

Éric Sarrion

Programmation avec Node.js, Express.js et MongoDB

JavaScript côté serveur

EYROLLES

Eric Sarrion

Formateur et développeur en tant que consultant indépendant, Éric Sarrion participe à toutes sortes de projets informatiques depuis plus de 25 ans. Auteur des deux best-sellers *jQuery & jQuery UI* et *jQuery mobile* aux éditions Eyrolles, il est réputé pour la limpidité de ses explications et de ses exemples.

Un livre incontournable pour développer des applications web professionnelles !

Né à la fin des années 1990, le langage JavaScript connaît aujourd’hui une deuxième vie. Après s’être installé sur pratiquement tous les sites web de la planète, il s’invite désormais sur de plus en plus de serveurs web, notamment grâce à Node.js qui symbolise le JavaScript côté serveur. Node.js est donc plus que jamais une plate-forme de développement très en vogue, utilisée notamment par Flickr, LinkedIn et PayPal.

Des outils de plus en plus populaires

Ce livre propose une présentation pas à pas pour mettre en place une application web avec Node.js, Express.js et MongoDB. Plus que de simples phénomènes de mode, ces outils sont désormais incontournables pour tous les développeurs web. Un livre extrêmement pratique où les aficionados d’Éric Sarrion retrouveront sa minutie dans la description de chaque étape et chausse-trappe.

À qui s’adresse ce livre ?

- Aux étudiants, développeurs et chefs de projet
- À tous les autodidactes férus de programmation qui veulent découvrir Node.js

Au sommaire

Introduction à Node.js. Un premier programme avec Node.js • Utiliser REPL. **Gestion des modules.** Visibilité des variables • Le fichier package.json. **Gestion des événements.** Méthodes utilitaires • Gestion des streams. **Gestion des fichiers. Gestion des processus.** Exécuter un processus grâce à la méthode exec() du module child_process • Exécuter un processus grâce à la méthode spawn() du module child_process • Communication entre le processus père et le processus fils. **Gestion des connexions TCP et UDP. Gestion des connexions HTTP.** Créer un serveur et un client HTTP. **Utiliser les web sockets avec socket.io. Introduction au module Connect.** Utiliser les middlewares définis dans Connect. **Introduction au framework Express.** Installer le framework Express • Créer une application web avec Express • Architecture d’une application Express • Le modèle MVC • Routage des requêtes avec Express • Qu’est-ce qu’une route ? • Architecture REST • Objet app.routes défini par Express • Définir l’URL dans les routes • Organiser l’écriture des routes en créant des modules séparés • Organiser l’écriture des routes en utilisant REST • Bien utiliser les middlewares. **Envoyer la réponse du serveur.** **Objets app, req et res utilisés par Express.** Objet app : gérer l’application Express • Gérer le format des variables dans les URL • Récupérer et modifier le code HTML généré par une vue • Partager des objets avec les vues • Objet req : gérer la requête reçue • Récupérer les informations transmises par les utilisateurs • Récupérer les informations sur la route utilisée • Objet res : gérer la réponse à envoyer • Créer les vues avec EJS • Installer EJS. Une première vue avec EJS • Transmettre des paramètres à la vue • Cas pratique : utiliser plusieurs vues dans une application. **Introduction à MongoDB.** Installer MongoDB • Documents et collections • Utiliser l’exécutable mongo • Établir une connexion à la base de données • Créer des documents • Rechercher des documents • Mettre à jour des documents • Supprimer des documents • Actions globales sur une collection • Actions globales sur une base de données. **Introduction au module Mongoose.** Installer le module Mongoose • Établir une connexion à la base de données avec Mongoose • Utiliser les schémas et les modèles. **Créer des documents.** Rechercher des documents. **Modifier des documents.** Supprimer des documents • Valider les données. **Utiliser le concept de population. Utiliser les middlewares dans Mongoose.** Construction d’une application client serveur • Le module async • Le module supervisor • Le module node-inspector • Le module mongo-express.

Programmation avec Node.js, Express.js et MongoDB

CHEZ LE MÊME ÉDITEUR

DANS LA MÊME COLLECTION



DANS LA COLLECTION BLANCHE



DANS LA COLLECTION DESIGN WEB



AUTRES OUVRAGES



Retrouvez aussi nos livres numériques sur
<http://izibook.eyrolles.com>

Éric Sarrion

Programmation avec Node.js, Express.js et MongoDB

JavaScript côté serveur

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

Avant-propos

Pourquoi un livre sur Node.js ?

Node.js séduit de plus en plus de développeurs web, satisfaits de pouvoir manipuler les couches hautes (réaliser une application web) et les couches basses (manipuler les octets échangés entre un client et un serveur) lors de la construction d'une application. Node.js est très extensible et il existe de nombreux modules, écrits par d'autres développeurs, qui répondront sûrement à vos besoins.

De plus, Node est écrit et s'utilise en JavaScript. Ce langage est initialement réservé aux programmes exécutés par les navigateurs, mais Node en a fait un langage utilisable également côté serveur. Cela permet, avec la connaissance d'un seul langage, d'écrire des programmes à la fois pour le client et pour le serveur.

Enfin, la popularité de Node vient aussi de sa rapidité d'exécution. Cette dernière est liée à son architecture événementielle, qui le rend différent des autres types de serveurs.

Pour toutes ces raisons (et sûrement d'autres encore), Node.js mérite bien d'avoir un livre en français, qui explique comment l'utiliser pour réaliser des applications complètes.

Structure du livre

Ce livre est divisé en quatre parties, qu'il vaut mieux lire dans l'ordre.

- La première partie décrit le cœur de Node.js. Cette partie est essentielle pour comprendre la suite de l'ouvrage.
- La deuxième partie décrit le module Express écrit pour Node, qui permet de construire plus aisément des applications web.
- La troisième partie décrit l'interaction avec une base de données orientée documents, telle que MongoDB. Le module Mongoose, permettant une interface

entre Node et MongoDB, est décrit en détail. Cette partie se termine par la construction d'une application web utilisant l'ensemble des connaissances acquises.

- La quatrième partie se consacre à l'étude de quelques modules utiles de Node.js.

On aurait pu écrire un nouveau livre rien que sur cette quatrième partie, tellement l'écosystème autour de Node est important.

À qui s'adresse ce livre ?

Ce livre s'adresse à tous ceux qui sont curieux de savoir comment fonctionne Node.js ! Donc le public sera varié, pouvant être des développeurs, des chefs de projet, mais également des étudiants en informatique.

Table des matières

PREMIÈRE PARTIE

Le cœur de Node.js 1

CHAPITRE 1

Introduction à Node.js 3

Installation de Node.js	3
Un premier programme avec Node.js	4
Principe de fonctionnement de Node.js	5
Utiliser REPL	6

CHAPITRE 2

Gestion des modules..... 7

Inclusion de modules dans un fichier JavaScript	7
Définir le module dans un fichier JavaScript externe	8
Définir le module par un ensemble de fichiers situés dans un même répertoire ..	9
Utiliser le fichier index.js	10
Utiliser le répertoire node_modules	11
Utiliser un module standard défini par Node	12
Télécharger de nouveaux modules avec npm	16
Écrire un module proposant de nouvelles fonctionnalités	20
Cas particulier : un module composé d'une fonction principale	23
L'objet module défini par Node	25
Mise en cache des modules	27
Visibilité des variables	28
Le fichier package.json	29

CHAPITRE 3

Gestion des événements..... 31

Pourquoi utiliser la classe events.EventEmitter ?	31
Créer un objet de la classe events.EventEmitter	32
Gérer des événements sur un objet de classe events.EventEmitter	33

Utiliser les méthodes addListener() et emit()	33
Utiliser plusieurs fois la méthode addListener()	35
Supprimer un gestionnaire d'événements	37
Supprimer tous les gestionnaires pour un événement	38
Transmettre des paramètres lors de l'événement	39
Créer une classe dérivant de events.EventEmitter	41
Implémentation d'une classe Server simple	41
Créer plusieurs serveurs associés à la classe Server	42
Amélioration du programme	44
Méthodes définies dans la classe events.EventEmitter	45
CHAPITRE 4	
Méthodes utilitaires	47
Gestion de l'affichage à l'écran : objet console	47
Utiliser console.log()	48
Utiliser console.time(label) et console.timeEnd(label)	49
Utiliser console.dir(obj)	50
Fonctions générales : module util	51
Formatage de chaîne de caractères	51
Inspecter des objets avec util.inspect()	54
Héritage de classes	55
Méthodes booléennes	56
Gestion des URL : module url	57
Utiliser url.parse()	58
Utiliser url.resolve()	59
Gestion des requêtes : module querystring	60
Gestion des chemins : module path	61
Gestion d'octets : classe Buffer	63
Créer un objet de classe Buffer	64
Modifier un buffer	65
Copier et découper un buffer	68
Changer l'encodage d'une chaîne au moyen d'un buffer	69
Connaître la taille d'une chaîne de caractères en octets	70
Gestion des timers	71
CHAPITRE 5	
Gestion des streams	73
Créer un stream en lecture	74
Utiliser l'événement data sur le stream	74
Définir la méthode _read() sur le stream	77
Indiquer l'encodage pour les paquets lus sur le stream	80

Utiliser la méthode read() sur le stream plutôt que l'événement data	81
Connaitre la fin du stream en lecture avec l'événement end	83
Exemple de gestion d'un stream en lecture	83
Utiliser les méthodes pause() et resume()	84
Créer un stream en écriture	86
Utiliser la méthode write() sur le stream	86
Définir la méthode _write() sur le stream	87
Indiquer la fin d'un stream en écriture	88
Connecter un stream en lecture sur un stream en écriture	90
Exemple de stream en écriture	91
Créer un stream en lecture et écriture	92
CHAPITRE 6	
Gestion des fichiers.....	95
Gestion synchrone et gestion asynchrone	95
Gestion synchrone	95
Gestion asynchrone	96
Quelle gestion (synchrone ou asynchrone) choisir ?	97
Ouvrir et fermer un fichier	98
Ouvrir un fichier	98
Fermer un fichier	99
Lire un fichier	99
Lecture du fichier en totalité	100
Lecture partielle du fichier	101
Écrire dans un fichier	103
Écaser le contenu du fichier	103
Ajouter des octets à la fin du fichier	104
Dupliquer un fichier	105
Supprimer un fichier	106
Renommer un fichier ou un répertoire	107
Créer ou supprimer un répertoire	108
Créer un répertoire	108
Supprimer un répertoire	109
Lister tous les fichiers d'un répertoire	110
Tester l'existence d'un fichier ou d'un répertoire	111
Obtenir des informations sur un fichier ou un répertoire	111
Relier les fichiers et les streams	112
Copier un fichier dans un autre au moyen d'un stream	113
Copier les caractères entrés au clavier dans un fichier	113

CHAPITRE 7**Gestion des processus 115**

Exécuter un processus grâce à la méthode exec() du module child_process	115
Exécuter une commande système en tant que nouveau processus	116
Exécuter un fichier Node en tant que nouveau processus	117
Exécuter un processus grâce à la méthode spawn() du module child_process ..	118
Exécuter un fichier Node en tant que nouveau processus	119
Gérer les cas d'erreur dans le processus fils	120
Communication entre le processus père et le processus fils	121

CHAPITRE 8**Gestion des connexions TCP 123**

Principe d'une connexion TCP	123
Le programme Telnet	124
Créer un serveur TCP	125
Utiliser la méthode net.createServer()	125
Autre forme de la méthode net.createServer()	127
Utiliser le paramètre socket pour indiquer au client qu'il est connecté	128
Utiliser l'événement listening pour indiquer que le serveur a démarré	128
Méthodes associées à un serveur TCP	130
Événements associés à un serveur TCP	133
Créer un client TCP	135
Utiliser la méthode net.connect(port)	135
Dissocier le programme du client et celui du serveur	136
Gérer la déconnexion d'un client TCP au serveur	138
Communication entre le serveur TCP et un client TCP	139
Gérer le flux entre le serveur et le client	139
Gérer l'arrêt du serveur	141
Exemple : communication entre plusieurs clients TCP via un serveur TCP ..	143

CHAPITRE 9**Gestion des connexions UDP 147**

Principe d'une connexion UDP	147
Le programme Netcat	148
Créer un serveur UDP	149
Utiliser la méthode dgram.createSocket()	149
Autre forme de la méthode dgram.createSocket()	151
Utiliser l'événement listening pour indiquer que le serveur a démarré	151
Connexion/déconnexion du client ou du serveur	152
Créer un client UDP	153

Utiliser la méthode dgram.createSocket()	153
Envoyer un message vers le serveur avec la méthode send()	153
Permettre au serveur de répondre au client	155
Dissocier le programme du client et celui du serveur	156
Événements associés à un objet de classe dgram.Socket	158
Méthodes associées à un objet de classe dgram.Socket	159
Exemple : communication entre deux sockets UDP	160
CHAPITRE 10	
Gestion des connexions HTTP	163
Créer un serveur HTTP	163
Utiliser la méthode http.createServer()	164
Autre forme de la méthode http.createServer()	165
Objet associé au paramètre request	166
Objet associé au paramètre response	169
Envoyer un fichier statique en réponse au navigateur	170
Utiliser plusieurs fichiers statiques dans la réponse	172
Envoyer un en-tête HTTP dans la réponse au navigateur	173
Événement associé à l'objet response	176
Événements associés au serveur	176
Méthodes définies sur l'objet server	177
Créer un client HTTP	177
Utiliser la méthode http.request()	177
Utiliser l'événement response plutôt que la fonction de callback	181
Utiliser l'événement end sur le stream en lecture	182
Envoyer les informations reçues dans un fichier	183
Utiliser la méthode http.get()	184
Transmettre des données vers le serveur HTTP	185
Dissocier le programme du client et celui du serveur	190
Transmettre un fichier vers le serveur (upload de fichier)	192
CHAPITRE 11	
Utiliser les web sockets avec socket.io.....	195
Installer le module socket.io	195
Communication du client vers le serveur	196
Programme côté client	196
Programme côté serveur	197
Exécution du programme	199
Déconnexion d'un client	200
Communication du serveur vers le client	201
Programme côté client	202

Programme côté serveur	203
Exécution du programme	203
Diffuser des informations à plusieurs clients	204
Transmission des informations entre le client et le serveur	206
Associer des données à une socket	207
Utiliser plusieurs programmes de traitement des sockets	209
Programme côté client	210
Programme côté serveur	211

DEUXIÈME PARTIE

Construire des applications web avec le framework Express **213**

CHAPITRE 12

Introduction au module Connect **215**

Installer le module Connect	215
Créer un serveur HTTP sans utiliser Connect	217
Créer un serveur HTTP en utilisant Connect	218
Écrire les callbacks en paramètres de la méthode connect.createServer()	218
Utiliser la méthode app.use()	221
Utiliser l'objet app en tant que fonction de callback	222
Définir et utiliser un middleware	223
Créer le middleware dans le programme de l'application	223
Créer le middleware dans un fichier externe	225
Transmettre des paramètres au middleware	226
Chaînage des méthodes dans Connect	227
Cas d'erreurs dans les middlewares	228

CHAPITRE 13

Utiliser les middlewares définis dans Connect **231**

Middleware logger : afficher les informations dans les logs	232
Middleware errorHandler : afficher les messages d'erreur	235
Middleware static : afficher les fichiers statiques	237
Middleware query : parser les données transmises dans la requête	238
Middleware bodyParser : parser les données transmises dans l'en-tête	239
Middleware favicon : gérer l'icône affichée dans la barre d'adresses	241
Middleware session : gérer les sessions	242
Ajouter des éléments dans la session	243
Supprimer des éléments dans la session	245

Mécanisme de stockage des données dans la session	245
Middleware methodOverride : gérer les requêtes REST	247
Introduction à REST	247
Utiliser le middleware methodOverride	248
Paramétriser le middleware methodOverride	250
 CAPITRE 14	
Introduction au framework Express	251
Installer le framework Express	251
Créer une application web avec Express	252
Architecture d'une application Express	256
Le fichier app.js	256
Autres répertoires et fichiers créés par Express	258
Le modèle MVC	259
 CAPITRE 15	
Routage des requêtes avec Express	261
Qu'est-ce qu'une route ?	261
Analyse des routes déjà présentes dans app.js	261
Ajout d'une nouvelle route dans app.js	263
Utiliser la route en tant que middleware	264
Architecture REST	266
Méthodes liées aux types de requêtes HTTP	266
Utiliser le middleware methodOverride dans Express	267
Utiliser l'outil Postman sous Chrome	268
Définir un middleware valable pour tous les types de routes	270
Objet app.routes défini par Express	271
Afficher les routes utilisées	271
Supprimer une route utilisée	273
Définir l'URL dans les routes	274
URL définie sous forme de chaîne de caractères	274
URL définie sous forme d'expression régulière	274
URL définie en utilisant des noms de variables	278
Utiliser plusieurs noms de variables dans l'URL	279
Noms de variables optionnels	280
Vérifier les valeurs possibles d'un nom de variable	281
Ordre de priorité des routes	281
Indiquer une route qui récupère les cas d'erreur	283
Organiser l'écriture des routes en créant des modules séparés	285
Organiser l'écriture des routes en utilisant REST	288

