribut `hidden`) et le programme JavaScript peut alors attribuer l'attribut `hidden` à toutes les sections sauf celle qui se trouve dans `content_visible`. Pour ce faire, vous devez itérer à travers le contenu de tous les éléments `div` qui sont différents de celui choisi, ce qui peut être fait en utilisant la méthode `querySelectorAll()` :

```
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
    break;
  case 1:
    content_visible = "#content_second";
    break;
  case 2:
    content_visible = "#content_third";
    break;
}

// Hide all content divs, except content_visible
for ( element of document.querySelectorAll(".content:not("+content_visible+")") )
{
  // Hidden is a boolean attribute, so any value will enable it
  element.setAttribute("hidden", "");
}
```

Si la variable `content_visible` a été définie sur `#content_first`, le sélecteur sera `.content:not(#content_first)`, ce qui signifie que tous les éléments ayant la classe `content` sauf ceux ayant l'identifiant `content_first`. La méthode `setAttribute()` ajoute ou modifie les attributs des éléments HTML. Son premier paramètre est le nom de l'attribut et le second est la valeur de l'attribut.

Cependant, le moyen le plus approprié pour modifier l'apparence des éléments est le CSS. Dans ce cas, nous pouvons définir la propriété CSS `display` sur `hidden` et ensuite la changer en `block` en utilisant JavaScript :

```
<style>
div.content { display: none }
</style>

<div class="content" id="content_first">
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second">
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third">
<p>Third section.</p>
</div><!-- #content_third -->

<script>
// Which content to show
let content_visible;

switch ( Math.floor(Math.random() * 3) )
{
  case 0:
    content_visible = "#content_first";
    break;
  case 1:
    content_visible = "#content_second";
    break;
  case 2:
    content_visible = "#content_third";
    break;
}
document.querySelector(content_visible).style.display = "block";
</script>
```

Les mêmes bonnes pratiques qui s'appliquent au mélange de balises HTML et de JavaScript s'appliquent également au CSS. Ainsi, il n'est pas recommandé d'écrire les propriétés CSS directement dans le code JavaScript. Au contraire, les règles CSS doivent être écrites séparément du code JavaScript. La bonne façon d'alterner le style visuel est de sélectionner une classe CSS prédéfinie pour l'élément.

Travailler avec les classes

Les éléments peuvent avoir plus d'une classe associée, ce qui facilite l'écriture de styles qui peuvent être ajoutés ou supprimés si nécessaire. Il serait épuisant de modifier de nombreux attributs CSS directement en JavaScript. Vous pouvez donc créer une nouvelle classe CSS avec ces attributs, puis ajouter la classe à l'élément. Les éléments DOM possèdent la propriété `classList`, qui peut être utilisée pour visualiser et manipuler les classes attribuées à l'élément correspondant.

Par exemple, au lieu de modifier la visibilité de l'élément, nous pouvons créer une classe CSS supplémentaire pour mettre en évidence le contenu `content` de notre élément `div` :

```
div.content {  
  border: 1px solid black;  
  opacity: 0.25;  
}  
div.content.highlight {  
  border: 1px solid red;  
  opacity: 1;  
}
```

Cette feuille de style va ajouter une fine bordure noire et une semi-transparence à tous les éléments ayant la classe `content`. Seuls les éléments qui ont également la classe `highlight` seront totalement opaques et auront une fine bordure rouge. Ensuite, au lieu de modifier les propriétés CSS directement comme nous l'avons fait auparavant, nous pouvons utiliser la méthode `classList.add("highlight")` dans l'élément sélectionné :

```
// Which content to highlight  
let content_highlight;  
  
switch ( Math.floor(Math.random() * 3) )  
{  
  case 0:  
    content_highlight = "#content_first";  
    break;  
  case 1:  
    content_highlight = "#content_second";  
    break;  
  case 2:  
    content_highlight = "#content_third";  
    break;  
}
```

```
// Highlight the selected div
document.querySelector(content_highlight).classList.add("highlight");
```

Toutes les techniques et tous les exemples que nous avons vus jusqu'à présent ont été réalisés à la fin du processus de chargement de la page, mais ils ne sont pas limités à cette étape. En fait, ce qui rend JavaScript si utile aux développeurs web, c'est sa capacité à réagir aux événements survenant sur la page, ce que nous verrons ensuite.

Gestionnaires d'événements

Tous les éléments visibles de la page sont susceptibles de subir des événements interactifs, tels que le clic ou le mouvement de la souris elle-même. Nous pouvons associer des actions personnalisées à ces événements, ce qui étend considérablement les possibilités d'un document HTML.

L'élément HTML le plus évident qui bénéficie d'une action associée est probablement l'élément bouton. Pour montrer comment cela fonctionne, ajoutez trois boutons au-dessus du premier élément div de la page d'exemple :

```
<p>
<button>First</button>
<button>Second</button>
<button>Third</button>
</p>

<div class="content" id="content_first">
<p>First section.</p>
</div><!-- #content_first -->

<div class="content" id="content_second">
<p>Second section.</p>
</div><!-- #content_second -->

<div class="content" id="content_third">
<p>Third section.</p>
</div><!-- #content_third -->
```

Les boutons ne font rien par eux-mêmes, mais supposons que vous vouliez mettre en évidence la div correspondant au bouton pressé. Nous pouvons utiliser l'attribut `onClick` pour associer une action à chaque bouton :

```
<p>
```

```

<button
onClick="document.getElementById('content_first').classList.toggle('highlight')">Fi
rst</button>
<button
onClick="document.getElementById('content_second').classList.toggle('highlight')">S
econd</button>
<button
onClick="document.getElementById('content_third').classList.toggle('highlight')">Th
ird</button>
</p>

```

La méthode `classList.toggle()` ajoute la classe spécifiée à l'élément si elle n'est pas présente, et supprime cette classe si elle est déjà présente. Si vous exécutez l'exemple, vous remarquerez que plus d'un `div` peut être mis en évidence en même temps. Pour mettre en évidence uniquement le `div` correspondant au bouton pressé, il est nécessaire de supprimer la classe `highlight` des autres éléments `div`. Néanmoins, si l'action personnalisée est trop longue ou implique plus d'une ligne de code, il est plus pratique d'écrire une fonction en dehors de la balise de l'élément :

```

function highlight(id)
{
  // Remove the "highlight" class from all content elements
  for ( element of document.querySelectorAll(".content") )
  {
    element.classList.remove('highlight');
  }

  // Add the "highlight" class to the corresponding element
  document.getElementById(id).classList.add('highlight');
}

```

Comme les exemples précédents, cette fonction peut être placée dans une balise `<script>` ou dans un fichier JavaScript externe associé au document. La fonction `highlight` supprime d'abord la classe `highlight` de tous les éléments `div` associés à la classe `content`, puis ajoute la classe `highlight` à l'élément choisi. Chaque bouton doit ensuite appeler cette fonction depuis son attribut `onClick`, en utilisant l'identifiant correspondant comme argument de la fonction :

```

<p>
<button onClick="highlight('content_first')">First</button>
<button onClick="highlight('content_second')">Second</button>
<button onClick="highlight('content_third')">Third</button>
</p>

```

En plus de l'attribut `onClick`, nous pourrions utiliser l'attribut `onMouseOver` (déclenché lorsque le dispositif de pointage est utilisé pour déplacer le curseur sur l'élément), l'attribut `onMouseOut` (déclenché lorsque le dispositif de pointage n'est plus contenu dans l'élément), etc. De plus, les gestionnaires d'événements ne sont pas limités aux boutons, vous pouvez donc leur attribuer des actions personnalisées pour tous les éléments HTML visibles.

Exercices guidés

1. En utilisant la méthode `document.getElementById()`, comment pourriez-vous insérer la phrase “Dynamic content” dans le contenu interne de l’élément dont l’identifiant est `message` ?

2. Quelle est la différence entre référencer un élément par son identifiant en utilisant la méthode `document.querySelector()` et le faire via la méthode `document.getElementById()` ?

3. Quel est l’objectif de la méthode `classList.remove()` ?

4. Quel est le résultat de l’utilisation de la méthode `myelement.classList.toggle("active")` si `myelement` n’est pas associé à la classe `active` ?

Exercices d'exploration

1. Quel argument de la méthode `document.querySelectorAll()` lui fera imiter la méthode `document.getElementsByTagName("input")` ?

2. Comment utiliser la propriété `classList` pour lister toutes les classes associées à un élément donné ?

Résumé

Cette leçon explique comment utiliser JavaScript pour modifier les contenus HTML et leurs propriétés CSS à l'aide du DOM (Document Object Model). Ces modifications peuvent être déclenchées par des événements utilisateur, ce qui est utile pour créer des interfaces dynamiques. La leçon aborde les concepts et procédures suivants :

- Comment inspecter la structure du document en utilisant des méthodes comme `document.getElementById()`, `document.getElementsByClassName()`, `document.getElementsByTagName()`, `document.querySelector()` et `document.querySelectorAll()`.
- Comment modifier le contenu du document avec la propriété `innerHTML`.
- Comment ajouter et modifier les attributs des éléments de la page avec les méthodes `setAttribute()` et `removeAttribute()`.
- La manière correcte pour manipuler les classes d'éléments en utilisant la propriété `classList` et sa relation avec les styles CSS.
- Comment lier des fonctions aux événements de souris dans des éléments spécifiques.

Réponses aux exercices guidés

1. En utilisant la méthode `document.getElementById()`, comment pourriez-vous insérer la phrase “Dynamic content” dans le contenu interne de l’élément dont l’identifiant est `message` ?

Vous pouvez le faire avec la propriété `innerHTML` :

```
document.getElementById("message").innerHTML = "Dynamic content"
```

2. Quelle est la différence entre référencer un élément par son identifiant en utilisant la méthode `document.querySelector()` et le faire via la méthode `document.getElementById()` ?

L’identifiant doit être accompagné du caractère dièse dans les fonctions qui utilisent des sélecteurs, comme `document.querySelector()`.

3. Quel est l’objectif de la méthode `classList.remove()` ?

Elle supprime la classe (dont le nom est donné en argument de la fonction) de l’attribut `class` de l’élément correspondant.

4. Quel est le résultat de l’utilisation de la méthode `myelement.classList.toggle("active")` si `myelement` n’est pas associé à la classe `active` ?

Cette méthode attribuera la classe `active` à `myelement`.

Réponses aux exercices d'exploration

1. Quel argument de la méthode `document.querySelectorAll()` lui fera imiter la méthode `document.getElementsByTagName("input")` ?

En utilisant la méthode `document.querySelectorAll("input")`, vous ferez correspondre tous les éléments `input` de la page, tout comme `document.getElementsByTagName("input")`.

2. Comment utiliser la propriété `classList` pour lister toutes les classes associées à un élément donné ?

La propriété `classList` est un objet de type tableau, donc une boucle `for` peut être utilisée pour itérer à travers toutes les classes qu'elle contient.



Thème 035: Programmation de serveur NodeJS



**Linux
Professional
Institute**

035.1 Bases de Node.js

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 035.1

Valeur

1

Domaines de connaissance les plus importants

- Comprendre les concepts de Node.js
- Exécuter une application NodeJS
- Installer des paquets NPM

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- `node [file.js]`
- `npm init`
- `npm install [module_name]`
- `package.json`
- `node_modules`



035.1 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	035 Programmation de serveur NodeJS
Objectif :	035.1 Bases de NodeJS
Leçon :	1 sur 1

Introduction

Node.js est un environnement d'exécution JavaScript qui exécute du code JavaScript dans les serveurs Web - ce que l'on appelle le *backend* web (côté serveur) - au lieu d'utiliser un deuxième langage comme Python ou Ruby pour les programmes côté serveur. Le langage JavaScript est déjà utilisé dans le côté frontal moderne des applications web, interagissant avec le HTML et le CSS de l'interface avec laquelle l'utilisateur interagit dans un navigateur web. L'utilisation de Node.js en tandem avec JavaScript dans le navigateur offre la possibilité d'utiliser un seul langage de programmation pour l'ensemble de l'application.