Activer les routes REST dans le fichier app.js	290
Écrire le traitement associé aux routes	291
Créer des services web en utilisant le composant :format dans les routes	295
Bien utiliser les middlewares	297
Objet app.stack défini par Express	297
Influence du middleware app.router	302
Enchaînement des middlewares avec next()	305
Utiliser la fonction next("route")	312
CHAPITRE 16	
Envoyer la réponse du serveur.....	319
Retourner un code HTTP	319
Les différentes catégories de code HTTP	320
Utiliser la méthode res.status(statusCode)	322
Retourner un en-tête au navigateur	324
Retourner le corps de la réponse	325
Retourner du code HTML encodé en UTF-8	325
Retourner du texte simple	327
Retourner du JSON	328
Retourner des fichiers statiques	329
Retourner des fichiers dynamiques	330
Retourner une vue	331
Permettre un affichage lisible du code HTML de la page	335
Rediriger vers une autre URL	336
CHAPITRE 17	
Objets app, req et res utilisés par Express	339
Objet app : gérer l'application Express	339
Gérer le format des variables dans les URL	340
Récupérer et modifier le code HTML généré par une vue	344
Partager des objets avec les vues	346
Objet req : gérer la requête reçue	348
Récupérer les informations transmises par les utilisateurs	349
Récupérer les informations sur la route utilisée	350
Objet res : gérer la réponse à envoyer	352
CHAPITRE 18	
Créer les vues avec EJS.....	353
Installer EJS	353
Une première vue avec EJS	354

Transmettre des paramètres à la vue	356
Cas pratique : utiliser plusieurs vues dans une application	359
Cinématique de l'application	359
Programmes de l'application	362
Ajouter des styles dans les vues	369

TROISIÈME PARTIE**Utiliser la base de données MongoDB avec Node ...371****CHAPITRE 19****Introduction à MongoDB373**

Installer MongoDB	373
Documents et collections	374
Définir un document	374
Définir une collection	375
Utiliser l'exécutable mongo	376
Exécution dans un shell	376
Exécution dans un fichier JavaScript	377
Établir une connexion à la base de données	378
Créer des documents	379
Rechercher des documents	381
Rechercher tous les documents de la collection	382
Spécifier une condition AND	383
Utiliser \$gt, \$lt ou \$in dans une condition	384
Spécifier une condition OR	386
Utiliser les conditions AND et OR simultanément	387
Rechercher selon l'existence ou le type d'un champ dans un document	388
Rechercher l'existence d'un champ dans un document avec \$exists	388
Rechercher selon un type de champ avec \$type	392
Rechercher à l'aide d'une expression JavaScript avec \$where	394
Rechercher dans des sous-documents	396
Rechercher dans des tableaux	399
Trier les documents lors d'une recherche	402
Indiquer les champs à retourner lors d'une recherche	405
Compter le nombre de documents trouvés lors d'une recherche	407
Rechercher le premier document qui satisfait une recherche	408
Mettre à jour des documents	408
Utiliser la méthode save(document)	409
Utiliser la méthode update(query, update, options)	412
Mettre à jour la totalité d'un document	413

Mettre à jour partiellement un document	414
Mettre à jour plusieurs documents simultanément	416
Supprimer des documents	417
Actions globales sur une collection	420
Actions globales sur une base de données	422
 CHAPITRE 20	
Introduction au module Mongoose	423
Installer le module Mongoose	423
Établir une connexion à la base de données avec Mongoose	424
Utiliser les schémas et les modèles	426
Définir un schéma	427
Définir un modèle	428
 CHAPITRE 21	
Créer des documents.....	431
Créer un document en utilisant la méthode d'instance save()	431
Insérer un document dans la collection	431
Récupérer la liste des documents de la collection	432
Insérer un document, puis récupérer la liste des documents de la collection ..	433
Créer un document en utilisant la méthode de classe create	436
Créer des sous-documents	438
 CHAPITRE 22	
Rechercher des documents	441
Utiliser la méthode find(conditions, callback)	441
Exemples de recherche	442
Écriture des conditions de recherche	445
Utiliser la méthode find(conditions)	446
Méthodes utilisables dans la classe mongoose.Query	452
Utiliser la méthode findOne()	453
Utiliser la méthode findById()	454
Utiliser la méthode count()	454
Utiliser count(conditions, callback)	455
Utiliser count(callback)	457
 CHAPITRE 23	
Modifier des documents	459
Utiliser la méthode de classe update()	459
Mise à jour d'un seul document correspondant aux critères de recherche ..	460

Mise à jour de plusieurs documents correspondant aux critères de recherche	461
Suppression de champs dans les documents	463
Utiliser la méthode save()	464
Utiliser la méthode findOneAndUpdate()	466
Utiliser la méthode findByIdAndUpdate()	467
CHAPITRE 24	
Supprimer des documents.....	469
Utiliser la méthode de classe remove()	469
Utiliser la méthode d'instance remove()	470
Utiliser la méthode findOneAndRemove()	472
Utiliser la méthode findByIdAndRemove()	473
CHAPITRE 25	
Valider les données	475
Préparation de la base de données	475
Valider un premier document	476
Afficher un message d'erreur si la validation échoue	478
Validations par défaut de Mongoose	481
Créer sa propre validation	484
Validations asynchrones	488
CHAPITRE 26	
Utiliser le concept de population.....	493
Indiquer les relations dans les schémas	493
Ajout de documents dans les collections	495
Recherche de documents dans les collections	497
Utiliser la méthode populate()	500
Utiliser la méthode populate() sur l'objet mongoose.Query	501
Utiliser la méthode populate() sur le modèle	502
Utiliser la méthode populate() sur un document	503
CHAPITRE 27	
Utiliser les middlewares dans Mongoose.....	505
Utiliser les pre middlewares	506
Méthode pre() définissant un middleware	506
Utiliser un pre middleware lors de la sauvegarde d'un document	506
Utiliser un pre middleware lors de la validation d'un document	508
Utiliser un pre middleware lors de la suppression d'un document	510
Utiliser le mot-clé this dans un pre middleware	512

Utiliser les post middlewares	513
Méthode post() définissant un middleware	513
Utiliser un post middleware	514

CHAPITRE 28**Construction d'une application client serveur..... 517**

Application construite en utilisant REST	518
Application construite sans utiliser REST	523

QUATRIÈME PARTIE**Quelques modules Node (très) utiles 529****CHAPITRE 29****Le module async** **531**

Installer le module async	531
Méthodes agissant sur les tableaux de données	532
Méthode each()	532
Méthode eachSeries()	536
Méthode map()	537
Méthode mapSeries()	541
Méthode filter()	542
Méthode filterSeries()	544
Méthode reject()	545
Méthode rejectSeries()	546
Méthode detect()	547
Méthode detectSeries()	548
Méthode sortBy()	549
Méthode some()	552
Méthode every()	554
Méthodes agissant sur l'enchaînement des fonctions de callback	556
Méthode series()	556
Méthode parallel()	560
Méthode parallelLimit()	563
Méthode waterfall()	564
Récapitulatif des méthodes du module async	565

CHAPITRE 30**Le module supervisor..... 569**

Installer le module supervisor	569
Utiliser le module supervisor	570

CHAPITRE 31	
Le module node-inspector	573
Installer le module node-inspector	573
Utiliser le module node-inspector	574
CHAPITRE 32	
Le module mongo-express	577
Installer le module mongo-express	577
Utiliser le module mongo-express	578
Index.....	581

PREMIÈRE PARTIE

Le cœur de Node.js

1

Introduction à Node.js

Ce chapitre constitue une introduction à Node.js. Il montre comment l'installer et vérifier que son installation s'est correctement déroulée.

Installation de Node.js

Node.js est un ensemble de fichiers permettant de développer des applications côté serveur en JavaScript.

Vous trouverez sur le site nodejs.org les différents fichiers à télécharger, sous forme d'un programme d'installation pour votre système (Windows, Mac et Linux) ou de fichiers à décompresser.

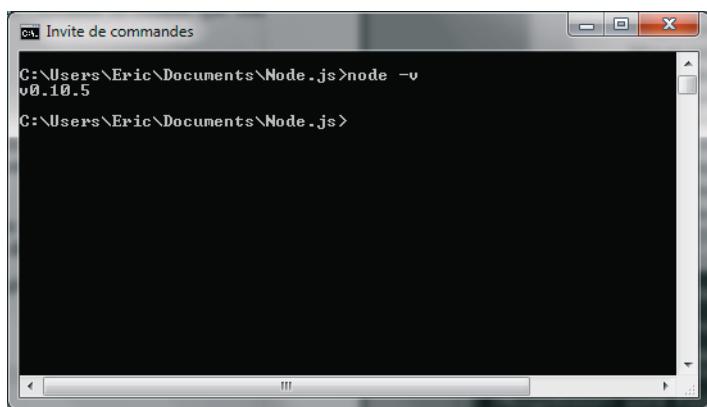
Une fois installé, Node est accessible via la commande `node` exécutée dans un interpréteur de commandes. Pour tester si tout fonctionne correctement, il suffit d'utiliser la commande `node -v`.

Afficher la version de Node utilisée

```
node -v
```

Figure 1–1

Afficher le numéro de la version de Node



Un premier programme avec Node.js

Un programme Node sera écrit dans un ou plusieurs fichiers en JavaScript. Les fichiers auront donc l'extension `.js`. Écrivons un programme JavaScript affichant les nombres de 1 à 10.

Afficher les nombres de 1 à 10

```
console.log("Afficher les nombres de 1 à 10");
for (var i = 1; i <= 10; i++)
    console.log(i);
```

L'instruction `console.log()` permet d'afficher des informations sur l'écran à partir duquel le fichier JavaScript est exécuté. La syntaxe JavaScript est ici accessible, mais Node nous offrira bien plus, afin de réaliser des applications client-serveur complètes.

Une fois le programme écrit, il faut l'exécuter. Pour cela, Node fournit la commande `node` suivie du nom du fichier JavaScript à exécuter. En supposant que le fichier précédent est sauvegardé dans le fichier nommé `test.js`, on tape la commande `node test.js` dans un interpréteur de commandes (depuis le répertoire dans lequel le fichier a été enregistré).

Figure 1–2
Premier programme
avec Node

```
C:\Invite de commandes
C:\Users\Eric\Documents\Node.js>node test.js
Afficher les nombres de 1 à 10
1
2
3
4
5
6
7
8
9
10
C:\Users\Eric\Documents\Node.js>
```

Pour l'instant, les informations sont affichées dans la console, mais nous apprendrons plus tard à afficher des informations dans une fenêtre du navigateur.

Principe de fonctionnement de Node.js

Avant d'écrire des programmes plus complexes, il est important de comprendre comment fonctionne Node.js. Celui-ci a une philosophie différente des autres types de serveurs. En effet, même si le nombre de connexions au serveur augmente, il n'instancie pas un nouveau thread par utilisateur connecté, comme le ferait un serveur Apache par exemple. Pour tout gérer avec un unique thread, il faut que le temps de traitement de chaque requête utilisateur au serveur soit le plus court possible. Mais comment faire court quand la requête peut parfois être importante, par exemple dans le cadre d'un accès à une base de données ?

La solution réside dans la gestion des événements. Dès qu'une action nécessite une certaine durée (accès à la base de données, par exemple), la requête est effectuée mais le programme n'attend pas la réponse. Il se met en disponibilité pour traiter une autre requête d'un autre utilisateur. Puis, lorsque la réponse à la première requête est reçue, un événement est déclenché par le programme qui peut alors traiter cette réponse.

On voit que ce mode de fonctionnement ne permet pas d'écrire du code de façon procédurale, car chaque traitement est effectué dans des gestionnaires d'événements, qui peuvent être nombreux. Le but sera d'écrire des traitements les plus courts possibles, de façon à ne pas bloquer le thread principal du programme qui doit à tout instant pouvoir recevoir les requêtes des utilisateurs.

JavaScript est un langage adapté à ce mode de fonctionnement, car la gestion des événements est incorporée dans le langage. On verra dans la suite de l'ouvrage comment utiliser au mieux la gestion des événements pour écrire des programmes de plus en plus complexes.

Utiliser REPL

REPL est l'abréviation de *Read Eval Print Loop*. C'est un programme qui permet de saisir des instructions JavaScript qui sont interprétées immédiatement, sans avoir à les insérer dans un fichier.

REPL se lance en tapant tout simplement la commande `node`. Un prompt s'affiche dans la console, permettant de saisir les instructions JavaScript.

Voici un exemple d'instructions JavaScript tapées lors d'une session REPL.

Figure 1-3
Une session REPL

```
C:\Users\Eric\Documents\Node.js>node
> var a = 3;
undefined
> var b = 10;
undefined
> a+b
13
> function add(a, b) {
...   return a+b;
...
undefined
> add(10, 20);
13
> add(10, 20);
30
>
```

Les caractères précédés du prompt (`>`) sont ceux tapés par l'utilisateur, tandis que ceux qui sont en faible luminosité sont les résultats affichés par l'interpréteur.

Cet interpréteur est pratique pour effectuer quelques tests, mais nos programmes seront évidemment introduits dans des fichiers que l'on pourra conserver, à la différence des lignes de programme écrites sous REPL.

2

Gestion des modules

La gestion des modules est un élément essentiel dans Node. Un module correspond à un fichier JavaScript, ou à un ensemble de fichiers JavaScript regroupés dans un répertoire. Un module servira à décrire une fonctionnalité qui pourra être utilisée à divers endroits de nos programmes.

On distingue trois types de modules.

- Les modules que nous avons créés pour les besoins de notre application.
- Les modules standards implantés dans Node, qui forment le cœur de celui-ci.
- Les modules externes ayant été créés par d'autres développeurs et que nous pouvons utiliser dans nos programmes.

Dans ce chapitre, nous allons voir comment créer et utiliser ces différents types de modules dans nos applications Node.

Inclusion de modules dans un fichier JavaScript

Node a enrichi le langage JavaScript afin de pouvoir inclure des fichiers externes dans un fichier JavaScript. Pour cela, on utilise l'instruction `require(module)`, dans laquelle `module` est une chaîne indiquant le nom du module à inclure.

Si le module est défini dans un seul fichier – ou correspond à un ensemble de fichiers situés dans un répertoire –, ou s'il s'agit d'un module externe défini par Node, le nom du module indiqué dans la méthode `require()` s'écrira de diverses façons. Dans les sections suivantes, nous vous proposons d'examiner les différentes formes de l'instruction `require()`.

Définir le module dans un fichier JavaScript externe

Utilisons l'instruction `require()` afin d'inclure le fichier `module1.js` dans le fichier `test.js`. Nous employons les instructions `console.log()` pour afficher sur la console du serveur les traitements effectués.

Fichier test.js

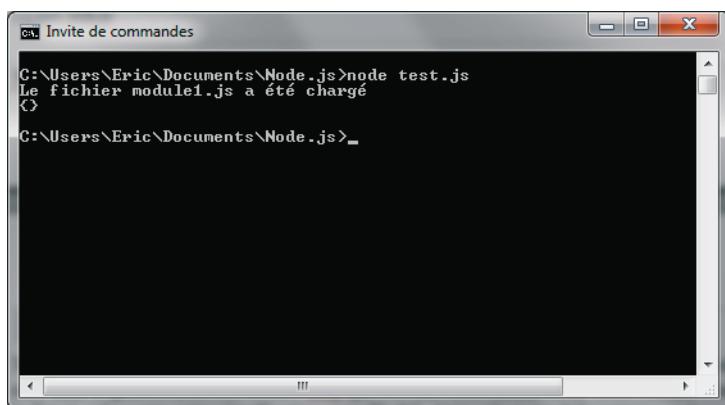
```
var mod1 = require("./module1.js");
console.log(mod1);
```

Nous chargeons le `module1` en indiquant le chemin d'accès du fichier (ici `./` désigne le répertoire courant dans lequel se trouve le fichier `test.js` qui exécute cette instruction). Si le chemin d'accès n'est pas indiqué, cela a une autre signification pour Node : il s'agit soit d'un module externe, soit d'un module défini dans le répertoire `node_modules` (voir plus loin dans cette section).