La principale raison de l'existence de Node.js est la façon dont il gère les connexions multiples et simultanées dans le backend. L'une des façons les plus courantes dont un serveur d'applications web gère les connexions est l'exécution de plusieurs processus. Lorsque vous ouvrez une application de bureau sur votre ordinateur, un processus démarre et utilise beaucoup de ressources. Imaginez maintenant que des milliers d'utilisateurs fassent la même chose dans une grande application web.

Node.js évite ce problème en utilisant une conception appelée la *boucle d'événement* (*event loop* en anglais), qui est une boucle interne qui vérifie continuellement les tâches entrantes à calculer. Grâce à l'utilisation répandue de JavaScript et à l'omniprésence des technologies web, Node.js a connu une

adoption massive dans les petites et grandes applications. D'autres caractéristiques ont également contribué à la large adoption de Node.js, comme le traitement asynchrone et non bloquant des entrées/sorties (E/S), qui est expliqué plus loin dans cette leçon.

L'environnement Node.js utilise un moteur JavaScript pour interpréter et exécuter le code JavaScript du côté serveur ou sur le bureau. Dans ces conditions, le code JavaScript que le programmeur écrit est analysé et compilé juste à temps pour exécuter les instructions machine générées par le code JavaScript original.

NOTE

Au fur et à mesure que vous progresserez dans ces leçons sur Node.js, vous remarquerez peut-être que le JavaScript de Node.js n'est pas exactement le même que celui qui s'exécute sur le navigateur (qui suit la [spécification ECMAScript](#)), mais qu'il est assez similaire.

Pour commencer

Cette section et les exemples suivants supposent que Node.js est déjà installé sur votre système d'exploitation Linux, et que l'utilisateur possède déjà des compétences de base telles que l'exécution de commandes dans le terminal.

Pour exécuter les exemples suivants, créez un répertoire de travail appelé `node_examples`. Ouvrez un terminal et tapez `node`. Si vous avez correctement installé Node.js, il présentera une invite `>` où vous pourrez tester les commandes JavaScript de manière interactive. Ce type d'environnement est appelé REPL, pour “read, evaluate, print, and loop” (en français : “lire, évaluer, afficher et boucler”). Tapez les entrées suivantes (ou d'autres instructions JavaScript) à l'invite `>`. Appuyez sur la touche Entrée après chaque ligne, et l'environnement REPL vous renverra les résultats de ses actions :

```
> let array = ['a', 'b', 'c', 'd'];
undefined
> array.map( (element, index) => (`Element: ${element} at index: ${index}`));
[
  'Element: a at index: 0',
  'Element: b at index: 1',
  'Element: c at index: 2',
  'Element: d at index: 3'
]
>
```

L'extrait a été écrit en utilisant la syntaxe ES6, qui propose une fonction `map` pour itérer sur le tableau et afficher les résultats à l'aide de modèles de chaîne. Vous pouvez écrire à peu près n'importe quelle commande qui soit valide. Pour quitter le terminal Node.js, tapez `.exit`, en

n'oubliant pas d'inclure le point initial.

Pour les scripts et modules plus longs, il est plus pratique d'utiliser un éditeur de texte tel que VS Code, Emacs ou Vim. Vous pouvez enregistrer les deux lignes de code que vous venez de voir (avec une petite modification) dans un fichier appelé `start.js` :

```
let array = ['a', 'b', 'c', 'd'];
array.map( (element, index) => ( console.log(`Element: ${element} at index: ${index}`)) );
```

Vous pouvez ensuite exécuter le script à partir du Shell pour obtenir les mêmes résultats que précédemment :

```
$ node ./start.js
Element: a at index: 0
Element: b at index: 1
Element: c at index: 2
Element: d at index: 3
```

Avant de plonger dans un peu plus de code, nous allons avoir un aperçu du fonctionnement de Node.js, en utilisant son environnement d'exécution à fil unique et la boucle d'événement.

Boucle d'événement et fil unique

Il est difficile de dire combien de temps votre programme Node.js prendra pour traiter une requête. Certaines requêtes peuvent être courtes - peut-être simplement en bouclant sur des variables en mémoire et en les retournant - alors que d'autres peuvent nécessiter des activités longues comme l'ouverture d'un fichier sur le système ou l'envoi d'une requête à une base de données et l'attente des résultats. Comment Node.js gère-t-il cette incertitude ? La réponse est la boucle d'événement.

Imaginez un chef cuisinier effectuant plusieurs tâches. La cuisson d'un gâteau est une tâche qui nécessite beaucoup de temps pour que le four le prépare. Le chef ne reste pas là à attendre que le gâteau soit prêt pour ensuite aller faire du café. Au contraire, pendant que le four cuit le gâteau, le chef prépare le café et d'autres tâches en parallèle. Mais le cuisinier vérifie toujours si c'est le bon moment pour se concentrer sur une tâche spécifique (faire du café) ou pour sortir le gâteau du four.

La boucle d'événement est comme le chef qui est constamment au courant des activités environnantes. Dans Node.js, un "event-checker" vérifie en permanence les opérations qui se sont terminées ou qui attendent d'être traitées par le moteur JavaScript.

Grâce à cette approche, une opération longue et asynchrone ne bloque pas les autres opérations rapides qui suivent. En effet, le mécanisme de boucle d'événement vérifie toujours si cette longue tâche, telle qu'une opération d'E/S, est déjà terminée. Si ce n'est pas le cas, Node.js peut continuer à traiter d'autres tâches. Une fois la tâche d'arrière-plan terminée, les résultats sont renvoyés et l'application située au-dessus de Node.js peut utiliser une fonction de déclenchement (rappel ou *callback*) pour poursuivre le traitement de la sortie.

Parce que Node.js évite l'utilisation de plusieurs fils, comme le font d'autres environnements, on l'appelle un *environnement à fil unique*, et donc une approche non bloquante est de la plus haute importance. C'est pourquoi Node.js utilise une boucle d'événement. Pour les tâches nécessitant des calculs intensifs, Node.js ne fait cependant pas partie des meilleurs outils : il existe d'autres langages et environnements de programmation qui traitent ces problèmes plus efficacement.

Dans les sections suivantes, nous allons examiner de plus près les fonctions de rappel. Pour l'instant, comprenez que les fonctions de rappel sont des déclencheurs qui sont exécutés à la fin d'une opération prédéfinie.

Modules

C'est une bonne pratique que de décomposer une fonctionnalité complexe et des morceaux de code importants en parties plus petites. Cette modularisation permet de mieux organiser la base de code, d'abstraire les implémentations et d'éviter les problèmes d'ingénierie compliqués. Pour répondre à ces besoins, les programmeurs regroupent des blocs de code source qui seront consommés par d'autres parties de code internes ou externes.

Prenons l'exemple d'un programme qui calcule le volume d'une sphère. Ouvrez votre éditeur de texte et créez un fichier nommé `volumeCalculator.js` contenant le code JavaScript suivant :

```
const sphereVol = (radius) => {
  return 4 / 3 * Math.PI * radius
}

console.log(`A sphere with radius 3 has a ${sphereVol(3)} volume.`);
console.log(`A sphere with radius 6 has a ${sphereVol(6)} volume.`);
```

Maintenant, exécutez le fichier en utilisant Node :

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
A sphere with radius 6 has a 904.7786842338603 volume.
```

Ici, une simple fonction a été utilisée pour calculer le volume d'une sphère, en fonction de son rayon. Imaginez que nous ayons également besoin de calculer le volume d'un cylindre, d'un cône, etc. : nous remarquons rapidement que ces fonctions spécifiques doivent être ajoutées au fichier `volumeCalculator.js`, qui peut devenir une énorme collection de fonctions. Pour mieux organiser la structure, nous pouvons utiliser l'idée derrière les modules comme des paquets de code séparé.

Pour ce faire, créez un fichier séparé appelé `polyhedrons.js` :

```
const coneVol = (radius, height) => {
  return 1 / 3 * Math.PI * Math.pow(radius, 2) * height;
}

const cylinderVol = (radius, height) => {
  return Math.PI * Math.pow(radius, 2) * height;
}

const sphereVol = (radius) => {
  return 4 / 3 * Math.PI * Math.pow(radius, 3);
}

module.exports = {
  coneVol,
  cylinderVol,
  sphereVol
}
```

Maintenant, dans le fichier `volumeCalculator.js`, supprimez l'ancien code et remplacez-le par ce bout de code :

```
const polyhedrons = require('./polyhedrons.js');

console.log(`A sphere with radius 3 has a ${polyhedrons.sphereVol(3)} volume.`);
console.log(`A cylinder with radius 3 and height 5 has a ${polyhedrons.cylinderVol(3, 5)} volume.`);
console.log(`A cone with radius 3 and height 5 has a ${polyhedrons.coneVol(3, 5)} volume.`);
```

Et ensuite exécuter le nom du fichier dans l'environnement Node.js :

```
$ node volumeCalculator.js
A sphere with radius 3 has a 113.09733552923254 volume.
```



```
A cylinder with radius 3 and height 5 has a 141.3716694115407 volume.  
A cone with radius 3 and height 5 has a 47.12388980384689 volume.
```

Dans l'environnement Node.js, chaque fichier de code source est considéré comme un module, mais le mot “module” dans Node.js indique un paquet de code enveloppé comme dans l'exemple précédent. En utilisant des modules, nous avons abstrait les fonctions de volume du fichier principal, `volumeCalculator.js`, réduisant ainsi sa taille et facilitant l'application de tests unitaires, ce qui est une bonne pratique lors du développement d'applications du monde réel.

Maintenant que nous savons comment les modules sont utilisés dans Node.js, nous pouvons utiliser l'un des outils les plus importants : le *Node Package Manager* (NPM).

L'une des principales tâches de NPM est de gérer, télécharger et installer des modules externes dans le projet ou dans le système d'exploitation. Vous pouvez initialiser un dépôt node avec la commande `npm init`.

NPM posera les questions par défaut sur le nom de votre dépôt, la version, la description, etc. Vous pouvez contourner ces étapes en utilisant `npm init --yes`, et la commande générera automatiquement un fichier `package.json` qui décrit les propriétés de votre projet/module.

Ouvrez le fichier `package.json` dans votre éditeur de texte préféré et vous verrez un fichier JSON contenant des propriétés telles que des mots-clés, des commandes de script à utiliser avec NPM, un nom, etc.

Une de ces propriétés est les dépendances qui sont installées dans votre dépôt local. NPM ajoutera le nom et la version de ces dépendances dans `package.json`, ainsi que `package-lock.json`, un autre fichier utilisé comme solution de repli par NPM au cas où `package.json` échouerait.

Tapez ce qui suit dans votre terminal :

```
$ npm i dayjs
```

Le drapeau `i` est un raccourci pour l'argument `install`. Si vous êtes connecté à Internet, NPM recherchera un module nommé `dayjs` dans le dépôt distant de Node.js, téléchargera le module et l'installera localement. NPM va également ajouter cette dépendance à vos fichiers `package.json` et `package-lock.json`. Maintenant vous pouvez voir qu'il y a un dossier appelé `node_modules`, qui contient le module installé avec d'autres modules s'ils sont nécessaires. Le répertoire `node_modules` contient le code réel qui va être utilisé lorsque la bibliothèque est importée et appelée. Cependant, ce répertoire n'est pas sauvegardé dans les systèmes de gestion de version utilisant Git, puisque le fichier `package.json` fournit toutes les dépendances utilisées. Un autre utilisateur peut prendre le

fichier `package.json` et simplement exécuter `npm install` sur sa propre machine, où NPM créera un dossier `node_modules` avec toutes les dépendances du fichier `package.json`, évitant ainsi le contrôle de version pour les milliers de fichiers disponibles sur le dépôt NPM.

Maintenant que le module `dayjs` est installé dans le répertoire local, ouvrez la console Node.js et tapez les lignes suivantes :

```
const dayjs = require('dayjs');
dayjs().format('YYYY MM-DDTHH:mm:ss')
```

Le module `dayjs` est chargé avec le mot-clé `require`. Lorsqu'une méthode du module est appelée, la bibliothèque prend la date du système actuel et l'affiche dans le format spécifié :

```
2020 11-22T11:04:36
```

Il s'agit du même mécanisme que celui utilisé dans l'exemple précédent, où le moteur d'exécution Node.js charge la fonction tierce dans le code.

Fonctionnalité du serveur

Étant donné que Node.js contrôle l'arrière-plan des applications Web, l'une de ses tâches principales consiste à traiter les requêtes HTTP.

Voici un résumé de la manière dont les serveurs web traitaient les requêtes HTTP entrantes. La fonctionnalité du serveur consiste à écouter les demandes, à déterminer aussi rapidement que possible la réponse dont chacune a besoin et à renvoyer cette réponse à l'expéditeur de la demande. Cette application doit recevoir une requête HTTP entrante déclenchée par l'utilisateur, analyser la requête, effectuer le calcul, générer la réponse et la renvoyer. Un module HTTP tel que Node.js est utilisé car il simplifie ces étapes, permettant au programmeur web de se concentrer sur l'application elle-même.

Considérons l'exemple suivant qui met en œuvre cette fonctionnalité très basique :

```
const http = require('http');
const url = require('url');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
```

```
const queryObject = url.parse(req.url, true).query;
let result = parseInt(queryObject.a) + parseInt(queryObject.b);

res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end(`Result: ${result}\n`);
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Sauvegardez ce contenu dans un fichier appelé `basic_server.js` et exécutez-le avec une commande `node`. Le terminal qui exécute Node.js affichera le message suivant :

```
Server running at http://127.0.0.1:3000/
```

Ensuite, visitez l'URL suivante dans votre navigateur web :

`http://127.0.0.1:3000/numbers?a=2&b=17`

Node.js fait tourner un serveur web dans votre ordinateur, et utilise deux modules : `http` et `url`. Le module `http` met en place un serveur HTTP basique, traite les requêtes web entrantes, et les transmet à notre code d'application simple. Le module `URL` analyse les arguments passés dans l'URL, les convertit au format entier et effectue l'opération d'addition. Le module `http` envoie ensuite la réponse sous forme de texte au navigateur web.

Dans une application web réelle, Node.js est généralement utilisé pour traiter et récupérer des données, souvent à partir d'une base de données, et renvoyer les informations traitées au front-end pour affichage. Mais l'application de base de cette leçon montre de manière concise comment Node.js utilise des modules pour traiter les requêtes web en tant que serveur web.

Exercices guidés

1. Quelles sont les raisons d'utiliser des modules au lieu d'écrire des fonctions simples ?

2. Pourquoi l'environnement Node.js est-il devenu si populaire ? Citez une caractéristique.

3. A quoi sert le fichier `package.json` ?

4. Pourquoi n'est-il pas recommandé d'enregistrer et de partager le dossier `node_modules` ?

Exercices d'exploration

1. Comment pouvez-vous exécuter des applications Node.js sur votre ordinateur ?

2. Comment délimiter les paramètres dans l'URL à analyser dans le serveur ?

3. Indiquez un scénario dans lequel une tâche spécifique pourrait constituer un goulot d'étranglement pour une application Node.js.

4. Comment implémenter un paramètre pour multiplier ou additionner les deux nombres dans l'exemple du serveur ?

Résumé

Cette leçon donne un aperçu de l'environnement Node.js, de ses caractéristiques et de la façon dont il peut être utilisé pour mettre en œuvre des programmes simples. Cette leçon comprend les concepts suivants :

- Ce qu'est Node.js, et pourquoi il est utilisé.
- Comment exécuter des programmes Node.js en utilisant la ligne de commande.
- Les boucles d'événements et le single thread.
- Modules.
- Node Package Manager (NPM).
- Fonctionnalité du serveur.

Réponses aux exercices guidés

1. Quelles sont les raisons d'utiliser des modules au lieu d'écrire des fonctions simples ?

En optant pour les modules au lieu des fonctions classiques, le programmeur crée une base de code plus simple à lire et à maintenir et pour laquelle il peut écrire des tests automatisés.

2. Pourquoi l'environnement Node.js est-il devenu si populaire ? Citez une caractéristique.

L'une des raisons est la flexibilité du langage JavaScript, qui était déjà largement utilisé dans le front-end des applications web. Node.js permet l'utilisation d'un seul langage de programmation dans l'ensemble du système.

3. A quoi sert le fichier `package.json` ?

Ce fichier contient les métadonnées du projet, comme le nom, la version, les dépendances (bibliothèques), etc. Avec un fichier `package.json`, d'autres personnes peuvent télécharger et installer les mêmes bibliothèques et exécuter des tests de la même manière que le créateur original.

4. Pourquoi n'est-il pas recommandé d'enregistrer et de partager le dossier `node_modules` ?

Le dossier `node_modules` contient les implémentations des bibliothèques disponibles dans des dépôts distants. La meilleure façon de partager ces bibliothèques est donc de les indiquer dans le fichier `package.json` et d'utiliser ensuite NPM pour télécharger ces bibliothèques. Cette méthode est plus simple et plus exempte d'erreurs, car vous ne devez pas suivre et maintenir les bibliothèques localement.

Réponses aux exercices d'exploration

1. Comment pouvez-vous exécuter des applications Node.js sur votre ordinateur ?

Vous pouvez les exécuter en tapant `node PATH/FILE_NAME.js` à la ligne de commande dans votre terminal, en changeant `PATH` par le chemin de votre fichier Node.js et en changeant `FILE_NAME.js` par le nom de votre fichier choisi.

2. Comment délimiter les paramètres dans l'URL à analyser dans le serveur ?

Le caractère `&` est utilisé pour délimiter ces paramètres, afin qu'ils puissent être extraits et analysés dans le code JavaScript.

3. Indiquez un scénario dans lequel une tâche spécifique pourrait constituer un goulot d'étranglement pour une application Node.js.

Node.js n'est pas un bon environnement pour exécuter des processus intensifs en termes de CPU car il utilise un seul thread. Un scénario de calcul numérique pourrait ralentir et bloquer l'ensemble de l'application. Si une simulation numérique est nécessaire, il est préférable d'utiliser d'autres outils.

4. Comment implémenter un paramètre pour multiplier ou additionner les deux nombres dans l'exemple du serveur ?

Utilisez un opérateur ternaire ou une condition `if-else` pour vérifier la présence d'un paramètre supplémentaire. Si le paramètre est la chaîne `mult`, retournez le produit des nombres, sinon retournez la somme. Remplacez l'ancien code par l'extrait ci-dessous. Redémarrez le serveur dans la ligne de commande en appuyant sur `Ctrl + C` et en ré-exécutant la commande pour redémarrer le serveur. Testez maintenant la nouvelle application en visitant l'URL `http://127.0.0.1:3000/numbers?a=2&b=17&operation=mult` dans votre navigateur. Si vous omettez ou modifiez le dernier paramètre, les résultats devraient être la somme des chiffres.

```
let result = queryObject.operation == 'mult' ? parseInt(queryObject.a) *  
parseInt(queryObject.b) : parseInt(queryObject.a) + parseInt(queryObject.b);
```




Linux
Professional
Institute

035.2 Bases de NodeJS Express

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 035.2

Valeur

4

Domaines de connaissance les plus importants

- Définir des routes vers des fichiers statiques et des modèles EJS
- Servir des fichiers statiques via Express
- Servir des modèles EJS via Express
- Créer des modèles EJS simples, non imbriqués
- Utiliser l'objet de requête pour accéder aux paramètres HTTP GET et POST et traiter les données soumises par des formulaires HTML
- Sensibilisation à la validation des entrées utilisateur
- Sensibilisation aux scripts intersites (XSS)
- Sensibilisation à la falsification des requêtes intersites (CSRF)

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- Module `express` et `body-parser`
- Objet Express `app`
- `app.get()`, `app.post()`
- `res.query()`, `res.body()`
- Module `ejs`

- `res.render()`
- `<% ... %>`, `<%= ... %>`, `<%# ... %>`, `<%- ... %>`
- `views/`



035.2 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	035 Programmation de serveur NodeJS
Objectif :	035.2 Bases de NodeJS Express
Leçon:	1 sur 2

Introduction

Express.js, ou simplement Express, est un framework populaire qui fonctionne sur Node.js et qui est utilisé pour écrire des serveurs HTTP qui traitent les demandes des clients d'applications web. Express prend en charge de nombreuses façons de lire les paramètres envoyés par HTTP.

Script initial pour le serveur

Pour démontrer les fonctionnalités de base d'Express pour la réception et le traitement des requêtes, simulons une application qui demande certaines informations au serveur. En particulier, le serveur d'exemple :

- Fournit une fonction `echo`, qui retourne simplement le message envoyé par le client.
- Indique au client son adresse IP sur demande.
- Utilise des cookies pour identifier les clients connus.

La première étape est de créer le fichier JavaScript qui servira de serveur. En utilisant `npm`, créez un répertoire appelé `myserver` avec le fichier JavaScript :

```
$ mkdir myserver
$ cd myserver/
$ npm init
```

Pour le point d'entrée, n'importe quel nom de fichier peut être utilisé. Ici, nous allons utiliser le nom de fichier par défaut : `index.js`. Le listing suivant montre un fichier `index.js` basique qui sera utilisé comme point d'entrée pour notre serveur :

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.get('/', (req, res) => {
  res.send('Request received')
})

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Quelques constantes importantes pour la configuration du serveur sont définies dans les premières lignes du script. Les deux premières, `express` et `app`, correspondent au module `express` inclus et à une instance de ce module qui exécute notre application. Nous allons ajouter les actions à effectuer par le serveur à l'objet `app`.

Les deux autres constantes, `host` et `port`, définissent l'hôte et le port de communication associés au serveur.

Si vous avez un hôte accessible publiquement, utilisez son nom au lieu de `myserver` comme valeur de `host`. Si vous ne fournissez pas le nom de l'hôte, Express choisira par défaut `localhost`, l'ordinateur sur lequel l'application s'exécute. Dans ce cas, aucun client extérieur ne pourra accéder au programme, ce qui peut convenir pour les tests mais n'a que peu d'intérêt en production.

Le port doit être fourni, sinon le serveur ne démarrera pas.

Ce script n'attache que deux procédures à l'objet `app` : l'action `app.get()` qui répond aux requêtes faites par les clients via HTTP GET, et l'appel `app.listen()`, qui est nécessaire pour activer le serveur et lui assigne un hôte et un port.

Pour démarrer le serveur, il suffit de lancer la commande `node`, en fournissant le nom du script

comme argument :

```
$ node index.js
```

Dès que le message `Server ready at http://myserver:8080` apparaît, le serveur est prêt à recevoir des requêtes d'un client HTTP. Les demandes peuvent être faites à partir d'un navigateur situé sur le même ordinateur que celui où tourne le serveur, ou à partir d'une autre machine qui peut accéder au serveur.

Tous les détails de la transaction que nous allons voir ici sont affichés dans le navigateur si vous ouvrez une fenêtre pour la console du développeur. Alternativement, la commande `curl` peut être utilisée pour la communication HTTP et vous permet d'inspecter les détails de la connexion plus facilement. Si vous n'êtes pas familier avec la ligne de commande du shell, vous pouvez créer un formulaire HTML pour soumettre des requêtes à un serveur.

L'exemple suivant montre comment utiliser la commande `curl` sur la ligne de commande pour faire une requête HTTP vers le serveur nouvellement déployé :

```
$ curl http://myserver:8080 -v
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> GET / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
> Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/html; charset=utf-8
< Content-Length: 16
< ETag: W/"10-1WVvDtVyAF0vX9evlsFlfiJTT5c"
< Date: Fri, 02 Jul 2021 14:35:11 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Request received
```

L'option `-v` de la commande `curl` affiche tous les en-têtes des requêtes et des réponses, ainsi que d'autres informations de débogage. Les lignes commençant par `>` indiquent les en-têtes des requêtes

envoyées par le client et les lignes commençant par `<` indiquent les en-têtes des réponses envoyées par le serveur. Les lignes commençant par `*` sont des informations générées par `curl` lui-même. Le contenu de la réponse est affiché seulement à la fin, qui dans ce cas est la ligne `Request received`.

L'URL du service, qui dans ce cas contient le nom d'hôte et le port du serveur (`http://myserver:8080`), a été donnée comme argument à la commande `curl`. Parce qu'aucun répertoire ou nom de fichier n'est donné, ceux-ci sont par défaut dans le répertoire racine `/`. La barre oblique apparaît comme le fichier de requête dans la ligne `> GET / HTTP/1.1`, qui est suivie dans la sortie par le nom d'hôte et le port.