Fichier module1.js (situé dans le même répertoire que test.js)

```
console.log("Le fichier module1.js a été chargé");
```

Figure 2-1
Chargement de module



L'instruction `console.log(mod1)` produit l'affichage de l'objet `mod1`, à savoir `{ }` qui est un objet vide. On verra dans la section « Visibilité des variables » comment affecter des propriétés et des méthodes aux objets de type module.

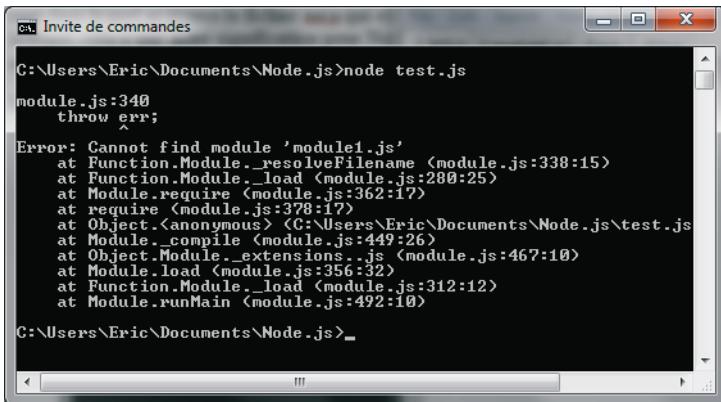
Remarquons que l'on peut aussi écrire `require("./module1")`, c'est-à-dire sans indiquer l'extension `.js` du fichier. En revanche, si vous écrivez `require("module1.js")` ou `require("module1")`, vous obtenez une erreur d'exécution car vous n'avez pas indiqué le chemin d'accès au fichier.

Fichier test.js provoquant une erreur d'exécution (chemin d'accès au module non indiqué)

```
var mod1 = require("module1.js");
console.log(mod1);
```

Figure 2–2

Erreur lors du chargement du module



Définir le module par un ensemble de fichiers situés dans un même répertoire

Plutôt que de définir un module dans un seul fichier comme précédemment, il sera parfois utile de définir plusieurs fichiers dans un même répertoire, s'incluant les uns les autres, et d'ajouter un seul fichier qui sera le fichier « maître » du module.

Fichier test.js

```
var mod1 = require("./module1/module1.js");
console.log(mod1);
```

Le fichier `module1.js` est situé dans un répertoire nommé `module1` (mais il pourrait porter un autre nom).

Fichier module1/module1.js

```
var a = require("./a.js");
var b = require("./b.js");
console.log("Le fichier module1.js a été chargé");
```

Le fichier `module1.js` utilise deux autres modules, nommés `a` et `b`. Ces trois fichiers sont situés dans le répertoire `module1`, défini dans le répertoire principal d'exécution (celui contenant `test.js`).

Fichier module1/a.js

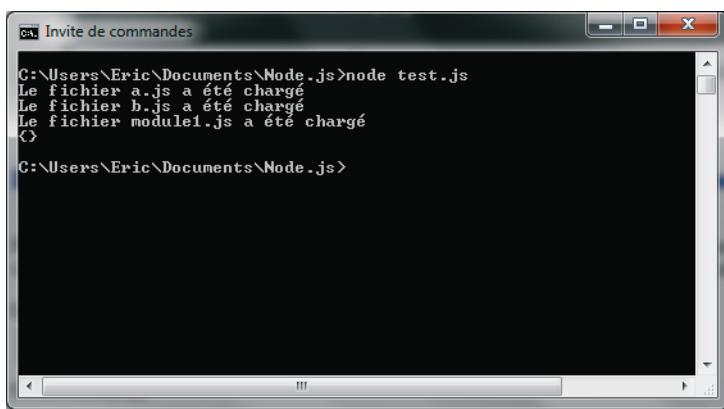
```
console.log("Le fichier a.js a été chargé");
```

Fichier module1/b.js

```
console.log("Le fichier b.js a été chargé");
```

Figure 2–3

Chargement de plusieurs modules

**Utiliser le fichier index.js**

Renommons le fichier `module1.js` en `index.js`. Le fichier `test.js` comportant l'inclusion du module `module1` peut s'écrire de la façon simplifiée suivante.

Fichier test.js

```
var mod1 = require("./module1");
console.log(mod1);
```

Au lieu d'indiquer le nom du fichier complet correspondant au module à inclure, on précise seulement son répertoire (ici `./module1`). Node va rechercher dans ce répertoire un fichier `index.js` qu'il considérera comme le fichier correspondant au module.

Maintenant, supposons que le fichier du module ne soit pas nommé `index.js`, mais `autrenom.js`. Node a prévu ce cas en permettant la création du fichier `package.json` (dans le répertoire du module) qui indiquera le nom du fichier associé à ce module. Un seul fichier `package.json` pourra ainsi se trouver dans chacun des répertoires contenant les modules.

Fichier module1/package.json

```
{"main" : "autrenom.js"}
```

La clé "main" dans le fichier `package.json` permet d'indiquer l'emplacement du fichier maître du module.

Le fichier `test.js` est le même que celui écrit ci-dessus. L'exécution du fichier `test.js` produit le même résultat que précédemment.

Plus d'informations sur le fichier `package.json` sont données dans la section « Le fichier `package.json` » en fin de chapitre.

Utiliser le répertoire `node_modules`

Node permet d'utiliser un répertoire spécial nommé `node_modules`, qui contiendra les modules de notre application. Son intérêt réside dans le fait de simplifier l'écriture du nom des modules, en particulier en omettant les chemins d'accès lors de l'inclusion du module maître dans notre programme.

Ainsi, au lieu d'écrire comme précédemment :

Fichier test.js

```
var mod1 = require("./module1");
console.log(mod1);
```

On pourra maintenant écrire :

Fichier test.js

```
var mod1 = require("module1");
console.log(mod1);
```

On inclut le `module1` en spécifiant uniquement son nom, sans chemin d'accès. Cela n'est possible que si l'on crée un répertoire `node_modules` dans le répertoire principal du serveur, et que l'on y déplace le répertoire `module1` précédemment créé. Le répertoire `node_modules` pourra contenir autant de répertoires que souhaité, chacun d'entre eux étant considéré comme un module.

Le répertoire `module1` associé au `module1` contiendra les mêmes fichiers que ceux décrits précédemment, à savoir :

- Le fichier `module1.js`, ou tout autre nom de fichier qui sera considéré comme maître. Ce fichier sera inclus dans l'application au moyen de l'instruction `require("module1")`.
- Les fichiers `a.js` et `b.js` qui sont inclus par le fichier maître au moyen des instructions `require("./a.js")` et `require("./b.js")`.
- Le fichier `package.json` indiquant le nom du fichier maître dans le cas où celui-ci n'est pas le fichier `index.js`. Le fichier `package.json` est ici inutile si le fichier maître est `index.js`.

Le résultat est identique aux précédentes exécutions.

Utiliser un module standard défini par Node

Pour l'instant, nous avons seulement utilisé les modules que nous avons nous-mêmes créés. Mais Node définit en interne un nombre important de modules, prêts à être immédiatement utilisés. La possibilité d'enrichir cette bibliothèque standard sera abordée dans la prochaine section.

Dans cet ouvrage, nous utiliserons les principaux modules proposés par défaut par Node. Il s'agit aussi bien des utilitaires, tels que la gestion de chaînes de caractères, mais également des modules servant à gérer des serveurs HTTP. Il existe aussi bien d'autres modules, nous les étudierons dans les chapitres qui suivent.

L'inclusion d'un module standard de Node s'effectue comme d'habitude au moyen de l'instruction `require(name)`, dans laquelle `name` est une chaîne de caractères indiquant le nom du module. Ainsi, pour inclure les fonctionnalités liées au module `http` de Node, on écrira simplement `require("http")`.

Inclure les fonctionnalités du module http

```
var http = require("http");
console.log(http);
```

La variable `http` correspondant au module contient les fonctionnalités accessibles dans ce module, telles que celles listées dans la figure 2-4. C'est un objet décrit au format JSON, ayant des propriétés et des méthodes.

Figure 2–4

Contenu du module http

```
C:\Users\Eric\Documents\Node.js>node test.js
{
  parsers: { name: 'parsers', constructor: [Function], max: 1000 },
  STATUS_CODES: {
    '100': 'Continue',
    '101': 'Switching Protocols',
    '102': 'Processing',
    '200': 'OK',
    '201': 'Created',
    '202': 'Accepted',
    '203': 'Non-Authoritative Information',
    '204': 'No Content',
    '205': 'Reset Content',
    '206': 'Partial Content',
    '207': 'Multi-Status',
    '300': 'Multiple Choices',
    '301': 'Moved Permanently',
    '302': 'Moved Temporarily',
    '303': 'See Other',
    '304': 'Not Modified',
    '305': 'Use Proxy',
    '307': 'Temporary Redirect',
    '400': 'Bad Request',
    '401': 'Unauthorized',
    '402': 'Payment Required',
    '403': 'Forbidden',
    '404': 'Not Found',
    '405': 'Method Not Allowed',
    '406': 'Not Acceptable',
    '407': 'Proxy Authentication Required',
    '408': 'Request Time-out',
    '409': 'Conflict',
    '410': 'Gone',
    '411': 'Length Required',
    '412': 'Precondition Failed',
    '413': 'Request Entity Too Large',
    '414': 'Request-URI Too Large',
    '415': 'Unsupported Media Type',
    '416': 'Requested Range Not Satisfiable',
    '417': 'Expectation Failed',
    '418': 'I\'m a teapot',
    '422': 'Unprocessable Entity',
    '423': 'Locked',
    '424': 'Failed Dependency',
    '425': 'Unordered Collection',
    '426': 'Upgrade Required',
    '428': 'Precondition Required',
    '429': 'Too Many Requests',
    '431': 'Request Header Fields Too Large',
    '500': 'Internal Server Error',
    '501': 'Not Implemented',
    '502': 'Bad Gateway',
    '503': 'Service Unavailable',
    '504': 'Gateway Time-out'
  }
}
```

Écrivons un programme permettant d'utiliser le module [http](#). Il s'agit simplement de créer un serveur HTTP qui sera à l'écoute du port 3000 et renverra la chaîne "Bonjour" à chaque fois qu'un utilisateur accédera à l'URL <http://localhost:3000>.

Créer un serveur HTTP qui écoute le port 3000

```
var http = require("http");
var server = http.createServer(function(request, response) {
  response.setHeader("Content-Type", "text/html");
  response.write("<h3>Bonjour</h3>");
  response.end();
});
server.listen(3000);
console.log("Serveur HTTP démarré sur le port 3000");
```

Une fois inclus le module `http`, nous utilisons la méthode `createServer()` sur celui-ci. Cette méthode possède un paramètre correspondant à une fonction de callback qui sera appelée lorsqu'un accès sera effectué sur le serveur. Les paramètres `request` et `response` correspondent respectivement à la requête d'entrée et à la réponse du serveur.

Toutefois, cela ne suffit pas à permettre l'affichage de la réponse sur l'écran du navigateur. En effet, une fois le serveur créé au moyen de `http.createServer()`, il faut le lancer en le mettant à l'écoute d'un port particulier (ici, le port 3000). Pour cela, on utilise la méthode `server.listen(3000)` sur l'objet `server` de classe `http.Server` retourné par `http.createServer()`, qui permet de mettre le serveur à l'écoute du port 3000 et de gérer les requêtes des clients.

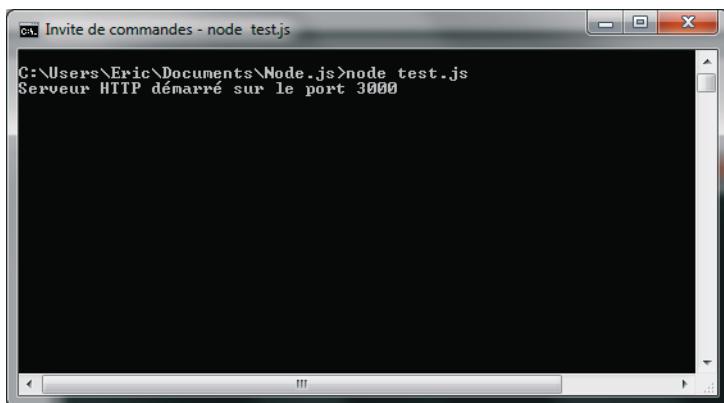
Le traitement de la fonction de callback va consister à fournir une réponse à afficher dans le navigateur qui a effectué la requête. Ceci se déroule de la façon suivante.

- On indique que le résultat devra être affiché sous forme HTML, de manière à ce que les balises HTML soient correctement interprétées, par `response.setHeader("Content-Type", "text/html")`. Si cette instruction n'était pas présente, les affichages contenant des balises HTML seraient effectués sans tenir compte de celles-ci.
- On effectue un affichage dans l'écran du navigateur au moyen de `response.write("<h3>Bonjour</h3>")`. Plusieurs instructions de ce type peuvent se suivre et le résultat sera cumulé à l'écran.
- Lorsque les instructions `response.write()` sont terminées, on indique au navigateur qu'il peut procéder à l'affichage au moyen de l'instruction `response.end()`. Sans cette instruction, le navigateur se bloque et n'affiche pas le résultat des instructions `response.write()` précédentes.

Testons le programme en le lançant dans un interpréteur de commandes.

Figure 2-5

Démarrage d'un serveur HTTP

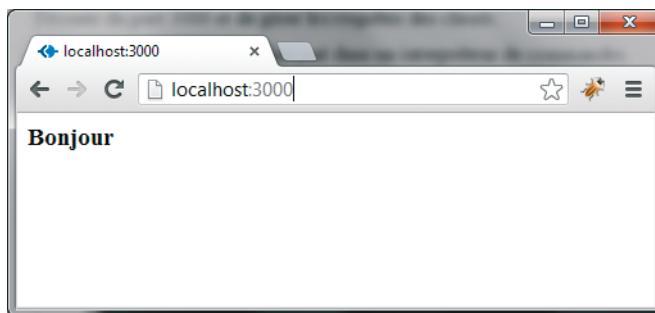


```
C:\Users\Eric\Documents\Node.js>node test.js
Serveur HTTP démarré sur le port 3000
```

Une fois le programme du serveur lancé, on peut utiliser un navigateur pour effectuer des requêtes sur le port 3000.

Figure 2–6

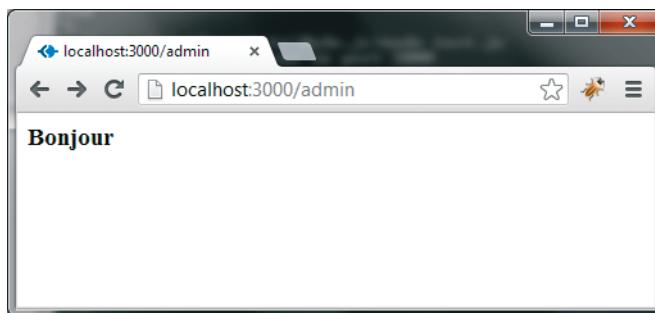
Visualisation d'une réponse du serveur HTTP



Remarquons que quelle que soit la forme de l'URL HTTP lancée sur le port 3000, le résultat envoyé par le serveur est toujours le même. Ceci est tout à fait normal car on n'effectue qu'un seul traitement, sans tenir compte de la forme de l'URL. Par exemple, avec l'URL `http://localhost:3000/admin` :

Figure 2–7

Utiliser une autre URL pour accéder au serveur HTTP



Notons que le précédent programme peut aussi être écrit sous la forme condensée suivante.

Autre forme d'écriture du précédent programme

```
var http = require("http");
http.createServer(function(request, response) {
    response.setHeader("Content-Type", "text/html");
    response.write("<h3>Bonjour</h3>");
    response.end();
}).listen(3000);
console.log("Serveur HTTP démarré sur le port 3000");
```

Nous n'utilisons plus la variable `server`, car nous chaînons directement la méthode `listen()` sur le résultat de `http.createServer()`. Cette seconde forme d'écriture est plus courante que la première.