En plus d'afficher les en-têtes de connexion HTTP, la commande `curl` aide au développement d'applications en vous permettant d'envoyer des données au serveur en utilisant différentes méthodes HTTP et dans différents formats. Cette flexibilité facilite le débogage des problèmes et l'implémentation de nouvelles fonctionnalités sur le serveur.

Routes

Les requêtes que le client peut faire au serveur dépendent des *routes* qui ont été définies dans le fichier `index.js`. Une route spécifie une méthode HTTP et définit un *chemin* (plus précisément, une URI) qui peut être demandé par le client.

Jusqu'à présent, le serveur n'a qu'une seule route configurée :

```
app.get('/', (req, res) => {  
  res.send('Request received')  
})
```

Même s'il s'agit d'une route très simple, qui renvoie simplement un message en texte clair au client, elle suffit à identifier les composants les plus importants qui sont utilisés pour structurer la plupart des routes :

- La méthode HTTP servie par la route. Dans l'exemple, la méthode HTTP GET est indiquée par la propriété `get` de l'objet `app`.
- Le chemin servi par la route. Lorsque le client ne spécifie pas de chemin pour la requête, le serveur utilise le répertoire racine, qui est le répertoire de base mis de côté pour être utilisé par le serveur web. Un exemple ultérieur dans ce chapitre utilise le chemin `/echo`, qui correspond à une demande faite à `myserver:8080/echo`.
- La fonction exécutée lorsque le serveur reçoit une requête sur cette route, généralement écrite sous forme abrégée comme une *fonction flèche* parce que la syntaxe `=>` pointe vers la définition de la fonction sans nom. Le paramètre `req` (abréviation de "request") et le paramètre `res`

(abréviation de “response”) donnent des détails sur la connexion, transmis à la fonction par l’instance de l’application elle-même.

Méthode POST

Pour étendre les fonctionnalités de notre serveur de test, voyons comment définir une route pour la méthode HTTP POST. Cette méthode est utilisée par les clients lorsqu’ils ont besoin d’envoyer des données supplémentaires au serveur, en plus de celles incluses dans l’en-tête de la requête. L’option `--data` de la commande `curl` invoque automatiquement la méthode POST, et inclut le contenu qui sera envoyé au serveur via POST. La ligne `POST / HTTP/1.1` dans la sortie suivante montre que la méthode POST a été utilisée. Cependant, notre serveur n’a défini qu’une méthode GET, donc une erreur se produit lorsque nous utilisons `curl` pour envoyer une requête via POST :

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"
* Trying 192.168.1.225:8080...
* TCP_NODELAY set
* Connected to myserver (192.168.1.225) port 8080 (#0)
> POST / HTTP/1.1
> Host: myserver:8080
> User-Agent: curl/7.68.0
> Accept: /
> Content-Length: 37
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 37 out of 37 bytes
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< X-Powered-By: Express
< Content-Security-Policy: default-src 'none'
< X-Content-Type-Options: nosniff
< Content-Type: text/html; charset=utf-8
< Content-Length: 140
< Date: Sat, 03 Jul 2021 02:22:45 GMT
< Connection: keep-alive
<
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot POST /</pre>
```

```
</body>
</html>
* Connection #0 to host myserver left intact
```

Dans l'exemple précédent, exécuter `curl` avec le paramètre `--data message="This is the POST request body"` équivaut à soumettre un formulaire contenant le champ texte nommé `message`, rempli avec `This is the POST request body`.

Comme le serveur est configuré avec une seule route pour le chemin `/`, et que cette route ne répond qu'à la méthode HTTP `GET`, l'en-tête de réponse contient la ligne `HTTP/1.1 404 Not Found`. De plus, Express a automatiquement généré une réponse HTML courte avec l'avertissement `Cannot POST`.

Après avoir vu comment générer une requête `POST` à travers `curl`, écrivons un programme Express qui peut traiter avec succès la requête.

Tout d'abord, notez que le champ `Content-Type` de l'en-tête de la requête indique que les données envoyées par le client sont au format `application/x-www-form-urlencoded`. Express ne reconnaît pas ce format par défaut, nous devons donc utiliser le module `express.urlencoded`. Lorsque nous incluons ce module, l'objet `req`—passé comme paramètre au gestionnaire de fonctions—a la propriété `req.body.message` définie, qui correspond au champ `message` envoyé par le client. Le module est chargé avec `app.use`, qui doit être placé avant la déclaration des routes :

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.use(express.urlencoded({ extended: true })))
```

Une fois ceci est fait, il suffirait de changer `app.get` en `app.post` dans la route existante pour répondre aux requêtes faites via `POST` et récupérer le corps de la requête :

```
app.post('/', (req, res) => {
  res.send(req.body.message)
})
```

Au lieu de remplacer la route, une autre possibilité serait de simplement ajouter cette nouvelle route, car Express identifie la méthode HTTP dans l'en-tête de la requête et utilise la route appropriée. Parce que nous sommes intéressés par l'ajout de plus d'une fonctionnalité à ce serveur, il est pratique de séparer chacune d'elles avec son propre chemin, comme `/echo` et `/ip`.

Chemin et gestionnaire de fonctions

Après avoir défini la méthode HTTP qui répondra à la demande, nous devons maintenant définir un chemin spécifique pour la ressource et une fonction qui traite et génère une réponse au client.

Pour étendre la fonctionnalité echo du serveur, nous pouvons définir une route en utilisant la méthode POST avec le chemin /echo :

```
app.post('/echo', (req, res) => {  
  res.send(req.body.message)  
})
```

Le paramètre req du gestionnaire de fonctions contient tous les détails de la demande stockés sous forme de propriétés. Le contenu du champ message du corps de la requête est disponible dans la propriété req.body.message. Dans l'exemple, ce champ est simplement renvoyé au client par l'appel res.send(req.body.message).

N'oubliez pas que les modifications que vous effectuez ne prennent effet qu'après le redémarrage du serveur. Comme vous exécutez le serveur depuis une fenêtre de terminal pendant les exemples de ce chapitre, vous pouvez arrêter le serveur en appuyant sur **Ctrl** + **C** sur ce terminal. Puis relancez le serveur par la commande `node index.js`. La réponse obtenue par le client à la requête `curl` que nous avons montrée auparavant est maintenant réussie :

```
$ curl http://myserver:8080/echo --data message="This is the POST request body"  
This is the POST request body
```

Autres moyens de transmettre et de renvoyer des informations dans une requête GET

Il peut être excessif d'utiliser la méthode HTTP POST si seuls des messages texte courts comme celui utilisé dans l'exemple sont envoyés. Dans ce cas, les données peuvent être envoyées dans une *chaîne de requête* qui commence par un point d'interrogation. Ainsi, la chaîne `?message=C'est+le+message` pourrait être incluse dans le chemin de requête de la méthode HTTP GET. Les champs utilisés dans la chaîne de requête sont disponibles pour le serveur dans la propriété req.query. Par conséquent, un champ nommé message est disponible dans la propriété req.query.message.

Une autre façon d'envoyer des données via la méthode HTTP GET est d'utiliser les *paramètres de route* d'Express :

```
app.get('/echo/:message', (req, res) => {  
  res.send(req.params.message)  
})
```

```
})
```

La route dans cet exemple correspond aux requêtes faites avec la méthode GET en utilisant le chemin `/echo/:message`, où `:message` est un caractère générique pour tout terme envoyé avec cette étiquette par le client. Ces paramètres sont accessibles dans la propriété `req.params`. Avec cette nouvelle route, la fonction `echo` du serveur peut être demandée plus succinctement par le client :

```
$ curl http://myserver:8080/echo/hello
hello
```

Dans d'autres situations, les informations dont le serveur a besoin pour traiter la requête ne doivent pas être explicitement fournies par le client. Par exemple, le serveur a un autre moyen de récupérer l'adresse IP publique du client. Cette information est présente par défaut dans l'objet `req`, dans la propriété `req.ip` :

```
app.get('/ip', (req, res) => {
  res.send(req.ip)
})
```

Maintenant le client peut demander le chemin `/ip` avec la méthode GET pour trouver sa propre adresse IP publique :

```
$ curl http://myserver:8080/ip
187.34.178.12
```

D'autres propriétés de l'objet `req` peuvent être modifiées par le client, notamment les en-têtes de requête disponibles dans `req.headers`. La propriété `req.headers.user-agent`, par exemple, identifie le programme qui effectue la requête. Bien que ce ne soit pas une pratique courante, le client peut modifier le contenu de ce champ, et le serveur ne doit donc pas l'utiliser pour identifier de manière fiable un client particulier. Il est encore plus important de valider les données explicitement fournies par le client, pour éviter les incohérences de limites et de formats qui pourraient nuire à l'application.

Ajustements de la réponse

Comme nous l'avons vu dans les exemples précédents, le paramètre `res` est responsable du retour de la réponse au client. De plus, l'objet `res` peut modifier d'autres aspects de la réponse. Vous avez peut-être remarqué que, bien que les réponses que nous avons implémentées jusqu'à présent ne

soient que de brefs messages en texte brut, l'en-tête `Content-Type` des réponses utilise `text/html` ; `charset=utf-8`. Bien que cela n'empêche pas la réponse en texte brut d'être acceptée, elle sera plus correcte si nous redéfinissons ce champ dans l'en-tête de la réponse en `text/plain` avec le paramètre `res.type('text/plain')`.

D'autres types d'ajustements de réponse impliquent l'utilisation de *cookies*, qui permettent au serveur d'identifier un client qui a déjà fait une demande. Les cookies sont importants pour les fonctionnalités avancées, comme la création de sessions privées qui associent les demandes à un utilisateur spécifique, mais nous allons ici nous contenter d'examiner un exemple simple d'utilisation d'un cookie pour identifier un client qui a déjà accédé au serveur.

Étant donné la conception modulaire d'Express, la gestion des cookies doit être installée avec la commande `npm` avant d'être utilisée dans le script :

```
$ npm install cookie-parser
```

Après l'installation, la gestion des cookies doit être incluse dans le script du serveur. La définition suivante doit être incluse vers le début du fichier :

```
const cookieParser = require('cookie-parser')
app.use(cookieParser())
```

Pour illustrer l'utilisation des cookies, modifions la fonction de gestion de la route avec le chemin racine `/` qui existe déjà dans le script. L'objet `req` possède une propriété `req.cookies`, où sont conservés les cookies envoyés dans l'en-tête de la requête. L'objet `res`, quant à lui, possède une méthode `res.cookie()` qui crée un nouveau cookie à envoyer au client. Le gestionnaire de fonctions, dans l'exemple suivant, vérifie si un cookie avec le nom `known` existe dans la requête. Si un tel cookie n'existe pas, le serveur suppose qu'il s'agit d'un premier visiteur et lui envoie un cookie de ce nom par l'intermédiaire de l'appel `res.cookie('known', '1')`. Nous attribuons arbitrairement la valeur `1` au cookie car il est censé avoir un contenu, mais le serveur ne consulte pas cette valeur. Cette application suppose simplement que la simple présence du cookie indique que le client a déjà demandé cette route auparavant :

```
app.get('/', (req, res) => {
  res.type('text/plain')
  if ( req.cookies.known === undefined ){
    res.cookie('known', '1')
    res.send('Welcome, new visitor!')
  }
  else
```

```
    res.send('Welcome back, visitor');  
  })
```

Par défaut, curl n'utilise pas de cookies dans les transactions. Mais il a des options pour stocker (-c cookies.txt) et envoyer les cookies stockés (-b cookies.txt) :

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -v  
* Trying 192.168.1.225:8080...  
* TCP_NODELAY set  
* Connected to myserver (192.168.1.225) port 8080 (#0)  
> GET / HTTP/1.1  
> Host: myserver:8080  
> User-Agent: curl/7.68.0  
>Accept: /  
>  
* Mark bundle as not supporting multiuse  
< HTTP/1.1 200 OK  
< X-Powered-By: Express  
< Content-Type: text/plain; charset=utf-8  
* Added cookie known="1" for domain myserver, path /, expire 0  
< Set-Cookie: known=1; Path=/  
< Content-Length: 21  
< ETag: W/"15-l7qrxqcicL4xv6EfA5fZFWCFrgY"  
< Date: Sat, 03 Jul 2021 23:45:03 GMT  
< Connection: keep-alive  
<  
* Connection #0 to host myserver left intact  
Welcome, new visitor!
```

Comme cette commande était le premier accès depuis l'implémentation des cookies sur le serveur, le client n'avait pas de cookies à inclure dans la requête. Comme prévu, le serveur n'a pas identifié le cookie dans la requête et a donc inclus le cookie dans les en-têtes de réponse, comme indiqué dans la ligne Set-Cookie : known=1 ; Path=/ de la sortie. Puisque nous avons activé les cookies dans curl, une nouvelle requête inclura le cookie known=1 dans les en-têtes de la requête, permettant au serveur d'identifier la présence du cookie :

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -v  
* Trying 192.168.1.225:8080...  
* TCP_NODELAY set  
* Connected to myserver (192.168.1.225) port 8080 (#0)  
> GET / HTTP/1.1  
> Host: myserver:8080
```

```
> User-Agent: curl/7.68.0
> Accept: /
> Cookie: known=1
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/plain; charset=utf-8
< Content-Length: 21
< ETag: W/"15-ATq2fLQYtLMYIUpJwwpb5SjV9Ww"
< Date: Sat, 03 Jul 2021 23:45:47 GMT
< Connection: keep-alive
<
* Connection #0 to host myserver left intact
Welcome back, visitor
```

Sécurité des cookies

Le développeur doit être conscient des vulnérabilités potentielles lors de l'utilisation de cookies pour identifier les clients effectuant des requêtes. Les attaquants peuvent utiliser des techniques telles que le *cross-site scripting* (XSS) et le *cross-site request forgery* (CSRF) pour voler les cookies d'un client et ainsi se faire passer pour lui lorsqu'il effectue une demande au serveur. En général, ces types d'attaques utilisent des champs de commentaires non validés ou des URL méticuleusement construites pour insérer du code JavaScript malveillant dans la page. Lorsqu'il est exécuté par un client authentique, ce code peut copier des cookies valides et les stocker ou les transmettre à une autre destination.

C'est pourquoi, en particulier dans les applications professionnelles, il est important d'installer et d'utiliser des fonctionnalités Express plus spécialisées, connues sous le nom de *middleware*. Les modules `express-session` ou `cookie-session` permettent un contrôle plus complet et plus sûr de la gestion des sessions et des cookies. Ces composants permettent des contrôles supplémentaires pour empêcher que les cookies soient détournés de leur émetteur d'origine.

Exercices guidés

1. Comment le contenu du champ `comment`, envoyé dans une chaîne de requête de la méthode HTTP `GET`, peut-il être lu dans un gestionnaire de fonctions ?

2. Écrivez une route qui utilise la méthode HTTP GET et le chemin `/agent` pour renvoyer au client le contenu de l'en-tête `user-agent`.

3. Express.js possède une fonctionnalité appelée *paramètres de route*, où un chemin tel que `/user/:name` peut être utilisé pour recevoir le paramètre `name` envoyé par le client. Comment accéder au paramètre `name` dans la fonction de gestion de la route ?

Exercices d'exploration

1. Si le nom d'hôte d'un serveur est `myserver`, quelle route d'Express recevrait la soumission dans le formulaire ci-dessous ?

```
<form action="/contact/feedback" method="post"> ... </form>
```

2. Pendant le développement du serveur, le programmeur ne parvient pas à lire la propriété `req.body`, même après avoir vérifié que le client envoie correctement le contenu via la méthode HTTP POST. Quelle est la cause probable de ce problème ?

3. Que se passe-t-il lorsque le serveur a une route définie sur le chemin `/user/:name` et que le client fait une requête sur `/user/` ?

Résumé

Cette leçon explique comment écrire des scripts Express pour recevoir et traiter des requêtes HTTP. Express utilise le concept de *routes* pour définir les ressources disponibles pour les clients, ce qui vous donne une grande flexibilité pour construire des serveurs pour tout type d'application web. Cette leçon aborde les concepts et procédures suivants :

- Les routes qui utilisent les méthodes HTTP `GET` et HTTP `POST`.
- Comment les données du formulaire sont stockées dans l'objet `request`.
- Comment utiliser les paramètres de route.
- Personnalisation des en-têtes de réponse.
- Gestion basique des cookies.

Réponses aux exercices guidés

1. Comment le contenu du champ `comment`, envoyé dans une chaîne de requête de la méthode HTTP `GET`, peut-il être lu dans un gestionnaire de fonctions ?

Le champ `comment` est disponible dans la propriété `req.query.comment`.

2. Écrivez une route qui utilise la méthode HTTP `GET` et le chemin `/agent` pour renvoyer au client le contenu de l'en-tête `user-agent`.

```
app.get('/agent', (req, res) => {  
  res.send(req.headers.user-agent)  
})
```

3. Express.js possède une fonctionnalité appelée *paramètres de route*, où un chemin tel que `/user/:name` peut être utilisé pour recevoir le paramètre `name` envoyé par le client. Comment accéder au paramètre `name` dans la fonction de gestion de la route ?

Le paramètre `name` est accessible dans la propriété `req.params.name`.

Réponses aux exercices d'exploration

1. Si le nom d'hôte d'un serveur est `myserver`, quelle route d'Express recevrait la soumission dans le formulaire ci-dessous ?

```
<form action="/contact/feedback" method="post"> ... </form>
```

```
app.post('/contact/feedback', (req, res) => {  
  ...  
})
```

2. Pendant le développement du serveur, le programmeur ne parvient pas à lire la propriété `req.body`, même après avoir vérifié que le client envoie correctement le contenu via la méthode HTTP `POST`. Quelle est la cause probable de ce problème ?

Le programmeur n'a pas inclus le module `express.urlencoded`, qui permet à Express d'extraire le corps d'une requête.

3. Que se passe-t-il lorsque le serveur a une route définie sur le chemin `/user/:name` et que le client fait une requête sur `/user/` ?

Le serveur émettra une réponse `404 Not Found`, car la route nécessite que le paramètre `:name` soit fourni par le client.



035.2 Leçon 2

Certification :	Web Development Essentials
Version :	1.0
Thème :	035 Programmation de serveur NodeJS
Objectif :	035.2 Bases de NodeJS Express
Leçon:	2 sur 2

Introduction

Les serveurs web disposent de mécanismes très polyvalents pour produire des réponses aux demandes des clients. Pour certaines requêtes, il suffit au serveur web de fournir une réponse statique, non traitée, car la ressource demandée est la même pour tous les clients. Par exemple, lorsqu'un client demande une image qui est accessible à tous, il suffit que le serveur envoie le fichier contenant l'image.

Mais lorsque les réponses sont générées dynamiquement, elles peuvent avoir besoin d'être mieux structurées que de simples lignes écrites dans le script du serveur. Dans ce cas, il est pratique pour le serveur web de pouvoir générer un document complet, qui peut être interprété et rendu par le client. Dans le contexte du développement d'applications web, les documents HTML sont généralement créés sous forme de modèles et conservés séparément du script du serveur, qui insère les données dynamiques à des endroits prédéterminés dans le modèle approprié, puis envoie la réponse formatée au client.

Les applications web consomment souvent des ressources statiques et dynamiques. Un document HTML, même s'il a été généré dynamiquement, peut contenir des références à des ressources statiques telles que des fichiers CSS et des images. Pour démontrer comment Express peut aider à

gérer ce type de demande, nous allons d'abord configurer un exemple de serveur qui fournit des fichiers statiques, puis mettre en œuvre des routes qui génèrent des réponses structurées, basées sur des modèles.

Fichiers statiques

La première étape consiste à créer le fichier JavaScript qui sera exécuté en tant que serveur. Suivons le même schéma que celui abordé dans les leçons précédentes pour créer une application Express simple : créez d'abord un répertoire appelé `server`, puis installez les composants de base avec la commande `npm` :

```
$ mkdir server
$ cd server/
$ npm init
$ npm install express
```

Pour le point d'entrée, n'importe quel nom de fichier peut être utilisé, mais ici nous allons utiliser le nom de fichier par défaut : `index.js`. Le listing suivant montre un fichier `index.js` basique qui sera utilisé comme point de départ pour notre serveur :

```
const express = require('express')
const app = express()
const host = "myserver"
const port = 8080

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})
```

Vous n'avez pas besoin d'écrire du code explicite pour envoyer un fichier statique. Express dispose d'un logiciel médiateur *middleware* à cet effet, appelé `express.static`. Si votre serveur doit envoyer des fichiers statiques au client, chargez simplement le middleware `express.static` au début du script :

```
app.use(express.static('public'))
```

Le paramètre `public` indique le répertoire qui stocke les fichiers que le client peut demander. Les chemins demandés par les clients ne doivent pas inclure le répertoire `public`, mais seulement le nom du fichier, ou le chemin d'accès au fichier par rapport au répertoire `public`. Pour demander le fichier

`public/layout.css`, par exemple, le client fait une demande à `/layout.css`.

Sortie formatée

Si l'envoi de contenu statique est simple, le contenu généré dynamiquement peut varier considérablement. La création de réponses dynamiques avec des messages courts permet de tester facilement les applications dans leurs premiers stades de développement. Par exemple, voici une route de test qui renvoie simplement au client un message envoyé par la méthode HTTP POST. La réponse peut juste reproduire le contenu du message en texte brut, sans aucun formatage :

```
app.post('/echo', (req, res) => {  
  res.send(req.body.message)  
})
```

Une route comme celle-ci est un bon exemple à utiliser lors de l'apprentissage d'Express et à des fins de diagnostic, où une réponse brute envoyée avec `res.send()` est suffisante. Mais un serveur utile doit être capable de produire des réponses plus complexes. Nous allons maintenant passer au développement de ce type de route plus sophistiqué.

Notre nouvelle application, au lieu de se contenter de renvoyer le contenu de la requête en cours, conserve une liste complète des messages envoyés dans les requêtes précédentes par chaque client et renvoie la liste de chaque client lorsqu'il en fait la demande. Une réponse fusionnant tous les messages est une option, mais d'autres modes de sortie formatés sont plus appropriés, surtout lorsque les réponses deviennent plus élaborées.

Pour recevoir et stocker les messages du client envoyés pendant la session en cours, nous devons d'abord inclure des modules supplémentaires pour gérer les cookies et les données envoyées via la méthode HTTP POST. L'exemple de serveur suivant a pour seul but d'enregistrer les messages envoyés via POST et d'afficher les messages précédemment envoyés lorsque le client émet une requête GET. Il y a donc deux routes pour le chemin `/`. La première route répond aux demandes faites avec la méthode HTTP POST et la seconde répond aux demandes faites avec la méthode HTTP GET :

```
const express = require('express')  
const app = express()  
const host = "myserver"  
const port = 8080  
  
app.use(express.static('public'))  
  
const cookieParser = require('cookie-parser')  
app.use(cookieParser())
```

```
const { v4: uuidv4 } = require('uuid')

app.use(express.urlencoded({ extended: true }))

// Array to store messages
let messages = []

app.post('/', (req, res) => {

  // Only JSON enabled requests
  if ( req.headers.accept !== "application/json" )
  {
    res.sendStatus(404)
    return
  }

  // Locate cookie in the request
  let uuid = req.cookies.uuid

  // If there is no uuid cookie, create a new one
  if ( uuid === undefined )
    uuid = uuidv4()

  // Add message first in the messages array
  messages.unshift({uuid: uuid, message: req.body.message})

  // Collect all previous messages for uuid
  let user_entries = []
  messages.forEach( (entry) => {
    if ( entry.uuid == req.cookies.uuid )
      user_entries.push(entry.message)
  })

  // Update cookie expiration date
  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  // Send back JSON response
  res.json(user_entries)

})

app.get('/', (req, res) => {
```

```

// Only JSON enabled requests
if ( req.headers.accept !== "application/json" )
{
  res.sendStatus(404)
  return
}

// Locate cookie in the request
let uuid = req.cookies.uuid

// Client's own messages
let user_entries = []

// If there is no uuid cookie, create a new one
if ( uuid === undefined ){
  uuid = uuidv4()
}
else {
  // Collect messages for uuid
  messages.forEach( (entry) => {
    if ( entry.uuid == req.cookies.uuid )
      user_entries.push(entry.message)
  })
}

// Update cookie expiration date
let expires = new Date(Date.now());
expires.setDate(expires.getDate() + 30);
res.cookie('uuid', uuid, { expires: expires })

// Send back JSON response
res.json(user_entries)
})

app.listen(port, host, () => {
  console.log(`Server ready at http://${host}:${port}`)
})

```

Nous avons gardé la configuration des fichiers statiques en haut, car il sera bientôt utile de fournir des fichiers statiques tels que `layout.css`. En plus du middleware `cookie-parser` introduit dans le chapitre précédent, l'exemple inclut également le middleware `uuid` pour générer un numéro d'identification unique passé comme cookie à chaque client qui envoie un message. S'ils ne sont pas

déjà installés dans le répertoire du serveur d'exemple, ces modules peuvent être installés avec la commande `npm install cookie-parser uuid`.