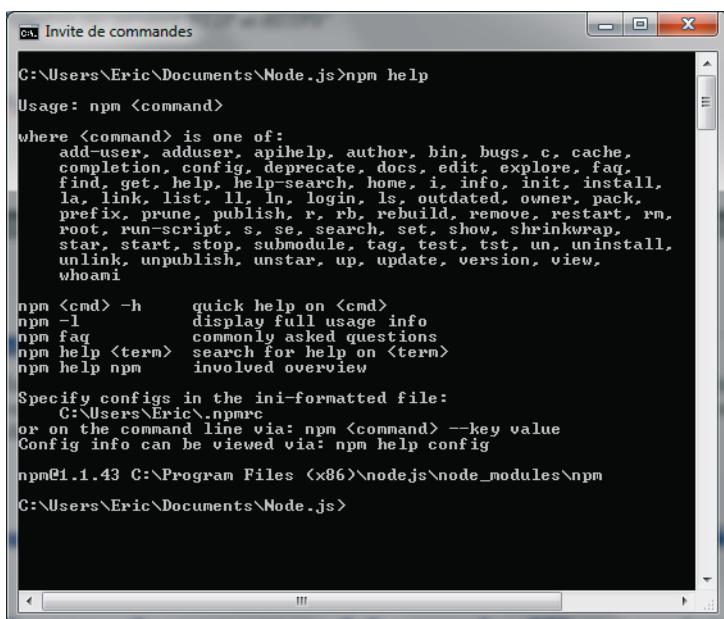
Cet exemple ne sert qu'à montrer comment utiliser les modules externes dans nos programmes. Des explications plus détaillées sur la construction d'un serveur HTTP sont données dans le chapitre 10, « Gestion des connexions HTTP ».

Télécharger de nouveaux modules avec npm

En plus des modules standards proposés par Node (comme le module `http` vu précédemment), de nombreux développeurs en ont créé d'autres qui permettent d'effectuer toutes sortes de traitements supplémentaires. Ces modules peuvent être téléchargés au moyen de l'utilitaire `npm` fourni avec Node (`npm` signifie *Node Package Manager*).

Pour voir les possibilités offertes par `npm`, il suffit de taper la commande `npm help` dans un interpréteur de commandes.

Figure 2-8
Aide sur la commande npm



```
C:\ Invité de commandes
C:\Users\Eric\Documents\Node.js>npm help
Usage: npm <command>

where <command> is one of:
  add-user, adduser, apihelp, author, bin, bugs, c, cache,
  completion, config, deprecate, docs, edit, explore, faq,
  find, get, help, help-search, home, i, info, init, install,
  la, link, list, ll, ln, login, ls, outdated, owner, pack,
  prefix, prune, publish, r, rb, rebuild, remove, restart, rm,
  root, run-script, s, se, search, set, show, shrinkwrap,
  star, start, stop, submodule, tag, test, tst, un, uninstall,
  unlink, unpublish, unstar, up, update, version, view,
  whoami

npm <cmd> -h      quick help on <cmd>
npm -l      display full usage info
npm faq    commonly asked questions
npm help <term>  search for help on <term>
npm help npm   involved overview

Specify configs in the ini-formatted file:
  C:\Users\Eric\.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

npm@1.1.43 C:\Program Files (x86)\nodejs\node_modules\npm
C:\Users\Eric\Documents\Node.js>
```

On voit que la plupart des commandes s'utilisent sous la forme `npm commande`, dans laquelle `commande` est le nom de la commande que l'on souhaite exécuter. Le détail d'une commande pourra être obtenu au moyen de `npm commande -h` (figure 2-8).

Une des commandes les plus utilisées de `npm` est celle permettant d'installer un nouveau module. On l'utilise sous la forme `npm install modulename`, dans laquelle `modulename` est le nom du module que l'on souhaite installer.

Bien sûr, lorsqu'on débute avec Node, on ne connaît pas les différents modules que celui-ci propose en téléchargement. Une liste des plus populaires se trouve sur le site <https://npmjs.org/>, accessible depuis le site officiel de Node (onglet NPM REGISTRY). La commande `npm search name` permet également de rechercher les modules incluant le `name` indiqué, que l'on pourra ensuite installer par `npm install`.

Le tableau 2-1 présente la liste de quelques-unes des commandes possibles avec l'exécutable `npm`.

Tableau 2-1 Commandes npm

Commande	Signification
<code>npm install modulename</code>	Installe le module indiqué dans le répertoire <code>node_modules</code> de l'application. Ce module ne sera accessible que pour l'application dans laquelle il est installé. Pour utiliser ce module dans une autre application Node, il faudra l'installer, de la même façon, dans cette autre application, ou l'installer en global (voir l'option <code>-g</code> ci-dessous).
<code>npm install modulename -g</code>	Installe le module indiqué en global, il est alors accessible pour toutes les applications Node.
<code>npm install modulename@version</code>	Installe la version indiquée du module. Par exemple, <code>npm install connect@2.7.3</code> pour installer la version 2.7.3 du module <code>connect</code> .
<code>npm install</code>	Installe dans le répertoire <code>node_modules</code> , les modules indiqués dans la clé <code>dependencies</code> du fichier <code>package.json</code> . Par exemple, le fichier <code>package.json</code> est de la forme suivante : <pre>{ "dependencies": { "express": "3.2.6", "jade": "*", "stylus": "*" } }</pre> Ceci indique de charger la version 3.2.6 d'Express, avec les dernières versions de Jade et de Stylus, lorsque la commande <code>npm install</code> sera lancée.
<code>npm start</code>	Démarre l'application Node indiquée dans la clé <code>start</code> , elle-même incluse dans la clé <code>scripts</code> . Par exemple, le fichier <code>package.json</code> est de la forme suivante : <pre>{ "scripts": { "start": "node app" } }</pre> Ceci indique d'exécuter la commande <code>node app</code> , lorsque la commande <code>npm start</code> sera lancée.

Tableau 2-1 Commandes npm (suite)

Commande	Signification
<code>npm uninstall modulename</code>	Supprime le module indiqué, s'il a été installé en local dans <code>node_modules</code> .
<code>npm uninstall modulename -g</code>	Supprime le module indiqué, s'il a été installé en global.
<code>npm update modulename</code>	Met à jour le module indiqué avec la dernière version.
<code>npm update</code>	Met à jour tous les modules déjà installés, avec la dernière version.
<code>npm outdated</code>	Liste les modules qui sont dans une version antérieure à la dernière version disponible.
<code>npm ls</code>	Affiche la liste des modules installés en local dans <code>node_modules</code> , avec leurs dépendances.
<code>npm ls -g</code>	Similaire à <code>npm ls</code> , mais affiche les modules installés en global.
<code>npm ll</code>	Similaire à <code>npm ls</code> , mais affiche plus de détails.
<code>npm ll modulename</code>	Affiche les détails sur le module indiqué.
<code>npm search name</code>	Recherche sur Internet les modules possédant le mot <code>name</code> dans leur nom ou description. Plusieurs champs <code>name</code> peuvent être indiqués, séparés par un espace. Par exemple, <code>npm search html5</code> pour rechercher tous les modules ayant <code>html5</code> dans leur nom ou description.
<code>npm link modulename</code>	Il peut parfois arriver qu'un module positionné en global soit malgré tout inaccessible par <code>require()</code> . Cette commande permet alors de lier le module global à un répertoire local (dans <code>node_modules</code>) de façon à le rendre accessible.
<code>npm info modulename version</code>	Indique le numéro de la version la plus récente du module, disponible sur Internet.

À titre d'exemple, installons le module `colors` à l'aide de la commande `npm install colors`. Ce module permet d'afficher du texte en couleurs dans les résultats donnés dans l'interpréteur de commandes, au moyen de l'instruction `console.log()`.

Figure 2-9
Installation du module colors

```
C:\Users\Eric\Documents\Node.js>npm install colors
npm http GET https://registry.npmjs.org/colors
npm http 304 https://registry.npmjs.org/colors
colors@0.6.0-1 node_modules\colors
C:\Users\Eric\Documents\Node.js>
```

Une fois ce module installé (ici, en local dans le répertoire `node_modules` de l'application), nous voyons le nouveau répertoire `colors` créé dans le répertoire `node_modules`. Ce répertoire contient divers fichiers, dont en particulier le fichier `colors.js` et le fichier `package.json`.

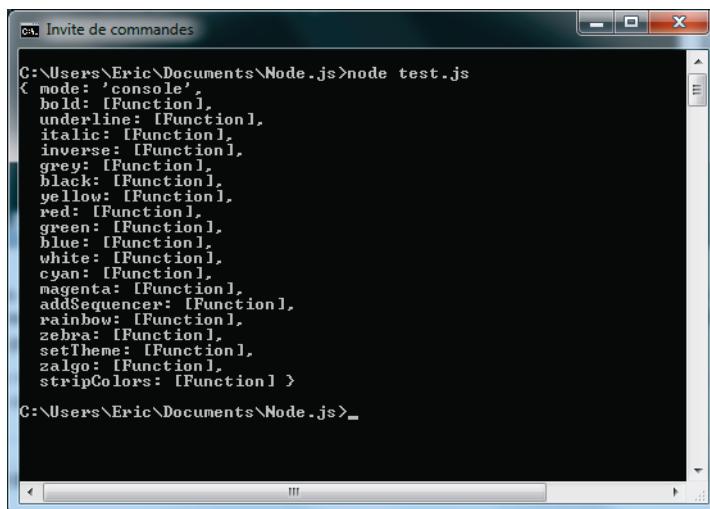
Afin de voir ce que permet le module `colors`, écrivons le petit programme de test (`test.js`) suivant qui affiche le contenu du module.

Afficher le contenu du module colors

```
var colors = require("colors");
console.log(colors);
```

Figure 2–10

Contenu du module colors



```
C:\Users\Eric\Documents\Node.js>node test.js
{
  mode: 'console',
  bold: [Function],
  underline: [Function],
  italic: [Function],
  inverse: [Function],
  grey: [Function],
  black: [Function],
  yellow: [Function],
  red: [Function],
  green: [Function],
  blue: [Function],
  white: [Function],
  cyan: [Function],
  magenta: [Function],
  addSequencer: [Function],
  rainbow: [Function],
  zebra: [Function],
  setTheme: [Function],
  zalgo: [Function],
  stripColors: [Function]
}
C:\Users\Eric\Documents\Node.js>
```

Le module `colors` s'utilise pour afficher des textes en diverses couleurs dans la console du serveur. Par exemple, le programme suivant affiche un premier texte en vert et le second en bleu.

Utiliser le module colors pour écrire en différentes couleurs

```
var colors = require("colors");
console.log("Ceci est en vert".green);
console.log("Ceci est en bleu".blue);
```

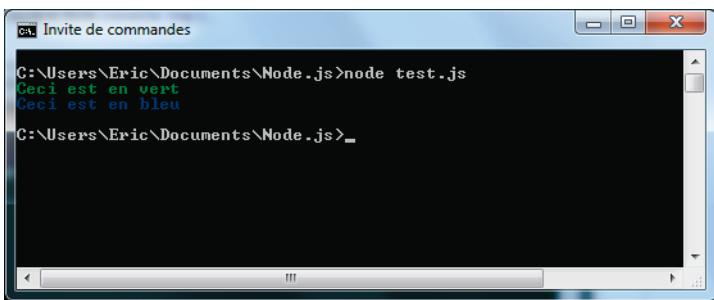
Le module `colors` définit (entre autres) les propriétés `green` et `blue` à la classe `String` de JavaScript, d'où la possibilité d'utiliser ces propriétés sur des chaînes de caractères. Le programme précédent peut également s'écrire comme suit.

Autre forme d'utilisation des méthodes du module colors

```
var colors = require("colors");
console.log(colors.green("Ceci est en vert"));
console.log(colors.blue("Ceci est en bleu"));
```

Dans cette forme d'écriture du programme, on voit mieux l'utilisation du module `colors` à travers l'usage des méthodes `green()` et `blue()`. Dans les deux cas, le résultat s'affiche comme représenté à la figure 2-11.

Figure 2-11
Utilisation du module colors



Écrire un module proposant de nouvelles fonctionnalités

L'intérêt d'utiliser des modules est d'offrir des nouvelles fonctionnalités aux programmes qui vont les inclure au moyen de l'instruction `require()`. Par exemple, l'inclusion du module `colors` dans le programme de la section précédente permet à ce programme d'utiliser les méthodes `green()` et `blue()` au moyen de `colors.green()` et `colors.blue()`. De même, l'inclusion du module `http` standard de Node permet d'utiliser la méthode `http.createServer()` qui crée un serveur HTTP.

Nous allons donc étudier dans cette section comment définir des méthodes dans un module afin qu'elles soient utilisables dans d'autres modules ou programmes.

Dans les exemples suivants, nous utilisons le module défini dans `module1.js`, situé au même emplacement que le programme `test.js`. Le module `module1.js` contient une méthode `add(a, b)` permettant l'ajout de deux nombres `a` et `b`.

Fichier module1.js

```
function add(a, b) {
  return a+b;
}
```

Nous souhaitons utiliser cette méthode dans le fichier `test.js`, en incluant directement le module.

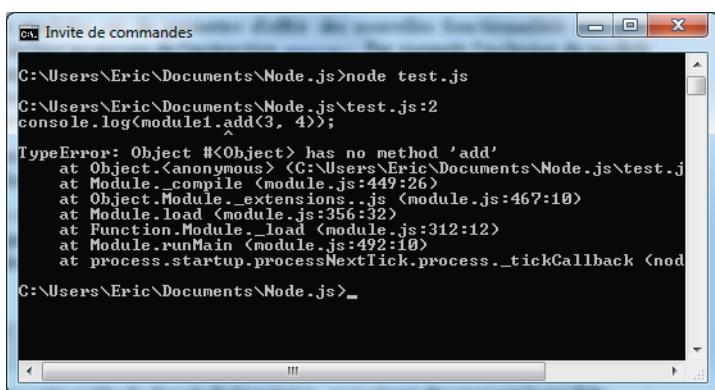
Fichier test.js

```
var module1 = require("./module1");
console.log(module1.add(3, 4));
```

Le `module1` est inclus au moyen de l'instruction `require()` et la méthode `add()` est appelée par `module1.add()`. Le résultat est affiché dans la console du serveur au moyen de `console.log()`.

Figure 2–12

Erreur dans l'utilisation de la méthode `add()` du module



On voit que la méthode `add()` n'est pas connue dans le programme du serveur, même si le module est inclus correctement. En fait, `add()` est une méthode privée du module `module1` et ne peut donc être utilisée que dans celui-ci. Pour la rendre accessible dans un autre module ou programme, il faut l'exporter. Ceci est possible au moyen de l'objet `module.exports` qui contient les propriétés et les méthodes que le module souhaite exporter. L'objet `module` est décrit en détail dans la prochaine section.

On modifie le fichier du module de la façon suivante.

Exportation de la méthode `add()` du module (fichier `module1.js`)

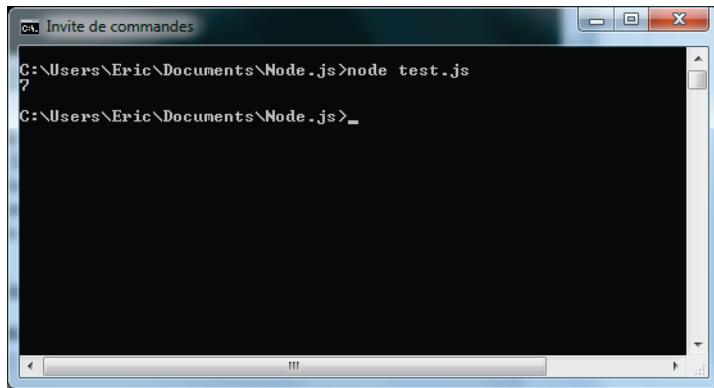
```
function add(a, b) {
  return a+b;
}

module.exports.add = add;
```

On ajoute une propriété `add` dans l'objet `module.exports` associé au module. La valeur de cette propriété correspond à la méthode `add()` définie dans la ligne précédente. L'ajout de la propriété `add` dans `module.exports` permet maintenant d'utiliser `module1.add()` dans le fichier `test.js`.

On obtient alors :

Figure 2–13
Utilisation correcte de la
méthode `add()` du module



Le résultat de `module1.add(3, 4)` produit l'affichage de la valeur 7, montrant que la méthode `add()` est maintenant accessible à l'extérieur du module.

Intéressons-nous maintenant à une autre façon d'exporter une méthode dans un module. Par exemple, nous créons la nouvelle méthode `mult(a, b)` qui multiplie les deux valeurs `a` et `b`.

Définir la méthode `mult(a, b)` dans le module

```
function add(a, b) {  
    return a+b;  
}  
  
module.exports.add = add;  
  
module.exports.mult = function(a, b) {  
    return a * b;  
}
```

La méthode `mult()` est directement définie dans `module.exports.mult`, sans passer par une fonction intermédiaire comme nous l'avons fait pour la méthode `add()`. Ces deux façons de procéder sont strictement équivalentes, la seconde permettant d'aller plus vite car il y a moins de lignes de code à écrire.