Le tableau global appelé `messages` stocke les messages envoyés par tous les clients. Chaque élément de ce tableau consiste en un objet avec les propriétés `uuid` et `message`.

Ce qui est vraiment nouveau dans ce script est la méthode `res.json()`, utilisée à la fin des deux routes pour générer une réponse au format JSON avec le tableau contenant les messages déjà envoyés par le client :

```
// Send back JSON response
res.json(user_entries)
```

JSON est un format de texte brut qui vous permet de regrouper un ensemble de données dans une structure unique qui est associative : c'est-à-dire que le contenu est exprimé sous forme de clés et de valeurs. JSON est particulièrement utile lorsque les réponses sont destinées à être traitées par le client. Grâce à ce format, un objet ou un tableau JavaScript peut être facilement reconstruit du côté client avec toutes les propriétés et tous les index de l'objet original sur le serveur.

Comme nous structurons chaque message en JSON, nous refusons les requêtes qui ne contiennent pas `application/json` dans leur en-tête `accept` :

```
// Only JSON enabled requests
if ( req.headers.accept !== "application/json" )
{
  res.sendStatus(404)
  return
}
```

Une requête faite avec une commande `curl` simple pour insérer un nouveau message ne sera pas acceptée, car `curl` par défaut ne spécifie pas `application/json` dans l'en-tête `accept` :

```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt
Not Found
```

L'option `-H "accept : application/json"` modifie l'en-tête de la requête pour spécifier le format de la réponse, qui cette fois sera acceptée et répondra dans le format spécifié :


```
$ curl http://myserver:8080/ --data message="My first message" -c cookies.txt -b
cookies.txt -H "accept: application/json"
["My first message"]
```

La récupération des messages en utilisant l'autre route se fait de manière similaire, mais cette fois en utilisant la méthode HTTP GET :

```
$ curl http://myserver:8080/ -c cookies.txt -b cookies.txt -H "accept:
application/json"
["Another message", "My first message"]
```

Modèles

Les réponses dans des formats tels que JSON sont pratiques pour communiquer entre programmes, mais l'objectif principal de la plupart des serveurs d'applications web est de produire du contenu HTML pour la consultation humaine. Intégrer du code HTML dans du code JavaScript n'est pas une bonne idée, car mélanger les langages dans un même fichier rend le programme plus sensible aux erreurs et nuit à la maintenance du code.

Express peut fonctionner avec différents *moteurs de modèles* qui séparent le HTML pour le contenu dynamique ; la liste complète se trouve sur le [site des moteurs de modèles Express](#). L'un des moteurs de modèles les plus populaires est *Embedded JavaScript* (EJS), qui vous permet de créer des fichiers HTML avec des balises spécifiques pour l'insertion de contenu dynamique.

Comme les autres composants d'Express, EJS doit être installé dans le répertoire où s'exécute le serveur :

```
$ npm install ejs
```

Ensuite, le moteur EJS doit être défini comme le moteur de rendu par défaut dans le script du serveur (vers le début du fichier `index.js`, avant les définitions de route) :

```
app.set('view engine', 'ejs')
```

La réponse générée avec le modèle est envoyée au client avec la fonction `res.render()`, qui reçoit comme paramètres le nom du fichier du modèle et un objet contenant les valeurs qui seront accessibles depuis le modèle. Les routes utilisées dans l'exemple précédent peuvent être réécrites pour générer des réponses HTML ainsi que JSON :

```
app.post('/', (req, res) => {

  let uuid = req.cookies.uuid

  if ( uuid === undefined )
    uuid = uuidv4()

  messages.unshift({uuid: uuid, message: req.body.message})

  let user_entries = []
  messages.forEach( (entry) => {
    if ( entry.uuid == req.cookies.uuid )
      user_entries.push(entry.message)
  })

  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
  res.cookie('uuid', uuid, { expires: expires })

  if ( req.headers.accept == "application/json" )
    res.json(user_entries)
  else
    res.render('index', {title: "My messages", messages: user_entries})

})

app.get('/', (req, res) => {

  let uuid = req.cookies.uuid

  let user_entries = []

  if ( uuid === undefined ){
    uuid = uuidv4()
  }
  else {
    messages.forEach( (entry) => {
      if ( entry.uuid == req.cookies.uuid )
        user_entries.push(entry.message)
    })
  }

  let expires = new Date(Date.now());
  expires.setDate(expires.getDate() + 30);
```

```
res.cookie('uuid', uuid, { expires: expires })

if ( req.headers.accept == "application/json" )
  res.json(user_entries)
else
  res.render('index', {title: "My messages", messages: user_entries})

})
```

Notez que le format de la réponse dépend de l'en-tête `accept` trouvé dans la requête :

```
if ( req.headers.accept == "application/json" )
  res.json(user_entries)
else
  res.render('index', {title: "My messages", messages: user_entries})
```

Une réponse au format JSON est envoyée uniquement si le client le demande explicitement. Sinon, la réponse est générée à partir du modèle `index`. Le même tableau `user_entries` alimente à la fois la sortie JSON et le modèle, mais l'objet utilisé comme paramètre pour ce dernier a aussi la propriété `title: "My messages"`, qui sera utilisée comme titre dans le modèle.

HTML Templates

Comme les fichiers statiques, les fichiers contenant les modèles HTML résident dans leur propre répertoire. Par défaut, EJS suppose que les fichiers de modèles se trouvent dans le répertoire `views/`. Dans l'exemple, un modèle nommé `index` a été utilisé, donc EJS recherche le fichier `views/index.ejs`. Le listing suivant est le contenu d'un modèle simple `views/index.ejs` qui peut être utilisé avec le code de l'exemple :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title><%= title %></title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/layout.css">
</head>
<body>

<div id="interface">
```

```
<form action="/" method="post">
  <p>
    <input type="text" name="message">
    <input type="submit" value="Submit">
  </p>
</form>

<ul>
  <% messages.forEach( (message) => { %>
  <li><%= message %></li>
  <% }) %>
</ul>

</div>

</body>
</html>
```

La première balise spéciale EJS est l'élément `<title>` dans la section `<head>` :

```
<%= title %>
```

Pendant le processus de rendu, cette balise spéciale sera remplacée par la valeur de la propriété `title` de l'objet passé en paramètre à la fonction `res.render()`.

La majeure partie du modèle est constituée de code HTML classique. Le modèle contient donc le formulaire HTML permettant d'envoyer de nouveaux messages. Le serveur de test répond aux méthodes HTTP GET et POST pour le même chemin `/`, d'où les attributs `action="/"` et `method="post"` dans la balise de formulaire.

D'autres parties du modèle sont un mélange de code HTML et de balises EJS. EJS dispose de balises à des fins spécifiques dans le modèle :

`<% ... %>`

Insère un contrôle de flux. Aucun contenu n'est directement inséré par cette balise, mais elle peut être utilisée avec des structures JavaScript pour choisir, répéter ou supprimer des sections du code HTML. Exemple de démarrage d'une boucle : `<% messages.forEach((message) => { %>`

`<%# ... %>`

Définit un commentaire, dont le contenu est ignoré par l'analyseur syntaxique. Contrairement aux commentaires écrits en HTML, ces commentaires ne sont pas visibles pour le client.

`<%= ... %>`

Insère le contenu échappatoire de la variable. Il est important d'échapper au contenu inconnu pour éviter l'exécution de code JavaScript, qui peut ouvrir des brèches pour les attaques de type cross-site scripting (XSS). Exemple : `<%= title %>`

`<%- ... %>`

Insère le contenu de la variable sans échappement.

Le mélange de code HTML et de balises EJS est évident dans l'extrait où les messages du client sont rendus sous forme de liste HTML :

```
<ul>
<% messages.forEach( (message) => { %>
<li><%= message %></li>
<% }) %>
</ul>
```

Dans cet extrait, la première balise `<% ... %>` lance une instruction `forEach` qui parcourt en boucle tous les éléments du tableau `message`. Les délimiteurs `<%` et `%>` vous permettent de contrôler les extraits de HTML. Un nouvel élément de liste HTML, `<%= message %>`, sera produit pour chaque élément de `messages`. Avec ces changements, le serveur enverra la réponse en HTML lorsqu'une requête comme la suivante sera reçue :

```
$ curl http://myserver:8080/ --data message="This time" -c cookies.txt -b
cookies.txt
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My messages</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/layout.css">
</head>
<body>

<div id="interface">

<form action="/" method="post">
<p>
  <input type="text" name="message">
  <input type="submit" value="Submit">
```

```
</p>
</form>

<ul>

<li>This time</li>

<li>in HTML</li>

</ul>

</div>

</body>
</html>
```

La séparation entre le code de traitement des requêtes et le code de présentation de la réponse rend le code plus propre et permet à une équipe de répartir le développement de l'application entre des personnes ayant des spécialités distinctes. Un concepteur web, par exemple, peut se concentrer sur les fichiers de modèle dans `views/` et les feuilles de style associées, qui sont fournis en tant que fichiers statiques stockés dans le répertoire `public/` sur le serveur d'exemple.

Exercices guidés

1. Comment configurer `express.static` pour que les clients puissent demander des fichiers dans le répertoire `assets` ?

2. Comment le type de réponse, qui est spécifié dans l'en-tête de la demande, peut-il être identifié dans une route d'Express ?

3. Quelle méthode du paramètre de route `res` (response) génère une réponse au format JSON à partir d'un tableau JavaScript appelé `content` ?

Exercices d'exploration

1. Par défaut, les fichiers de modèles d'Express se trouvent dans le répertoire `views`. Comment modifier ce paramètre pour que les fichiers de modèles soient stockés dans `templates` ?

2. Supposons qu'un client reçoive une réponse HTML sans titre (c'est-à-dire `<title></title>`). Après avoir vérifié le modèle EJS, le développeur trouve la balise `<title><% title %></title>` dans la section `head` du fichier. Quelle est la cause probable du problème ?

3. Utilisez les balises de modèle EJS pour écrire une balise HTML `<h2></h2>` avec le contenu de la variable JavaScript `h2`. Cette balise ne doit être rendue que si la variable `h2` n'est pas vide.

Résumé

Cette leçon couvre les méthodes de base fournies par Express.js pour générer des réponses statiques et formatées mais dynamiques. Peu d'efforts sont nécessaires pour configurer un serveur HTTP pour les fichiers statiques et le système de modèles EJS offre un moyen facile de générer du contenu dynamique à partir de fichiers HTML. Cette leçon aborde les concepts et procédures suivants :

- Utilisation de `express.static` pour les réponses aux fichiers statiques.
- Comment créer une réponse qui correspond au champ de type de contenu dans l'en-tête de la requête.
- Réponses structurées en JSON.
- Utilisation des balises EJS dans les modèles HTML.

Réponses aux exercices guidés

1. Comment configurer `express.static` pour que les clients puissent demander des fichiers dans le répertoire `assets` ?

Un appel à `app.use(express.static('assets'))` doit être ajouté au script du serveur.

2. Comment le type de réponse, qui est spécifié dans l'en-tête de la demande, peut-il être identifié dans une route d'Express ?

Le client définit les types acceptables dans le champ d'en-tête `accept`, qui est mis en correspondance avec la propriété `req.headers.accept`.

3. Quelle méthode du paramètre de route `res` (response) génère une réponse au format JSON à partir d'un tableau JavaScript appelé `content` ?

La méthode `res.json() : res.json(content)`.

Réponses aux exercices d'exploration

1. Par défaut, les fichiers de modèles d'Express se trouvent dans le répertoire `views`. Comment modifier ce paramètre pour que les fichiers de modèles soient stockés dans `templates` ?

Le répertoire peut être défini dans les paramètres initiaux du script avec `app.set('views', './templates')`.

2. Supposons qu'un client reçoive une réponse HTML sans titre (c'est-à-dire `<title></title>`). Après avoir vérifié le modèle EJS, le développeur trouve la balise `<title><% title %></title>` dans la section `head` du fichier. Quelle est la cause probable du problème ?

La balise `<%= %>` doit être utilisée pour entourer le contenu d'une variable, comme dans `<%= title %>`.

3. Utilisez les balises de modèle EJS pour écrire une balise HTML `<h2></h2>` avec le contenu de la variable JavaScript `h2`. Cette balise ne doit être rendue que si la variable `h2` n'est pas vide.