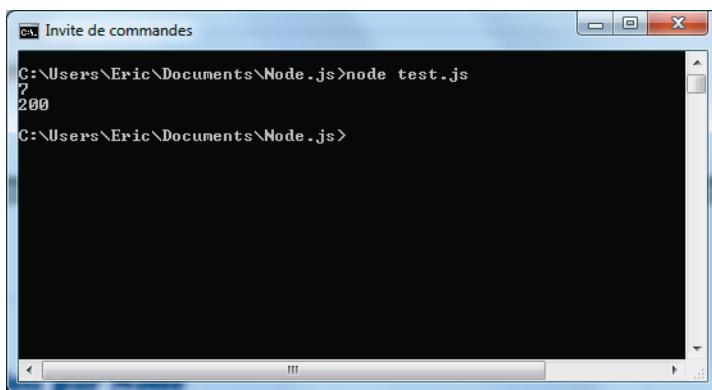
Le fichier `test.js` utilisant les méthodes `add()` et `mult()` est le suivant.

Fichier test.js

```
var module1 = require("./module1");
console.log(module1.add(3, 4));
console.log(module1.mult(10, 20));
```

Figure 2–14

Utilisation des méthodes `add()` et `mult()` du module



L'objet `module.exports` contient donc les propriétés et méthodes du module `modulename` qui seront accessibles dans un autre module via l'instruction `require(modulename)`.

Cas particulier : un module composé d'une fonction principale

Il est fréquent qu'un module soit composé d'une fonction principale. Par exemple, le framework Express que l'on étudiera dans la suite de l'ouvrage comprend la méthode `express()` en tant que principale fonction du module `express.js`. Cela ne signifie pas que le module contient une seule fonction, mais que cette fonction est celle que l'on considère comme étant la fonction principale du module.

Le fait de désigner une fonction principale du module procure des facilités d'écriture lors de son utilisation. Ainsi, en supposant que le `module1` défini précédemment comporte la fonction `add()` en tant que fonction principale, on écrirait alors :

Fichier module.1.js

```
function add(a, b) {  
    return a+b;  
}  
  
module.exports = add;  
  
module.exports.mult = function(a, b) {  
    return a * b;  
}
```

Plutôt que de rattacher la méthode `add()` à la propriété `add` de l'objet `module.exports`, on l'affecte directement à l'objet `module.exports`. La fonction `add()` devient ainsi la fonction principale du module, vu que l'objet `module.exports` est unique pour un module. Bien sûr, on ne peut avoir qu'une seule fonction principale par module.

En revanche, la fonction `mult()` est rattachée, comme précédemment, à la propriété `mult` de l'objet `module.exports`. D'autres fonctions peuvent ainsi être rattachées à d'autres propriétés de l'objet `module.exports`. Ces fonctions sont utilisées comme indiqué précédemment.

Voyons maintenant comment utiliser la fonction `add()` dans un autre module, ici le fichier `test.js`.

Fichier test.js

```
var module1 = require("./module1");  
console.log(module1(3, 4));          // appelle la fonction add(3, 4)  
console.log(module1.mult(3, 4));     // appelle la fonction mult(3, 4)
```

Remarquez que la fonction `mult()` est accédée de façon traditionnelle, car il ne s'agit pas de la fonction principale du module.

Le `module1` est chargé par `require("./module1")`. La fonction `add()` étant la fonction principale du module, elle est accédée via l'instruction `module1(3, 4)`, ce qui peut également s'écrire de la façon suivante.

Autre façon d'accéder à la fonction add()

```
console.log(require("./module1")(3, 4));
```

Remarquez que dans ce cas, le nom du module devrait plutôt être `add.js`, car il serait plus lisible d'écrire `add(3, 4)` que `module1(3, 4)`.

L'objet module défini par Node

Chaque module définit en interne un objet nommé `module`. Ainsi, tout module pourra accéder à cette variable interne, qui sera différente selon chaque module. Par exemple, utilisons le `module1` défini précédemment et affichons la variable `module` dans le `module1` ainsi que dans le module associé au programme principal.

Même si le nom de la variable `module` est le même dans tous les modules, sa valeur est différente d'un module à l'autre.

Fichier test.js

```
var module1 = require("./module1");
console.log(module);
```

Fichier module1.js

```
function add(a, b) {
    return a+b;
}

module.exports.add = add;

module.exports.mult = function(a, b) {
    return a * b;
}

console.log(module);
```

Dans chacun de ces modules, nous affichons la valeur de l'objet `module` associé au module.

On obtient le résultat suivant.

Figure 2-15
Contenu de l'objet module

```
C:\Users\Eric\Documents\Node.js>node test.js
{
  id: 'C:\\Users\\Eric\\Documents\\Node.js\\module1.js',
  exports: { add: [Function: add], mult: [Function] },
  parent: null,
  __filename: 'C:\\Users\\Eric\\Documents\\Node.js\\test.js',
  __dirname: 'C:\\Users\\Eric\\Documents\\Node.js',
  __loaded: true,
  __children: [Circular],
  __paths: [
    'C:\\Users\\Eric\\Documents\\Node.js\\node_modules',
    'C:\\Users\\Eric\\Documents\\node_modules',
    'C:\\Users\\node_modules',
    'C:\\node_modules',
    'C:\\\\node_modules' ]
}
{
  id: 'C:\\\\node_modules',
  exports: { },
  parent: null,
  __filename: 'C:\\\\Users\\Eric\\Documents\\Node.js\\node_modules',
  __dirname: 'C:\\\\Users\\Eric\\Documents\\Node.js',
  __loaded: false,
  __children: [Circular],
  __paths: [
    'C:\\\\Users\\Eric\\Documents\\Node.js\\node_modules',
    'C:\\\\Users\\Eric\\node_modules',
    'C:\\\\Users\\node_modules',
    'C:\\node_modules',
    'C:\\\\node_modules' ]
}
C:\Users\Eric\Documents\Node.js>
```

Le premier module qui s'affiche est `module1`, vu que c'est celui qui est chargé dès le début. Il est suivi par le module associé au fichier `test.js`. Le résultat est ici affiché au format JSON. On voit que l'objet `module` possède des propriétés et des méthodes telles que `id`, `exports`, `parent`, `filename`, `loaded`, `children` et `paths`, dont les valeurs sont différentes d'un module à l'autre. Ceci signifie que l'objet `module` est bien associé à chaque module de façon indépendante des autres modules.

La tableau 2-2 reprend les différentes propriétés définies dans l'objet `module`.

Des variables internes aux modules ont été créées afin de faciliter l'accès aux propriétés les plus usuelles :

- `_filename` est équivalente à `module.filename` ;
- `_dirname` permet d'accéder au répertoire contenant le fichier `module.filename` ;
- `exports` est équivalente à `module.exports`.

Tableau 2–2 Propriétés et méthodes définies dans l'objet module

Propriété	Signification
<code>id</code>	Identifiant unique (string) associé au module. Correspond au chemin d'accès vers le module sur le serveur, ou « <code>.</code> » pour indiquer le module associé au <code>main</code> (celui qui est directement exécuté par la commande <code>node</code>).
<code>exports</code>	Objet contenant les propriétés et méthodes du module exportées vers les autres modules. Elles seront accessibles dans ces autres modules grâce à l'instruction <code>require(modulename)</code> qui retourne l'objet <code>module.exports</code> .
<code>parent</code>	Objet <code>module</code> parent de celui-ci, ou null si le module est le <code>main</code> .
<code>filename</code>	Chemin d'accès (complet) vers le module sur le serveur.
<code>loaded</code>	Indique si le module a été complètement chargé (<code>true</code>) ou non (<code>false</code>).
<code>children</code>	Tableau des objets <code>module</code> correspondant aux différents modules inclus dans celui-ci.
<code>paths</code>	Répertoires <code>node_modules</code> dans lesquels les modules sont recherchés. Plusieurs répertoires peuvent s'y trouver car Node les recherche d'abord dans le répertoire courant de l'application, puis remonte vers le parent, puis le parent du parent, etc.

Mise en cache des modules

Lorsqu'un module est chargé grâce à l'instruction `require()`, le module est mis en cache et n'est donc pas rechargé si une autre instruction `require()` le redemande.

Pour s'en assurer, le programme suivant effectue deux appels successifs à `require("./module1")`, mais seul le premier appel affiche un message dans la console du serveur.

Fichier test.js

```
var module1 = require("./module1");
console.log("module1 est chargé");
var module1_bis = require("./module1");
console.log("module1 est chargé");
```

Fichier module1.js

```
console.log("module1 est en cours de chargement");
```

Figure 2-16

Plusieurs chargements
du même module

```
C:\Users\Eric\Documents\Node.js>node test.js
module1 est en cours de chargement
module1 est chargé
module1 est chargé
C:\Users\Eric\Documents\Node.js>_
```

Le message inscrit dans le `module1` est effectivement affiché une seule fois, montrant que le code inscrit dans ce module n'est exécuté qu'une seule fois.

Visibilité des variables

Chaque variable créée dans un module, au moyen du mot-clé `var`, est locale à ce module. Une telle variable ne peut être visible dans un autre module que si elle est exportée (c'est-à-dire mise dans `module.exports`, ou plus simplement `exports`).

Une variable créée dans un module, sans utiliser le mot-clé `var`, est considérée comme une variable globale, donc accessible aux autres modules. Dans ce cas, Node inscrit la variable en tant que propriété de l'objet `global`, qui est commun à tous les modules.

Dans le programme suivant, nous définissons les variables globales `a` et `b` respectivement dans les modules `test` et `module1`, et nous les utilisons dans l'autre module afin de montrer qu'elles sont communes à tous les modules.

Fichier `test.js`

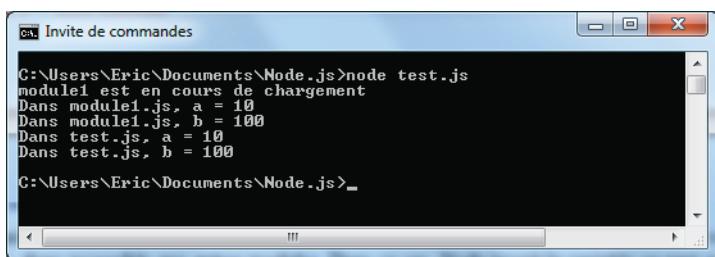
```
a = 10;
require("./module1");
console.log("Dans test.js, a = " + a);
console.log("Dans test.js, b = " + b);
```

Fichier module1.js

```
console.log("module1 est en cours de chargement");
console.log("Dans module1.js, a = " + a);
b = 100;
console.log("Dans module1.js, b = " + b);
```

Exécutons `test.js` :

Figure 2–17
Visibilité des variables
dans les modules



Le fichier package.json

On a vu que le fichier `package.json` pouvait servir à indiquer le nom du module que l'on souhaite charger (voir la section « Utiliser le fichier `index.js` » décrite dans ce chapitre). Mais en tant que fichier JSON, on peut aussi indiquer d'autres paramètres, que nous décrivons ci-après.

Voici un exemple de fichier `package.json` reprenant certaines des informations dont nous avons précédemment parlé.

Fichier package.json

```
{
  "name" : "application-name",
  "version" : "0.0.1",
  "scripts" : {
    "start" : "node app"
  },
  "main" : "colors",
  "dependencies" : {
    "express" : "3.2.6",
    "jade" : "*",
    "stylus" : "*"
  }
}
```

Le tableau 2-3 présente la description des champs du fichier `package.json`.

Tableau 2-3 Champs du fichier package.json

Champ	Signification
<code>name</code>	Nom de l'application.
<code>version</code>	Numéro de version de l'application, de la forme x.y.z. Le dernier indice (champ z) indique un changement de version mineure.
<code>scripts</code>	Objet dont la clé <code>start</code> indique la commande Node exécutée lors de <code>npm start</code> .
<code>main</code>	Fichier principal du module, si celui-ci est chargé par <code>require(nom_reperoire)</code> . Si la clé <code>main</code> n'est pas indiquée, le fichier <code>index.js</code> est chargé.
<code>dependencies</code>	Objet décrivant les noms des modules et leurs numéros de version, requis pour que le module fonctionne. Les modules seront récupérés sur Internet lors de l'exécution de l'instruction <code>npm install</code> . Pour indiquer la dernière version disponible d'un module, indiquer * pour le numéro de version.

3

Gestion des événements

Dans Node, la gestion des événements s'effectue au moyen du module standard `events`. Celui-ci comporte une classe nommée `EventEmitter` qui permet de créer des objets JavaScript pouvant émettre et recevoir des événements.

Pourquoi utiliser la classe `events.EventEmitter` ?

Tout d'abord, commençons par le plus important à nos yeux en expliquant l'utilité de la classe `events.EventEmitter`. Chaque fois que vous devrez gérer des événements quelconques (frappe d'une touche du clavier, réception de message, connexion à un serveur, etc.), vous ferez intervenir des objets ou des classes d'objets pouvant supporter la gestion des événements. Ces objets, pour avoir la capacité de recevoir de tels événements, doivent hériter des fonctionnalités de la classe `events.EventEmitter` définie dans Node.

Bien sûr, Node définit déjà en standard des objets ou des classes supportant la gestion des événements. Par exemple, la classe `http.Server` définie dans le module `http` de Node, et permettant de créer des serveurs HTTP, dérive de la classe `events.EventEmitter`, ce qui lui permet de recevoir des événements tels que les requêtes des utilisateurs. On avait ainsi écrit dans le précédent chapitre un petit bloc de code permettant de créer un serveur HTTP simple, qui retourne « Bonjour » chaque fois qu'un utilisateur se connecte sur le port 3000 du serveur au moyen du protocole HTTP (par exemple, en saisissant l'URL `http://localhost:3000` dans la barre d'adresses du navigateur). Ce code est le suivant.

Créer un serveur HTTP qui écoute le port 3000

```
var http = require("http");
var server = http.createServer(function(request, response) {
  response.setHeader("Content-Type", "text/html");
  response.write("<h3>Bonjour</h3>");
  response.end();
});
server.listen(3000);
console.log("Serveur HTTP démarré sur le port 3000");
```

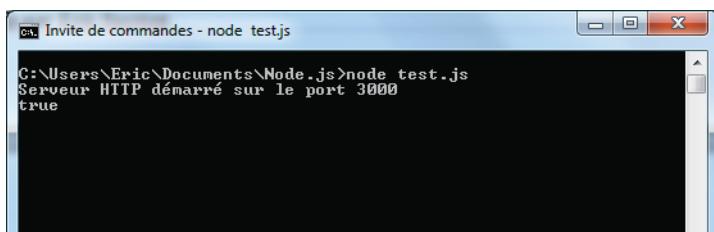
Dans cet exemple, la classe `events.EventEmitter` n'est pas visible directement dans le code, car c'est la classe `http.Server` qui en hérite. Pour s'en assurer, il suffit d'ajouter les lignes suivantes à la fin du code précédent.

Vérifier que la classe `http.Server` hérite de `events.EventEmitter`

```
var events = require("events");
console.log(server instanceof events.EventEmitter);
```

Figure 3-1

Démarrage d'un serveur HTTP



L'affichage de la valeur `true` signifie ici que l'objet `server` de classe `http.Server` est bien une instance de la classe `events.EventEmitter`.

Maintenant que nous avons vu l'utilité de la classe `events.EventEmitter`, regardons comment créer des objets de cette classe et comment les utiliser.

Créer un objet de la classe `events.EventEmitter`

La classe `EventEmitter` fait partie du module `events`. Pour y accéder, il faut donc inclure le module standard `events`. On crée un objet de cette classe au moyen de `new events.EventEmitter()`.

Créer un objet de classe `events.EventEmitter`

```
var events = require("events");
var obj1 = new events.EventEmitter();
console.log(obj1);
```

Nous créons ici un objet `obj1` de classe `events.EventEmitter`. Nous l'affichons ensuite au moyen de `console.log()`.

Ceci peut également s'écrire de façon raccourcie comme dans le code suivant.

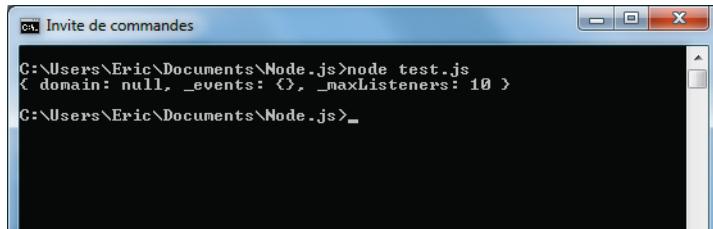
Autre façon d'écrire la création d'un objet de classe `events.EventEmitter`

```
var obj1 = new (require("events")).EventEmitter();
console.log(obj1);
```

Notez ici les parenthèses entourant l'instruction `require("events")`. Si vous les omettez, cela ne fonctionnera pas comme souhaité.

Dans les deux cas, on obtient l'affichage du contenu de l'objet `obj1`.

Figure 3–2
Objet de classe
`events.EventEmitter`



Gérer des événements sur un objet de classe `events.EventEmitter`

Utiliser les méthodes `addListener()` et `emit()`

Une fois l'objet `events.EventEmitter` créé, il peut être utilisé pour gérer les événements qui lui sont adressés. Pour cela, les méthodes `addListener()` et `emit()` sont disponibles pour les objets de la classe `events.EventEmitter`. La méthode `addListener()`

permet d'ajouter un gestionnaire d'événements sur l'objet, tandis que la méthode `emit()` envoie un événement sur cet objet.

Tableau 3–1 Méthodes de gestion des événements

Méthode	Signification
<code>obj.addListener(event, listener)</code>	Ajouter, sur l'objet qui utilise la méthode (ici, <code>obj</code>), un gestionnaire d'événements pour l'événement <code>event</code> . Lorsque cet événement se produit, la fonction de callback <code>listener</code> se déclenche.
<code>obj.on(event, listener)</code>	Équivaut à <code>addListener(event, listener)</code> .
<code>obj.emit(event)</code>	Déclenche l'événement <code>event</code> sur l'objet qui utilise la méthode (ici, <code>obj</code>).

Voyons comment utiliser ces méthodes dans le programme suivant.

Utiliser `addListener()` et `emit()`

```
var events = require("events");
var obj1 = new events.EventEmitter();

obj1.addListener("event1", function() {
    console.log("L'objet obj1 a reçu un événement event1");
});

obj1.emit("event1");
```

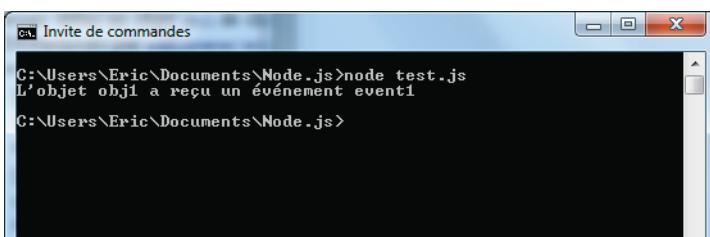
Nous avons défini un objet `obj1` de classe `events.EventEmitter`, sur lequel nous attachons un gestionnaire d'événements par `addListener(event, listener)`. Cette méthode prend les deux paramètres `event` et `listener` suivants.

- Le paramètre `event` est une chaîne de caractères servant à indiquer le nom de l'événement que l'on souhaite gérer dans ce gestionnaire. C'est une chaîne quelconque (dans notre exemple, nous l'avons appelé "`event1`").
- Le paramètre `listener` correspond à une fonction de callback qui sera appelée automatiquement lorsque l'événement se produira. Ici, la fonction de traitement consiste simplement à afficher un message sur la console du serveur, mais dans la réalité le traitement sera plus conséquent.

La seconde méthode utilisée ici est `emit(event)`. Elle consiste à générer l'événement indiqué dans le paramètre `event` (ici, "`event1`").

Remarquez que c'est le même objet `obj1` qui observe l'événement (par l'intermédiaire de `addListener()`) et qui l'émet (par `emit()`).

Figure 3–3
Émission et réception
d'événement sur un objet



On voit ici que le déclenchement de l'événement par `emit()` provoque l'exécution de la fonction de callback mise en place dans `addListener()`. Si l'événement était déclenché avant que la fonction de callback soit positionnée, ce déclenchement ne serait pas pris en compte par la fonction de callback, car il serait déclenché trop tôt.

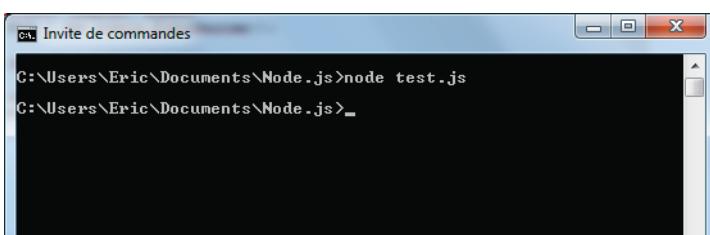
Déclenchement d'un événement trop tôt

```
var events = require("events");
var obj1 = new events.EventEmitter();

obj1.emit("event1");

obj1.addListener("event1", function() {
  console.log("L'objet obj1 a reçu un événement event1");
});
```

Figure 3–4
Déclenchement
d'un événement trop tôt,
pas de prise en compte



On voit ici que l'affichage du message a disparu, la fonction de callback n'est donc pas déclenchée.

Utiliser plusieurs fois la méthode `addListener()`

L'intérêt de la méthode `addListener()` est qu'elle peut s'utiliser à plusieurs reprises sur le même objet, permettant ainsi de cumuler les fonctions de traitement de l'événement.

Dans l'exemple qui suit, on utilise deux fois la méthode `addListener()` sur l'objet `obj1`. Un seul déclenchement de l'événement `event1` par `obj1.emit("event1")` provoque l'activation des deux fonctions de callback.

Utiliser deux fonctions de callback pour un même événement

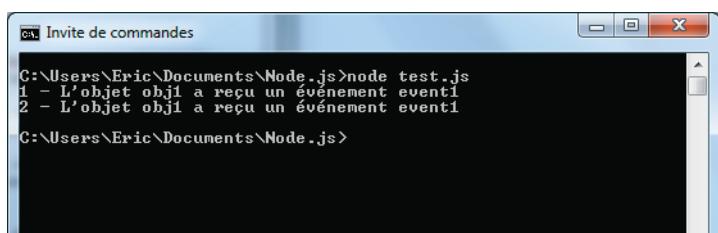
```
var events= require("events");
var obj1 = new events.EventEmitter();

obj1.addListener("event1", function() {
    console.log("1 - L'objet obj1 a reçu un événement event1");
});

obj1.addListener("event1", function() {
    console.log("2 - L'objet obj1 a reçu un événement event1");
});

obj1.emit("event1");
```

Figure 3-5
Utilisation de la méthode
addListener()



Lors de l'activation de l'événement `event1` par `obj1.emit("event1")`, les deux fonctions sont déclenchées dans l'ordre où elles ont été ajoutées par `addListener()`.

De plus, un second déclenchement de l'événement provoque un nouveau déclenchement des deux fonctions de callback.

Déclencher deux fois l'événement

```
var events= require("events");
var obj1 = new events.EventEmitter();

obj1.addListener("event1", function() {
    console.log("1 - L'objet obj1 a reçu un événement event1");
});

obj1.addListener("event1", function() {
    console.log("2 - L'objet obj1 a reçu un événement event1");
});

obj1.emit("event1");
obj1.emit("event1");
```

Figure 3–6
Émission et réception
d'événements en
les déclenchant deux fois

```
C:\Users\Eric\Documents>node test.js
1 - L'objet obj1 a reçu un événement event1
2 - L'objet obj1 a reçu un événement event1
1 - L'objet obj1 a reçu un événement event1
2 - L'objet obj1 a reçu un événement event1

C:\Users\Eric\Documents>
```

Supprimer un gestionnaire d'événements

On a vu comment ajouter une fonction de traitement de l'événement à l'aide de la méthode `addListener()`. Node a également défini une méthode inverse qui permet de supprimer une fonction de traitement ajoutée au préalable par `addListener()`. Il s'agit de la méthode `removeListener()`.

Tableau 3–2 Méthode de suppression d'un événement

Méthode	Signification
<code>obj.removeListener(event, listener)</code>	Supprimer, sur l'objet qui utilise la méthode, le gestionnaire d'événements représenté par la méthode <code>listener</code> , pour l'événement <code>event</code> . Si cet événement se produit, la fonction de callback <code>listener</code> ne sera plus déclenchée.

Dans l'exemple qui suit, nous attachons deux fonctions de traitement sur l'événement `event1` que nous déclençons, puis nous supprimons la seconde fonction de callback et déclençons une seconde fois l'événement. Lors du second déclenchement, seule la première fonction de callback est déclenchée vu que la seconde a été supprimée.

Suppression d'un gestionnaire d'événements

```
var events= require("events");
var obj1 = new events.EventEmitter();

obj1.addListener("event1", function() {
  console.log("1 - L'objet obj1 a reçu un événement event1");
});

obj1.addListener("event1", f = function() {
  console.log("2 - L'objet obj1 a reçu un événement event1");
});

obj1.emit("event1");

obj1.removeListener("event1", f);
obj1.emit("event1");
```

Pour supprimer un gestionnaire d'événements en particulier, il faut l'indiquer par sa référence en second paramètre de la méthode `removeListener()`. Pour accéder à cette référence, on lui donne un nom (ici, `f`) lors de sa création dans `addListener()`. On peut ensuite utiliser ce nom (donc la référence) dans la méthode `removeListener()`.

Figure 3–7

La gestion de l'événement a été supprimée.

```
C:\> node test.js
1 - L'objet obj1 a reçu un événement event1
2 - L'objet obj1 a reçu un événement event1
3 - L'objet obj1 a reçu un événement event1
C:\>
```

Supprimer tous les gestionnaires pour un événement

L'utilisation de la méthode `removeListener()` implique d'indiquer une référence à la fonction de traitement dans les arguments de la fonction. Pour supprimer tous les gestionnaires d'un événement particulier, ce qui est un cas très fréquent, Node a prévu une méthode permettant de ne pas avoir à supprimer les gestionnaires un par un comme précédemment. Il s'agit de la méthode `removeAllListeners(event)`, dans laquelle `event` est le nom de l'événement pour lequel on souhaite supprimer tous les gestionnaires.

Tableau 3–3 Méthode de suppression de tous les événements

Méthode	Signification
<code>obj.removeAllListeners([event])</code>	Supprimer, sur l'objet qui utilise la méthode, tous les gestionnaires pour l'événement indiqué. Si l'événement n'est pas indiqué, cela supprime tous les gestionnaires pour tous les événements se produisant sur l'objet <code>obj</code> .

Supprimer tous les gestionnaires d'un événement

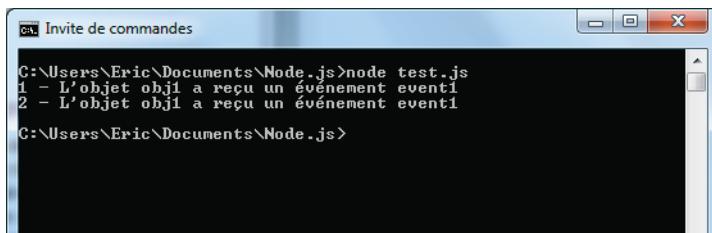
```
var events= require("events");
var obj1 = new events.EventEmitter();

obj1.addListener("event1", function() {
  console.log("1 - L'objet obj1 a reçu un événement event1");
});
```

```
obj1.addListener("event1", f = function() {  
    console.log("2 - L'objet obj1 a reçu un événement event1");  
});  
  
obj1.emit("event1");  
  
obj1.removeAllListeners("event1");  
obj1.emit("event1");
```

Figure 3–8

Suppression de tous les gestionnaires d'un événement particulier



Une fois les gestionnaires de l'événement `event1` supprimés, le déclenchement de l'événement `event1` n'est plus traité. À noter que l'on pourrait ajouter de nouveaux gestionnaires par la suite.

Pour supprimer tous les gestionnaires pour tous les événements inscrits depuis le lancement de l'application, il suffit d'appeler la méthode `removeAllListeners()` sans indiquer d'arguments.

Transmettre des paramètres lors de l'événement

En plus de pouvoir déclencher un événement au moyen de la méthode `emit(event)`, il est possible d'indiquer des paramètres qui seront transmis à la fonction de traitement lors de l'événement. On indique ces arguments dans la méthode `emit(event)`, à la suite du paramètre `event`. Par exemple, pour transmettre un nom et un prénom dans l'événement, on écrirait :

Transmettre un nom et un prénom lors du déclenchement de l'événement

```
obj1.emit("event1", "Sarrion", "Eric");
```

Si d'autres arguments sont à transmettre, il suffit de les indiquer à la suite, séparés par une virgule. Ces arguments sont ensuite reçus dans les paramètres de la fonction de traitement de l'événement, dans l'ordre où ils ont été envoyés. Pour les récupérer, on écrira donc dans la fonction de traitement :

Récupérer le nom et le prénom transmis dans la fonction de traitement de l'événement

```
obj1.addListener("event1", function(nom, prenom) {
    console.log("Nom = " + nom, " prenom = " + prenom);
});
```

Un exemple complet utilisant la transmission de paramètres dans l'événement pourrait être le suivant.

Transmettre des paramètres lors de l'événement

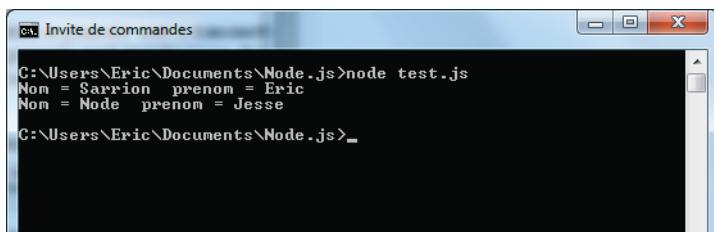
```
var events= require("events");
var obj1 = new events.EventEmitter();

obj1.addListener("event1", function(nom, prenom) {
    console.log("Nom = " + nom, " prenom = " + prenom);
});

obj1.emit("event1", "Sarrion", "Eric");
obj1.emit("event1", "Node", "Jesse");
```

Lors du premier déclenchement de l'événement, nous transmettons les arguments **"Sarrion"** et **"Eric"**, tandis que lors du second déclenchement, nous transmettons **"Node"** et **"Jesse"**. Ces noms et prénoms se retrouvent affichés dans la console du serveur.

Figure 3–9
Événement reçu
avec paramètres



Il est possible de transmettre des objets en paramètres, plutôt que de simples chaînes de caractères comme précédemment. Voici un exemple dans lequel on crée les objets **p1** et **p2** qui contiennent les champs **nom** et **prénom** qui seront affichés.

Utiliser un objet pour transmettre des paramètres lors de l'événement

```
var events= require("events");
var obj1 = new events.EventEmitter();

obj1.addListener("event1", function(p) {
    console.log("Nom = " + p.nom, " prenom = " + p.prenom);
```

```
});  
  
var p1 = { nom : "Sarrion", prenom : "Eric" };  
obj1.emit("event1", p1);  
var p2 = { nom : "Node", prenom : "Jesse" };  
obj1.emit("event1", p2);
```

La fonction de traitement reçoit un seul paramètre qui est l'objet `p` contenant le nom et le prénom, qui sont accessibles avec `p.nom` et `p.prenom`.

Le résultat affiché est identique au précédent.

Créer une classe dérivant de events.EventEmitter

Dans les sections précédentes de ce chapitre, nous avons appris à créer des objets émettant et recevant des événements. Il est également intéressant d'écrire une classe dont tous les objets pourront émettre et recevoir des événements. Cette fonctionnalité a, par exemple, été implémentée dans la classe `HttpServer`, dont les objets peuvent recevoir et émettre des événements. Les objets de la classe `HttpServer` sont des serveurs HTTP pouvant recevoir des connexions HTTP et afficher des résultats sur l'écran du navigateur. Cette fonctionnalité sera étudiée au chapitre 10, « Gestion des connexions HTTP ».

Implémentation d'une classe Server simple

En attendant de voir en détail comment fonctionne la classe `HttpServer`, il peut être intéressant de créer une classe similaire que l'on appellera simplement `Server`. Elle permettra de créer des objets qui émettront et recevront des événements `connexion`.

Créer une classe Server permettant d'émettre et de recevoir des événements connexion

```
var events = require("events");  
var util = require("util");  
  
function Server () {  
}  
  
util.inherits(Server, events.EventEmitter);  
  
var server = new Server();
```

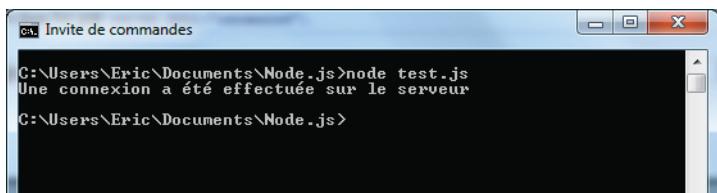
```
server.addListener("connexion", function() {
  console.log("Une connexion a été effectuée sur le serveur");
});

server.emit("connexion");
```

La classe `Server` est créée au moyen de la fonction `Server()`, vide pour l'instant. Un objet de cette classe est créé au moyen de `new Server()`. Afin que les objets de cette classe puissent bénéficier des fonctionnalités de `events.EventEmitter`, c'est-à-dire pouvoir recevoir et émettre des événements, il faut que la nouvelle classe `Server` hérite des fonctionnalités de `events.EventEmitter`. Pour cela, Node a créé dans le module `util` (voir chapitre 4) la méthode `util.inherits(newClass, oldClass)` qui permet de transférer les fonctionnalités de `oldClass` vers `newClass`. Donc l'instruction `util.inherits(Server, events.EventEmitter)` permet d'ajouter les fonctionnalités de `events.EventEmitter` à la classe `Server`. Les objets de la classe `Server` pourront ainsi émettre et recevoir des événements.