```
<% if ( h2 != "" ) { %>
<h2><%= h2 %></h2>
<% } %>
```



035.3 Bases de SQL

Référence aux objectifs de LPI

Web Development Essentials version 1.0, Exam 030, Objective 035.3

Valeur

3

Domaines de connaissance les plus importants

- Établir une connexion à une base de données depuis NodeJS
- Récupérer des données de la base de données dans NodeJS
- Exécuter des requêtes SQL à partir de NodeJS
- Créer des requêtes SQL simples excluant les jointures
- Comprendre les clés primaires
- Les variables d'échappement utilisées dans les requêtes SQL
- Connaître les injections SQL

Liste partielle de termes, fichiers et utilitaires utilisés pour cet objectif

- Module NPM `sqlite3`
- `Database.run()`, `Database.close()`, `Database.all()`, `Database.get()`, `Database.each()`
- `CREATE TABLE`
- `INSERT`, `SELECT`, `DELETE`, `UPDATE`



035.3 Leçon 1

Certification :	Web Development Essentials
Version :	1.0
Thème :	035 Programmation de serveur NodeJS
Objectif :	035.3 Bases de SQL
Leçon :	1 sur 1

Introduction

Bien que vous puissiez écrire vos propres fonctions pour mettre en œuvre le stockage persistant, il peut être plus pratique d'utiliser un système de gestion de base de données pour accélérer le développement et garantir une meilleure sécurité et stabilité pour les données sous forme de tables. La stratégie la plus populaire pour stocker des données organisées dans des tables interdépendantes, en particulier lorsque ces tables sont fortement interrogées et mises à jour, consiste à installer une base de données relationnelle qui prend en charge le langage SQL (*Structured Query Language*), un langage orienté vers les bases de données relationnelles. Node.js prend en charge divers systèmes de gestion de bases de données SQL. Conformément aux principes de portabilité et d'exécution en espace utilisateur adoptés par Node.js Express, SQLite est un choix approprié pour le stockage persistant des données utilisées par ce type de serveur HTTP.

SQL

Le langage de requête structuré est spécifique aux bases de données. Les opérations d'écriture et de lecture sont exprimées dans des phrases appelées *déclarations* et *requêtes*. Les déclarations et les requêtes sont composées de *clauses*, qui définissent les conditions d'exécution de l'opération.

Les noms et les adresses électroniques, par exemple, peuvent être stockés dans une table de base de données qui contient les champs `name` et `email`. Une base de données peut contenir plusieurs tables, donc chaque table doit avoir un nom unique. Si nous utilisons le nom `contacts` pour la table des noms et des adresses électroniques, un nouvel enregistrement peut être inséré avec l'instruction suivante :

```
INSERT INTO contacts (name, email) VALUES ("Carol", "carol@example.com");
```

Cette instruction d'insertion est composée de la clause `INSERT INTO`, qui définit la table et les champs où les données seront insérées. La deuxième clause, `VALUES`, définit les valeurs qui seront insérées. Il n'est pas nécessaire de mettre la majuscule aux clauses, mais c'est une pratique courante, afin de mieux reconnaître les mots-clés SQL dans une instruction ou une requête.

Une requête sur la table des contacts est effectuée de manière similaire, mais en utilisant la clause `SELECT` :

```
SELECT email FROM contacts;  
dave@example.com  
carol@example.com
```

Dans ce cas, la clause `SELECT email` sélectionne un champ parmi les entrées de la table `contacts`. La clause `WHERE` restreint la requête à des lignes spécifiques :

```
SELECT email FROM contacts WHERE name = "Dave";  
dave@example.com
```

SQL possède de nombreuses autres clauses, et nous en examinerons certaines dans les sections suivantes. Mais d'abord il faut voir comment intégrer la base de données SQL avec Node.js.

SQLite

SQLite est probablement la solution la plus simple pour incorporer des fonctionnalités de base de données SQL dans une application. Contrairement à d'autres systèmes de gestion de base de données populaires, SQLite n'est pas un serveur de base de données auquel un client se connecte. Au lieu de cela, SQLite fournit un ensemble de fonctions qui permettent au développeur de créer une base de données comme un fichier classique. Dans le cas d'un serveur HTTP mis en œuvre avec Node.js Express, ce fichier est généralement situé dans le même répertoire que le script du serveur.

Avant d'utiliser SQLite dans Node.js, vous devez installer le module `sqlite3`. Exécutez la commande

suivante dans le répertoire d'installation du serveur, c'est-à-dire le répertoire contenant le script Node.js que vous allez exécuter.

```
$ npm install sqlite3
```

Sachez qu'il existe plusieurs modules qui supportent SQLite, comme `better-sqlite3`, dont l'utilisation est subtilement différente de `sqlite3`. Les exemples de cette leçon sont pour le module `sqlite3`, donc ils pourraient ne pas fonctionner comme prévu si vous choisissez un autre module.

Ouverture de la base de données

Pour démontrer comment un serveur Node.js Express peut fonctionner avec une base de données SQL, écrivons un script qui stocke et affiche les messages envoyés par un client identifié par un cookie. Les messages sont envoyés par le client via la méthode HTTP POST et la réponse du serveur peut être formatée en JSON ou en HTML (à partir d'un modèle), selon le format demandé par le client. Cette leçon n'entrera pas dans le détail de l'utilisation des méthodes HTTP, des cookies et des modèles. Les extraits de code présentés ici supposent que vous disposez déjà d'un serveur Node.js Express où ces fonctionnalités sont configurées et disponibles.

La façon la plus simple de stocker les messages envoyés par le client est de les stocker dans un tableau global, où chaque message précédemment envoyé est associé à une clé d'identification unique pour chaque client. Cette clé peut être envoyée au client sous la forme d'un cookie, qui est présenté au serveur lors de futures demandes de récupération de ses messages précédents.

Cependant, cette approche présente une faiblesse : comme les messages sont stockés uniquement dans un tableau global, tous les messages seront perdus lorsque la session actuelle du serveur sera terminée. C'est l'un des avantages de travailler avec des bases de données, car les données sont stockées de manière persistante et ne sont pas perdues si le serveur est redémarré.

En utilisant le fichier `index.js` comme script principal du serveur, nous pouvons incorporer le module `sqlite3` et indiquer le fichier qui sert de base de données, comme suit :

```
const sqlite3 = require('sqlite3')
const db = new sqlite3.Database('messages.sqlite3');
```

S'il n'existe pas déjà, le fichier `messages.sqlite3` sera créé dans le même répertoire que le fichier `index.js`. Dans ce fichier unique, toutes les structures et les données respectives seront stockées. Toutes les opérations de base de données effectuées dans le script seront relayées par la constante `db`, qui est le nom donné au nouvel objet `sqlite3` qui ouvre le fichier `messages.sqlite3`.

Structure d'une table

Aucune donnée ne peut être insérée dans la base de données tant qu'au moins une table n'a pas été créée. Les tables sont créées avec l'instruction `CREATE TABLE` :

```
db.run('CREATE TABLE IF NOT EXISTS messages (id INTEGER PRIMARY KEY AUTOINCREMENT,
      uuid CHAR(36), message TEXT)')
```

La méthode `db.run()` est utilisée pour exécuter des instructions SQL dans la base de données. L'instruction elle-même est écrite en tant que paramètre de la méthode. Bien que les instructions SQL doivent se terminer par un point-virgule lorsqu'elles sont saisies dans un processeur de ligne de commande, le point-virgule est facultatif dans les instructions passées comme paramètres dans un programme.

Comme la méthode `run` sera exécutée à chaque fois que le script sera exécuté avec `node index.js`, l'instruction SQL inclut la clause conditionnelle `IF NOT EXISTS` pour éviter les erreurs lors des futures exécutions lorsque la table `messages` existe déjà.

Les champs qui composent la table `messages` sont `id`, `uuid`, et `message`. Le champ `id` est un entier unique utilisé pour identifier chaque entrée de la table, il est donc créé comme `PRIMARY KEY`. Les clés primaires ne peuvent pas être nulles et il ne peut pas y avoir deux clés primaires identiques dans la même table. Par conséquent, presque toutes les tables SQL ont une clé primaire afin de suivre le contenu de la table. Bien qu'il soit possible de choisir explicitement la valeur de la clé primaire d'un nouvel enregistrement (à condition qu'il n'existe pas encore dans la table), il est pratique que la clé soit générée automatiquement. L'indicateur `AUTOINCREMENT` du champ `id` est utilisé à cet effet.

NOTE

La définition explicite des clés primaires dans SQLite est facultative, car SQLite crée lui-même une clé primaire automatiquement. Comme indiqué dans la documentation de SQLite : “En SQLite, les lignes d'une table ont normalement un entier signé de 64 bits ROWID qui est unique parmi toutes les lignes de la même table. Si une table contient une colonne de type `INTEGER PRIMARY KEY`, alors cette colonne devient un alias pour le ROWID. Vous pouvez alors accéder au ROWID en utilisant l'un des quatre noms différents, les trois noms originaux décrits ci-dessus, ou le nom donné à la colonne `INTEGER PRIMARY KEY`. Tous ces noms sont des alias les uns pour les autres et fonctionnent aussi bien dans n'importe quel contexte.”

Les champs `uuid` et `message` stockent respectivement l'identification du client et le contenu du message. Un champ de type `CHAR(36)` stocke un nombre fixe de 36 caractères, et un champ de type `TEXT` stocke des textes de longueur arbitraire.

Entrée des données

La fonction principale de notre exemple de serveur est de stocker les messages qui sont liés au client qui les a envoyés. Le client envoie le message dans le champ `message` du corps de la requête envoyée avec la méthode HTTP POST. L'identification du client se trouve dans un cookie appelé `uuid`. Avec ces informations, nous pouvons écrire la route Express pour insérer de nouveaux messages dans la base de données :

```
app.post('/', (req, res) => {

  let uuid = req.cookies.uuid

  if ( uuid === undefined )
    uuid = uuidv4()

  // Insert new message into the database
  db.run('INSERT INTO messages (uuid, message) VALUES (?, ?)', uuid, req.body
    .message)

  // If an error occurs, err object contains the error message.
  db.all('SELECT id, message FROM messages WHERE uuid = ?', uuid, (err, rows) => {

    let expires = new Date(Date.now());
    expires.setDate(expires.getDate() + 30);
    res.cookie('uuid', uuid, { expires: expires })

    if ( req.headers.accept == "application/json" )
      res.json(rows)
    else
      res.render('index', {title: "My messages", rows: rows})

  })