L'objet `server` de classe `Server`, créé au moyen de `new Server()`, peut donc recevoir des événements au moyen de `server.addListener()`. Nous affichons ici l'événement que nous appelons `connexion`, qui est déclenché par `server.emit("connexion")`.

Figure 3-10
Création d'une classe Server



L'événement `connexion` a bien été reçu sur le serveur.

Créer plusieurs serveurs associés à la classe Server

L'intérêt de créer des classes qui dérivent de la classe `events.EventEmitter` est de pouvoir créer plusieurs instances d'objets de la classe. Dans l'exemple précédent, nous avons créé un seul objet (appelé `server`), car nous avions un seul serveur. Supposons que l'on veuille en créer plusieurs. Un objet de classe `Server` sera par exemple caractérisé par son `port`, indiqué lors de la création de l'objet.

Nouvelle classe Server

```
function Server (port) {
  this.port = port;
  console.log("Création d'un serveur sur le port " + port);
}
```

Le port indiqué en paramètres est stocké dans l'attribut `port` de la classe (symbolisé par `this.port`). Un message est affiché dans la console afin de visualiser la création effective du serveur.

La nouvelle classe `Server` peut être utilisée de la façon suivante, en créant par exemple un serveur sur le port 3000 et un serveur sur le port 3001. Des connexions sur chacun de ces serveurs sont alors effectuées via l'envoi de l'événement `connexion`.

Création des serveurs sur les ports 3000 et 3001

```
var events = require("events");
var util = require("util");

function Server (port) {
    this.port = port;
    console.log("Création d'un serveur sur le port " + port);
}

util.inherits(Server, events.EventEmitter);

var server0 = new Server(3000);
var server1 = new Server(3001);

server0.addListener("connexion", function() {
    console.log("Une connexion a été effectuée sur le port " + this.port);
});

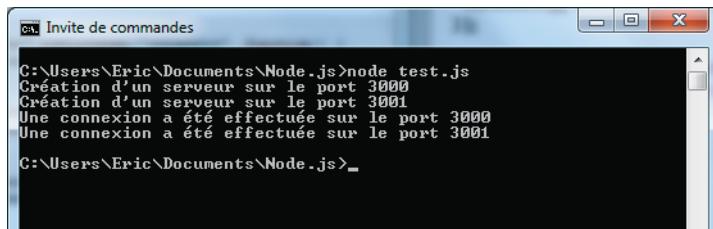
server1.addListener("connexion", function() {
    console.log("Une connexion a été effectuée sur le port " + this.port);
});

server0.emit("connexion");
server1.emit("connexion");
```

Les deux objets de classe `Server` sont respectivement nommés `server0` (pour le port 3000) et `server1` (pour le port 3001). Des gestionnaires d'événements sont ajoutés sur chacun de ces objets, permettant la réception des événements `connexion`, qui sont émis par la suite.

Figure 3–11

Connexions sur le serveur



Amélioration du programme

Une amélioration du précédent programme consiste à ne pas dupliquer la fonction de traitement de l'événement pour chacun des serveurs créés.

Amélioration du programme de création des serveurs

```
var events = require("events");
var util = require("util");

function Server (port) {
    this.port = port;
    console.log("Création d'un serveur sur le port " + port);
}

util.inherits(Server, events.EventEmitter);

var server0 = new Server(3000);
var server1 = new Server(3001);

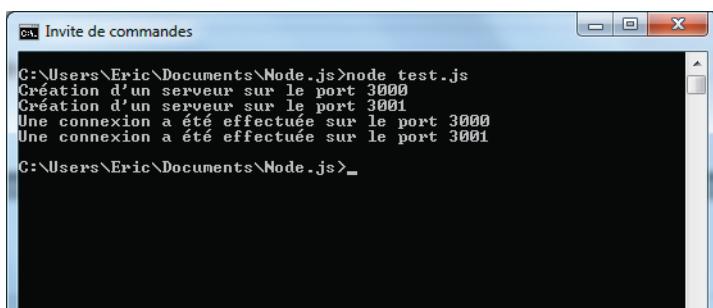
[server0, server1].forEach(function(server) {
    server.addListener("connexion", function() {
        console.log("Une connexion a été effectuée sur le port " + this.port);
    });
});

server0.emit("connexion");
server1.emit("connexion");
```

Plutôt que d'utiliser plusieurs fois la méthode `addListener()`, on l'utilise une seule fois mais dans une boucle de traitement. Node permet de parcourir le contenu d'un tableau au moyen de la méthode `forEach()` utilisée sur le tableau. Cette méthode prend en paramètre une fonction de callback appelée pour chaque élément du tableau.

Figure 3-12

Connexions sur le serveur
(résultat identique à celui
de la figure précédente)



Méthodes définies dans la classe events.EventEmitter

Nous avons vu et utilisé quelques-unes des méthodes de la classe `events.EventEmitter` :

- `addListener(event, listener)` ;
- `removeListener(event, listener)` ;
- `removeAllListeners(event)` ;
- `emit(event, arg1, arg2, ...)`.

D'autres méthodes existent, même si elles sont moins utilisées. La tableau 3-4 liste toutes les méthodes définies sur les objets de la classe `events.EventEmitter`.

Tableau 3-4 Méthodes définies dans la classe `events.EventEmitter`

Méthode	Signification
<code>obj.addListener(event, listener)</code>	Ajoute un gestionnaire d'événements pour l'événement <code>event</code> (chaîne de caractères). Le paramètre <code>listener</code> est une fonction de callback pouvant avoir 0 à <code>n</code> paramètres, selon que le déclenchement de l'événement par <code>emit(event)</code> transmet des arguments ou pas (voir la méthode <code>emit()</code> ci-dessous).
<code>obj.on(event, listener)</code>	Équivaut à <code>addListener(event, listener)</code> .
<code>obj.once(event, listener)</code>	Ajoute un gestionnaire d'événements comme <code>addListener()</code> , mais le supprime dès que le premier événement a été déclenché. Cela permet de déclencher la fonction de traitement indiquée dans le paramètre <code>listener</code> une fois seulement.
<code>obj.removeListener(event, listener)</code>	Supprime le gestionnaire d'événements géré par la fonction représentée par <code>listener</code> pour l'événement indiqué.
<code>obj.removeAllListeners(event)</code>	Supprime tous les gestionnaires d'événements mis en place pour l'événement <code>event</code> . Si <code>event</code> n'est pas indiqué, supprime tous les gestionnaires de tous les événements.
<code>obj.emit(event, arg1, arg2, ...)</code>	Émet l'événement <code>event</code> indiqué, en transmettant les éventuels arguments précisés dans la suite des arguments. Ces arguments pourront être récupérés dans la fonction de traitement de l'événement.
<code>obj.setMaxListeners(n)</code>	Indique un nombre maximal de gestionnaires d'événements pouvant être positionnés sur un objet particulier. Par défaut, on peut en positionner au maximum 10 sur un même objet. Indiquer 0 pour ne plus avoir de limite sur cet objet.
<code>obj.listeners(event)</code>	Retourne un tableau contenant les références vers les gestionnaires d'événements associés à <code>event</code> et positionnés pour cet objet (celui qui utilise la méthode <code>listeners()</code>).

Parmi ces méthodes, la méthode `once()` retient notre attention. Voyons comment l'utiliser en analysant ses caractéristiques.

La méthode `on()` équivaut à la méthode `addListener()`. Étant donné qu'elle est plus courte à écrire, elle est en fait plus utilisée que la méthode d'origine.

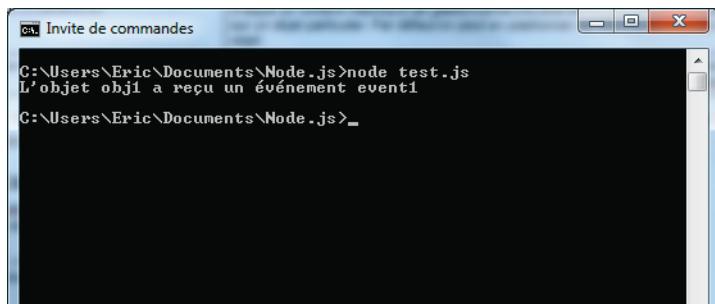
Utiliser la méthode `once()`

```
var events = require("events");
var obj1 = new events.EventEmitter();

obj1.once("event1", function() {
    console.log("L'objet obj1 a reçu un événement event1");
});

obj1.emit("event1"); // déclencher une première fois
obj1.emit("event1"); // puis une seconde fois
```

Figure 3–13
Gestion pour la réception
d'un seul événement



Bien que l'événement soit déclenché deux fois sur l'objet `obj1`, seul le premier déclenchement est pris en compte du fait de l'utilisation de la méthode `once()`.

4

Méthodes utilitaires

Node fournit en standard plusieurs modules dits « utilitaires », qui facilitent la programmation (manipulation des chaînes de caractères et des buffers d'octets, gestion des URL, etc.). Ce sont ces différents modules que nous vous proposons d'étudier dans ce chapitre.

Gestion de l'affichage à l'écran : objet console

Nous avons souvent utilisé la méthode `console.log()` dans les chapitres précédents, qui permet d'afficher un texte à l'écran pendant l'exécution du programme Node. L'objet `console` définit la méthode `log()` mais également d'autres méthodes que nous décrivons dans le tableau 4-1.

Tableau 4-1 Méthodes d'affichage à l'aide de l'objet console

Méthode	Signification
<code>console.log([data], [...])</code>	Méthode d'affichage sur l'écran, représenté par <code>process.stdin</code> (voir la gestion des streams au chapitre 5). La chaîne <code>data</code> peut contenir des caractères de formatage comme ceux décrits dans <code>util.format()</code> : <ul style="list-style-type: none">- une chaîne de caractères (utiliser "<code>%s</code>") ;- un entier (utiliser "<code>%d</code>") ;- un objet (utiliser "<code>%j</code>"). Les caractères de formatage éventuels seront remplacés par les valeurs situées dans la suite des paramètres.

Tableau 4-1 Méthodes d'affichage à l'aide de l'objet console (suite)

Méthode	Signification
<code>console.info([data], [...])</code>	Équivaut à <code>console.log()</code> .
<code>console.error([data], [...])</code>	Équivaut à <code>console.log()</code> , mais affiche sur <code>process.stderr</code> .
<code>console.warn([data], [...])</code>	Équivaut à <code>console.error()</code> .
<code>console.dir(obj)</code>	Affiche le contenu de l'objet <code>obj</code> .
<code>console.time(label)</code>	Permet de positionner un label afin d'indiquer le début d'une séquence dont on veut mesurer la durée en millisecondes. Les instructions situées entre cette instruction et celle indiquant la fin de la mesure seront chronométrées et la durée totale affichée.
<code>console.timeEnd(label)</code>	Permet de positionner un label de fin de la mesure de temps. La durée totale d'exécution des instructions encapsulées par ce label sera affichée à la suite du label.

Le paramètre `label` utilisé dans les deux instructions est le même. S'il est différent, l'instruction de fin ne peut plus être rattachée à celle de début, et provoque alors une erreur d'exécution.

Utiliser `console.log()`

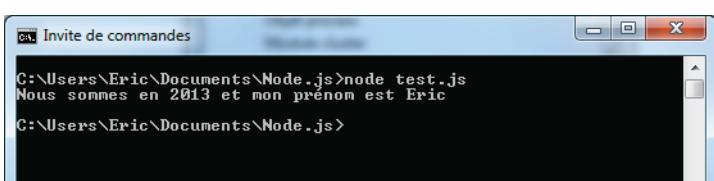
La méthode `console.log()` permet d'afficher des chaînes de caractères, mais également de les formater pour l'affichage.

Utiliser `console.log` avec le formatage de caractères

```
console.log("Nous sommes en %d et mon prénom est %s", 2013, "Eric");
```

Nous utilisons ici "`%d`" pour afficher une valeur entière et "`%s`" pour afficher une chaîne de caractères.

Figure 4-1
Affichage d'un texte
dans la console



L'année `2013` et le prénom `"Eric"` sont venus remplacer les caractères de formatage. Remarquons que si d'autres arguments suivent lors de l'appel, ils viennent compléter les caractères déjà affichés à l'écran. Par exemple :

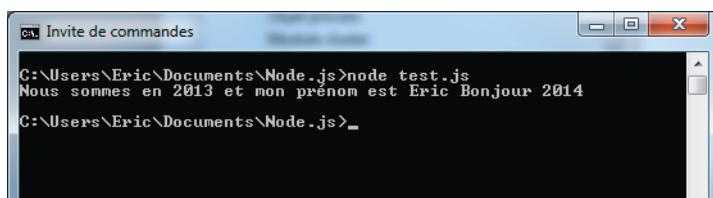
Utiliser plusieurs arguments dans console.log()

```
console.log("Nous sommes en %d et mon prénom est %s", 2013, "Eric", "Bonjour", 2014);
```

Seuls deux caractères de formatage ("`%d`" et "`%s`") sont présents dans l'instruction, mais quatre arguments sont effectivement transmis lors de l'appel. Les arguments qui ne correspondent pas aux caractères de formatage sont insérés à la suite des caractères déjà affichés, séparés par une espace.

Figure 4–2

Affichage d'un texte avec formatage



Utiliser console.time(label) et console.timeEnd(label)

Les deux instructions `console.time(label)` et `console.timeEnd(label)` sont complémentaires et sont employées pour mesurer la durée d'un traitement.

- La méthode `console.time(label)` s'utilise en début de traitement pour indiquer le début de celui-ci.
- La méthode `console.timeEnd(label)` s'utilise en fin de traitement pour indiquer la fin de celui-ci. Toutes les instructions exécutées entre ces deux instructions ont leur durée d'exécution comptabilisée et affichée par l'instruction de fin.

Afficher la durée d'un traitement

```
console.log("Début de la boucle");
console.time("Durée de la boucle");
for (var i = 0; i < 100000000; i++) {
    for (var j = 0; j < 10; j++) {
        ;
    }
}
console.timeEnd("Durée de la boucle");
console.log("Fin de la boucle");
```

Nous effectuons ici deux boucles imbriquées, de façon à provoquer un traitement d'une certaine durée. Nous affichons un message en début de traitement et un message à la fin de celui-ci. Afin de mesurer sa durée, nous insérons l'instruction `console.time(label)` au début et l'instruction `console.timeEnd(label)` à la fin, en utilisant le même label aux deux endroits.

Figure 4–3
Utilisation de `console.time()`

```
C:\Users\Eric\Documents\Node.js>node test.js
Début de la boucle
Durée de la boucle: 1530ms
Fin de la boucle
C:\Users\Eric\Documents\Node.js>
```

Lors de l'instruction `console.timeEnd()`, le label est affiché à l'écran, suivi de la durée d'exécution des instructions entre l'instruction de début et l'instruction de fin.

Utiliser `console.dir(obj)`

La méthode `console.dir()` permet d'afficher les propriétés d'un objet, plus facilement qu'une boucle traditionnelle pourrait le faire. Utilisons cette méthode pour afficher, par exemple, le contenu de l'objet `console`.

Afficher le contenu de l'objet `console`

```
console.dir(console);
```

Figure 4–4
Affichage de l'objet `console` avec `console.dir()`

```
C:\Users\Eric\Documents\Node.js>node test.js
{
  log: [Function],
  info: [Function],
  warn: [Function],
  error: [Function],
  dir: [Function],
  time: [Function],
  timeEnd: [Function],
  trace: [Function],
  assert: [Function],
  Console: [Function: Console]
}
C:\Users\Eric\Documents\Node.js>
```

Le nom de la propriété s'affiche, sa valeur est indiquée à la suite.