})
```

Cette fois, la méthode `db.run()` exécute une déclaration d'insertion, mais notez que le `uuid` et le `req.body.message` ne sont pas écrits directement dans la ligne de déclaration. Au lieu de cela, des points d'interrogation ont été substitués aux valeurs. Chaque point d'interrogation correspond à un paramètre qui suit l'instruction SQL dans la méthode `db.run()`.

L'utilisation de points d'interrogation comme caractères de remplissage dans l'instruction qui est exécutée dans la base de données permet à SQLite de distinguer plus facilement les éléments statiques de l'instruction et ses données variables. Cette stratégie permet à SQLite d'échapper ou

d'assainir le contenu des variables qui font partie de l'instruction, empêchant ainsi une faille de sécurité courante appelée *injection SQL*. Dans cette attaque, les utilisateurs malveillants insèrent des instructions SQL dans les données variables dans l'espoir que les instructions seront exécutées par inadvertance ; la désinfection déjoue l'attaque en désactivant les caractères dangereux dans les données.

Requêtes

Comme le montre l'exemple de code, notre intention est d'utiliser la même route pour insérer de nouveaux messages dans la base de données et pour générer la liste des messages précédemment envoyés. La méthode `db.all()` renvoie la collection de toutes les entrées de la table qui correspondent aux critères définis dans la requête.

Contrairement aux instructions effectuées par `db.run()`, `db.all()` génère une liste d'enregistrements qui sont traités par la fonction flèche désignée dans le dernier paramètre :

```
(err, rows) => {}
```

Cette fonction, à son tour, prend deux paramètres : `err` et `rows`. Le paramètre `err` sera utilisé si une erreur se produit et empêche l'exécution de la requête. En cas de succès, tous les enregistrements sont disponibles dans le tableau `rows`, où chaque élément est un objet correspondant à un seul enregistrement de la table. Les propriétés de cet objet correspondent aux noms des champs indiqués dans la requête : `uuid` et `message`.

Le tableau `rows` est une structure de données JavaScript. En tant que tel, il peut être utilisé pour générer des réponses avec des méthodes fournies par Express, telles que `res.json()` et `res.render()`. Lorsqu'elle est rendue à l'intérieur d'un modèle EJS, une boucle conventionnelle peut lister tous les enregistrements :

```
<ul>
<% rows.forEach( (row) => { %>
<li><strong><%= row.id %></strong>: <%= row.message %></li>
<% }) %>
</ul>
```

Au lieu de remplir le tableau `rows` avec tous les enregistrements retournés par la requête, dans certains cas, il peut être plus pratique de traiter chaque enregistrement individuellement avec la méthode `db.each()`. La syntaxe de la méthode `db.each()` est similaire à celle de la méthode `db.all()`, mais le paramètre `row` dans `(err, row) => {}` correspond à un seul enregistrement à la fois.

Modification du contenu de la base de données

Jusqu'à présent, notre client ne peut qu'ajouter et interroger des messages sur le serveur. Puisque le client connaît maintenant le `id` des messages précédemment envoyés, nous pouvons implémenter une fonction pour modifier un enregistrement spécifique. Le message modifié peut également être envoyé à une route de méthode HTTP POST, mais cette fois avec un paramètre de route pour attraper le `id` donné par le client dans le chemin de la requête :

```
app.post('/:id', (req, res) => {
  let uuid = req.cookies.uuid

  if ( uuid === undefined ){
    uuid = uuidv4()
    // 401 Unauthorized
    res.sendStatus(401)
  }
  else {

    // Update the stored message
    // using named parameters
    let param = {
      $message: req.body.message,
      $id: req.params.id,
      $uuid: uuid
    }

    db.run('UPDATE messages SET message = $message WHERE id = $id AND uuid = $uuid', param, function(err){

      if ( this.changes > 0 )
      {
        // A 204 (No Content) status code means the action has
        // been enacted and no further information is to be supplied.
        res.sendStatus(204)
      }
      else
        res.sendStatus(404)

    })
  }
})
```

Ce parcours montre comment utiliser les clauses UPDATE et WHERE pour modifier un enregistrement

existant. Une différence importante avec les exemples précédents est l'utilisation de paramètres nommés, où les valeurs sont regroupées dans un seul objet (`param`) et passées à la méthode `db.run()` au lieu de spécifier chaque valeur par elle-même. Dans ce cas, les noms des champs (précédés de `$`) sont les propriétés de l'objet. Les paramètres nommés permettent d'utiliser les noms de champs (précédés de `$`) comme caractères de remplissage au lieu de points d'interrogation.

Une instruction comme celle de l'exemple ne provoquera aucune modification de la base de données si la condition imposée par la clause `WHERE` ne correspond pas à un enregistrement de la table. Pour évaluer si des enregistrements ont été modifiés par l'instruction, une fonction de rappel peut être utilisée comme dernier paramètre de la méthode `db.run()`. Dans la fonction, le nombre d'enregistrements modifiés peut être demandé à partir de `this.changes`. Notez que les fonctions flèches ne peuvent pas être utilisées dans ce cas, car seules les fonctions régulières de la forme `function(){}` définissent l'objet `this`.

La suppression d'un enregistrement est très similaire à sa modification. Nous pouvons, par exemple, continuer à utiliser le paramètre de route `:id` pour identifier le message à supprimer, mais cette fois dans une route invoquée par la méthode HTTP DELETE du client :

```
app.delete('/:id', (req, res) => {
  let uuid = req.cookies.uuid

  if ( uuid === undefined ){
    uuid = uuidv4()
    res.sendStatus(401)
  }
  else {
    // Named parameters
    let param = {
      $id: req.params.id,
      $uuid: uuid
    }

    db.run('DELETE FROM messages WHERE id = $id AND uuid = $uuid', param, function
(err){
      if ( this.changes > 0 )
        res.sendStatus(204)
      else
        res.sendStatus(404)
    })
  }
})
```

Les enregistrements sont supprimés d'une table avec la clause `DELETE FROM`. Nous avons à nouveau utilisé la fonction de rappel pour évaluer combien d'entrées ont été supprimées de la table.

Fermeture de la base de données

Une fois défini, l'objet `db` peut être référencé à tout moment pendant l'exécution du script, car le fichier de la base de données reste ouvert pendant toute la session en cours. Il n'est pas courant de fermer la base de données pendant l'exécution du script.

Une fonction de fermeture de la base de données est toutefois utile pour éviter de fermer brusquement la base de données lorsque le processus du serveur se termine. Bien que peu probable, l'arrêt brutal de la base de données peut entraîner des incohérences si les données en mémoire ne sont pas encore enregistrées dans le fichier. Par exemple, un arrêt brutal de la base de données avec perte de données peut se produire si le script est terminé par l'utilisateur en appuyant sur le raccourci clavier `Ctrl + C`.

Dans le scénario `Ctrl + C` que nous venons de décrire, la méthode `process.on()` peut intercepter les signaux envoyés par le système d'exploitation et exécuter un arrêt ordonné de la base de données et du serveur :

```
process.on('SIGINT', () => {  
  db.close()  
  server.close()  
  console.log('HTTP server closed')  
})
```

Le raccourci `Ctrl + C` invoque le signal `SIGINT` du système d'exploitation, qui met fin à un programme de premier plan dans le terminal. Avant de terminer le processus à la réception du signal `SIGINT`, le système invoque la fonction de rappel (le dernier paramètre de la méthode `process.on()`). À l'intérieur de la fonction callback, vous pouvez mettre n'importe quel code de nettoyage, en particulier la méthode `db.close()` pour fermer la base de données et `server.close()`, qui ferme gracieusement l'instance Express elle-même.

Exercices guidés

1. Quel est le rôle d'une clé primaire dans une table de base de données SQL ?

2. Quelle est la différence entre les requêtes utilisant `db.all()` et `db.each()` ?

3. Pourquoi est-il important d'utiliser des caractères de remplacement et de ne pas inclure les données envoyées par le client directement dans une instruction ou une requête SQL ?

Exercices d'exploration

1. Quelle méthode du module `sqlite3` peut être utilisée pour renvoyer une seule entrée de table, même si la requête correspond à plusieurs entrées ?

2. Supposons que le tableau `rows` soit passé comme paramètre à une fonction de callback et qu'il contienne le résultat d'une requête effectuée avec `db.all()`. Comment un champ appelé `price`, qui est présent en première position de `rows`, peut-il être référencé dans la fonction de rappel ?

3. La méthode `db.run()` exécute les instructions de modification de la base de données, telles que `INSERT INTO`. Après avoir inséré un nouvel enregistrement dans une table, comment pouvez-vous récupérer la clé primaire de l'enregistrement nouvellement inséré ?

Résumé

Cette leçon couvre l'utilisation de base des bases de données SQL dans les applications Node.js Express. Le module `sqlite3` offre un moyen simple de stocker des données persistantes dans une base de données SQLite, où un seul fichier contient toute la base de données et ne nécessite pas un serveur de base de données spécialisé. Cette leçon aborde les concepts et procédures suivants :

- Comment établir une connexion à une base de données depuis Node.js.
- Comment créer une table simple et le rôle des clés primaires.
- Utilisation de l'instruction SQL `INSERT INTO` pour ajouter de nouvelles données depuis le script.
- Les requêtes SQL utilisant les méthodes SQLite standard et les fonctions de rappel.
- Modifier des données dans la base de données en utilisant les instructions SQL `UPDATE` et `DELETE`.

Réponses aux exercices guidés

1. Quel est le rôle d'une clé primaire dans une table de base de données SQL ?

La clé primaire est le champ d'identification unique de chaque enregistrement dans une table de base de données.

2. Quelle est la différence entre les requêtes utilisant `db.all()` et `db.each()` ?

La méthode `db.all()` invoque la fonction de rappel avec un tableau unique contenant toutes les entrées correspondant à la requête. La méthode `db.each()` invoque la fonction de rappel pour chaque ligne de résultat.

3. Pourquoi est-il important d'utiliser des caractères de remplacement et de ne pas inclure les données envoyées par le client directement dans une instruction ou une requête SQL ?

Avec les caractères de remplacement, les données soumises par l'utilisateur sont protégées avant d'être incluses dans la requête ou l'instruction. Cela permet d'éviter les attaques par injection SQL, qui consistent à placer des instructions SQL dans des données variables afin d'effectuer des opérations arbitraires sur la base de données.

Réponses aux exercices d'exploration

1. Quelle méthode du module `sqlite3` peut être utilisée pour renvoyer une seule entrée de table, même si la requête correspond à plusieurs entrées ?

La méthode `db.get()` a la même syntaxe que `db.all()`, mais ne retourne que la première entrée correspondant à la requête.

2. Supposons que le tableau `rows` soit passé comme paramètre à une fonction de callback et qu'il contienne le résultat d'une requête effectuée avec `db.all()`. Comment un champ appelé `price`, qui est présent en première position de `rows`, peut-il être référencé dans la fonction de rappel ?

Chaque élément de `rows` est un objet dont les propriétés correspondent aux noms des champs de la base de données. Ainsi, la valeur du champ `price` dans le premier résultat est dans `rows[0].price`.

3. La méthode `db.run()` exécute les instructions de modification de la base de données, telles que `INSERT INTO`. Après avoir inséré un nouvel enregistrement dans une table, comment pouvez-vous récupérer la clé primaire de l'enregistrement nouvellement inséré ?

Une fonction régulière de la forme `function(){}` peut être utilisée comme fonction de rappel de la méthode `db.run()`. À l'intérieur de cette fonction, la propriété `this.lastID` contient la valeur de la clé primaire du dernier enregistrement inséré.

Impression

© 2022 par Linux Professional Institute: Supports de cours, “Web Development Essentials (030) (Version 1.0)”.

PDF généré: 2022-08-22

Cette oeuvre est placée sous licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 4.0 International (CC-BY-NC-ND 4.0). Pour consulter une copie de cette licence, visitez

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Bien que le Linux Professional Institute ait fait tout son possible pour s'assurer que les informations et les instructions contenues dans cette publication soient exactes, le Linux Professional Institute décline toute responsabilité en cas d'erreurs ou d'omissions, y compris, mais sans s'y limiter, la responsabilité des dommages résultant de l'utilisation ou de la confiance accordée à cette publication. L'utilisation des informations et des instructions contenues dans cet ouvrage se fait à vos risques et périls. Si les exemples de code ou toute autre technologie décrite ou contenue dans cet ouvrage sont soumis à des licences open source ou aux droits de propriété intellectuelle de tiers, il vous incombe de veiller à ce que leur utilisation soit conforme à ces licences et/ou ces droits.

Les supports de cours du LPI sont une initiative du Linux Professional Institute (<https://lpi.org>). Les supports de cours et leurs traductions sont disponibles à l'adresse <https://learning.lpi.org>.

Pour toute question ou commentaire sur cette édition ou sur l'ensemble du projet, contactez-nous à l'adresse suivante : learning@lpi.org.