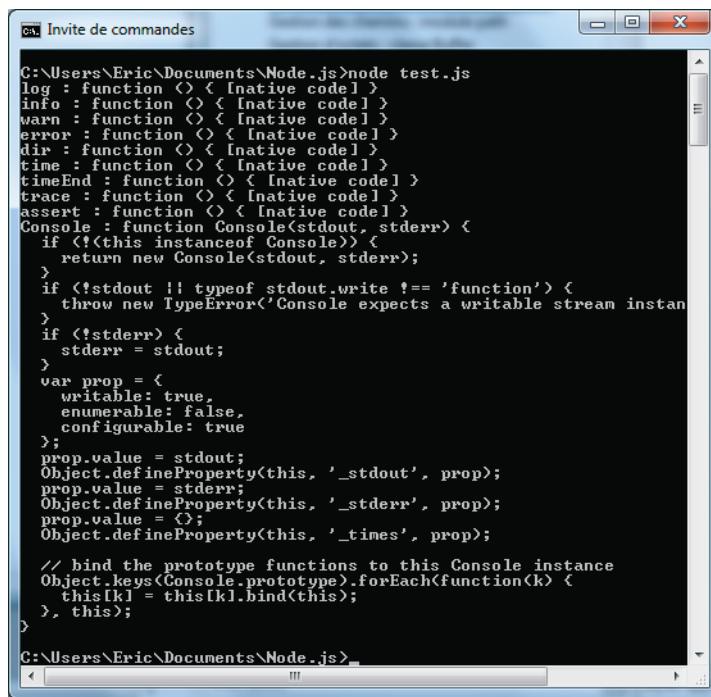
Si on utilisait une boucle traditionnelle parcourant les propriétés de l'objet, on écrirait plutôt ceci.

Utiliser une boucle pour afficher les propriétés de l'objet `console`

```
for (var prop in console)
  console.log("%s : %s", prop, console[prop]);
```

Figure 4–5

Contenu de l'objet console



The screenshot shows a Windows Command Prompt window with the title 'Invite de commandes'. The command entered is 'node test.js'. The output displays the source code of the 'Console' constructor function from Node.js. This code defines methods like log, info, warn, error, dir, time, trace, assert, and times, and sets up properties for _stdout and _stderr.

```
C:\Users\Eric\Documents\Node.js>node test.js
log : function () { [native code] }
info : function () { [native code] }
warn : function () { [native code] }
error : function () { [native code] }
dir : function () { [native code] }
time : function () { [native code] }
timeEnd : function () { [native code] }
trace : function () { [native code] }
assert : function () { [native code] }
Console : function Console(_stdout, _stderr) {
  if (!(this instanceof Console)) {
    return new Console(_stdout, _stderr);
  }
  if (!_stdout || typeof _stdout.write !== 'function') {
    throw new TypeError('Console expects a writable stream instance');
  }
  if (!_stderr) {
    _stderr = _stdout;
  }
  var prop = {
    writable: true,
    enumerable: false,
    configurable: true
  };
  prop.value = _stdout;
  Object.defineProperty(this, '_stdout', prop);
  prop.value = _stderr;
  Object.defineProperty(this, '_stderr', prop);
  prop.value = {};
  Object.defineProperty(this, '_times', prop);

  // bind the prototype functions to this Console instance
  Object.keys(Console.prototype).forEach(function(k) {
    this[k] = this[k].bind(this);
  }, this);
}
C:\Users\Eric\Documents\Node.js>
```

Nous verrons dans la section suivante que le module `util` comporte une méthode `util.inspect()` qui permet d'inspecter un objet selon une profondeur donnée.

Fonctions générales : module `util`

Le module `util` contient des méthodes de mise en forme de chaînes de caractères. On trouve également un système d'héritage de classes, et des méthodes booléennes permettant de tester le type de certains objets.

Formatage de chaîne de caractères

Il est possible de mettre en forme des chaînes de caractères selon un format indiqué, comme on pouvait le faire dans un langage tel que le langage C. La méthode `console.log()` vue précédemment utilise ce mécanisme en interne.

Tableau 4–2 Méthode de formatage de chaîne de caractères

Méthode	Signification
<code>util.format(format, [...])</code>	Retourne une chaîne de caractères selon le format indiqué par la chaîne <code>format</code> . Les arguments qui suivent (indiqués par [...] pour spécifier que leur nombre peut varier) sont remplacés dans la chaîne <code>format</code> en tenant compte du format à cet emplacement. Le format peut spécifier : - une chaîne de caractères (utiliser "%s") ; - un entier (utiliser "%d") ; - un objet (utiliser "%j").

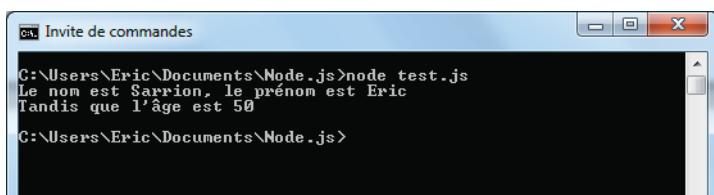
Utilisation de util.format()

```
var util = require("util");
var txt = util.format("Le nom est %s, le prénom est %s\n", "Sarrion", "Eric");
txt += util.format("Tandis que l'âge est %d", 50);

console.log(txt);
```

Les endroits où nous avons indiqué `%s`, `%d` ou `%j` dans le format seront remplacés par les arguments qui suivent, dans le même ordre. De plus, `util.format()` retournant une chaîne de caractères, elle peut être concaténée avec une autre chaîne comme ici.

Figure 4–6
Formatage de chaînes
avec `util.format()`



Une variante du précédent programme est celui où les valeurs à remplacer sont les champs d'un objet précédemment créé.

Utiliser les champs d'un objet pour effectuer le formatage

```
var util = require("util");
var obj = { nom : "Sarrion", prenom : "Eric", age : 50 };
var txt = util.format("Le nom est %s, le prénom est %s\n", obj.nom, obj.prenom);
txt += util.format("Tandis que l'âge est %d", obj.age);

console.log(txt);
```

Nous avons ici créé un objet `obj` dans lequel nous définissons les champs `nom`, `prenom` et `age` initialisés aux valeurs indiquées. Puis ce sont les champs de cet objet qui sont remplacés par leur valeur dans la chaîne de formatage. Le résultat affiché est le même que précédemment.

En utilisant `%j` pour afficher l'objet complet sous forme JSON, on peut aussi écrire :

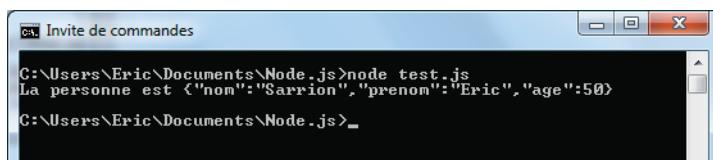
Afficher l'objet sous forme JSON

```
var util = require("util");
var obj = { nom : "Sarrion", prenom : "Eric", age : 50 };
var txt = util.format("La personne est %j", obj);

console.log(txt);
```

Figure 4–7

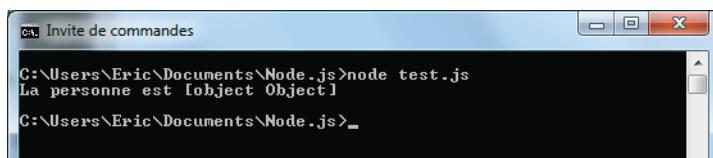
Affichage au format JSON



Si on utilise `%s` au lieu de `%j` pour afficher l'objet, on obtient à la place :

Figure 4–8

Affichage sous forme d'objet



En effet, l'objet devant s'afficher maintenant sous forme de chaîne de caractères (à cause du `%s`), c'est la méthode `toString()` définie en standard dans la classe `Object` de JavaScript qui est utilisée ici. Pour modifier son comportement, il est possible de redéfinir la méthode `toString()` sur l'objet `obj`.

Redéfinir la méthode `toString()` sur l'objet `obj`

```
var util = require("util");
var obj = { nom : "Sarrion", prenom : "Eric", age : 50 };

obj.toString = function() {
    return util.format(": %j", this);
}

var txt = util.format("La personne est %s", obj);

console.log(txt);
```

La chaîne est toujours formatée au moyen de `%s`, mais c'est maintenant la méthode `toString()` définie sur l'objet `obj` qui est appelée. Cette méthode retourne une chaîne de caractères devant laquelle on ajoute le signe `:`, puis l'objet au format JSON.

Figure 4–9
Utilisation de `util.format()`

```
C:\Users\Eric\Documents\Node.js>node test.js
La personne est : {"nom":"Sarrion", "prenom":"Eric", "age":50}
C:\Users\Eric\Documents\Node.js>
```

Inspecter des objets avec `util.inspect()`

La méthode `util.inspect(obj, options)` permet de retourner une chaîne de caractères correspondant au contenu de l'objet `obj`. Par défaut, l'objet est affiché sur une profondeur de deux niveaux au maximum, mais cela peut être modifié grâce à la propriété `depth` dans le paramètre `options`.

Tableau 4–3 Méthode d'inspection d'objet

Méthode	Signification
<code>util.inspect(obj, options)</code>	Retourne une chaîne correspondant à l'objet <code>obj</code> , sur une profondeur de deux niveaux au maximum. Le paramètre <code>options</code> est un objet ayant la propriété <code>depth</code> suivante : - <code>depth</code> : profondeur d'analyse de l'objet (dans le cas où l'objet contient des propriétés qui sont des objets). Par défaut, 2. Indiquer 0 pour afficher seulement les propriétés directes de l'objet <code>obj</code> .

Remarquons que `util.inspect()` ne fait que retourner une chaîne de caractères, elle devra ensuite être affichée au moyen de `console.log()`, par exemple.

Utiliser `util.inspect()` pour afficher le contenu d'un objet

```
var util = require("util");
var obj = {
  nom : "Sarrion",
  prenom : "Eric",
  adresse : {
    ville : "Paris",
    pays : "France"
  }
}
console.log("\nAvec une profondeur de 2 par défaut");
console.log(util.inspect(obj));
console.log("\nAvec une profondeur de 0");
console.log(util.inspect(obj, { depth : 0 }));
```

Figure 4-10

Affichage du contenu d'un objet

```
C:\Users\Eric\Documents\Node.js>node test.js
Avec une profondeur de 2 par défaut
{
  nom: 'Sarrion',
  prenom: 'Eric',
  adresse: {
    ville: 'Paris',
    pays: 'France'
  }
}
Avec une profondeur de 0
{
  nom: 'Sarrion',
  prenom: 'Eric',
  adresse: [Object]
}
C:\Users\Eric\Documents\Node.js>
```

Selon la profondeur désirée, l'objet est inspecté avec plus ou moins de précision.

Héritage de classes

Il est possible de permettre à une classe d'hériter d'une autre classe. JavaScript ne le permettant pas de façon simple (il faut passer par la propriété `prototype` de JavaScript), Node a implémenté cette possibilité dans le module `util`, au moyen de la méthode `util.inherits()`.

Tableau 4-4 Méthode permettant l'héritage

Méthode	Signification
<code>util.inherits(newClass, oldClass)</code>	Permet à la classe <code>newClass</code> d'hériter de la classe <code>oldClass</code> . Les propriétés et méthodes de <code>oldClass</code> sont ajoutées à celles déjà présentes dans <code>newClass</code> .

Un exemple classique de démonstration de ces possibilités est celui que nous avons utilisé dans le précédent chapitre, en créant une classe dérivant de `events.EventEmitter`. Grâce à l'héritage mis en place, la nouvelle classe `Server` pouvait émettre et recevoir des événements.

Nous reproduisons ici le code que nous avions écrit, afin d'illustrer l'héritage.

Créer une classe Server qui hérite de events.EventEmitter

```
var events = require("events");
var util = require("util");

function Server () {
}

util.inherits(Server, events.EventEmitter);

var server = new Server();
```

```

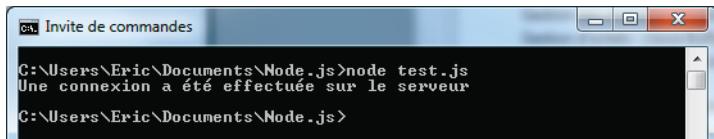
server.on("connexion", function() {
  console.log("Une connexion a été effectuée sur le serveur");
});

server.emit("connexion");

```

La classe `Server` peut maintenant émettre et recevoir des événements, comme le montre la figure 4-11.

Figure 4-11
Création d'une classe Server

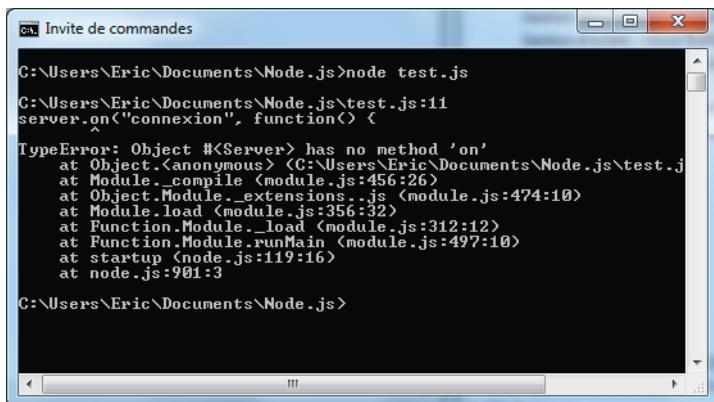


Si on supprime l'héritage dans l'exemple précédent, les méthodes `on()` et `emit()` sont inconnues sur l'objet `server` de classe `Server`.

Suppression de l'héritage (mise en commentaires)

```
// util.inherits(Server, events.EventEmitter);
```

Figure 4-12
Erreur lors de la création de la classe Server



Méthodes booléennes

Il existe enfin quelques méthodes du module `util` permettant d'effectuer des tests afin de déterminer la classe d'un objet. On peut ainsi tester si l'objet est de classe `Array`, de classe `RegExp`, de classe `Date` ou encore de classe `Error`.

Tableau 4–5 Méthodes booléennes testant le type d'un objet

Méthode	Signification
<code>util.isArray(object)</code>	Retourne <code>true</code> si l'objet est de classe <code>Array</code> .
<code>util.isRegExp(object)</code>	Retourne <code>true</code> si l'objet est de classe <code>RegExp</code> .
<code>util.isDate(object)</code>	Retourne <code>true</code> si l'objet est de classe <code>Date</code> .
<code>util.isError(object)</code>	Retourne <code>true</code> si l'objet est de classe <code>Error</code> .

Test de classes

```
var util = require("util");
var tab = [1, 2, 3];
console.log(util.isArray(tab));    // true

var d = new Date();
console.log(util.isDate(d));      // true

var e = new Error();
console.log(util.isError(e));     // true
```

Gestion des URL : module url

Un serveur, tel que Node nous le propose, doit pouvoir manipuler facilement les URL qui lui sont transmises par les utilisateurs. Pour cela, Node offre la possibilité de gérer les différents composants de l'URL au moyen du module `url`. Trois méthodes composent ce module.

Tableau 4–6 Méthodes du module url

Méthode	Signification
<code>url.parse(urlStr, [parseQueryString])</code>	Retourne un objet (voir tableau 4–7) contenant les principaux composants de l'URL transmise dans <code>urlStr</code> . Si le paramètre <code>parseQueryString</code> vaut <code>true</code> , analyse également la partie de l'URL située à droite du "?" (correspondant à la partie <code>query</code>).
<code>url.format(urlObj)</code>	Construit une URL à partir de l'objet (voir tableau 4–7) indiqué en paramètres.
<code>url.resolve(from, to)</code>	Construit une URL (relative ou absolue) selon les paramètres <code>from</code> et <code>to</code> qui lui sont transmis. Le paramètre <code>from</code> correspond au début de l'URL, tandis que le paramètre <code>to</code> vient s'ajouter à celui-ci. Le résultat, l'URL finale, est une combinaison des deux parties d'URL.

Les deux méthodes `url.parse()` et `url.format()` utilisent un objet contenant les composants de l'URL. La description de cet objet est indiquée dans le tableau 4–7.

Tableau 4–7 Propriétés de l'objet retourné par `url.parse(urlStr, parseQueryString)` et utilisé par `url.format(urlObj)`

Propriété	Signification
<code>href</code>	URL d'origine qui a été transmise pour analyse (correspond au paramètre <code>urlStr</code>).
<code>protocol</code>	Protocole utilisé, incluant ":" en fin (par exemple, " <code>http:</code> ").
<code>host</code>	Nom du <code>host</code> , incluant le port éventuel (par exemple, <code>localhost:3000</code>).
<code>hostname</code>	Nom du <code>host</code> , sans l'indication du port éventuel (par exemple, <code>localhost</code>).
<code>port</code>	Port utilisé (par exemple, 3000).
<code>pathname</code>	Partie qui suit le <code>host</code> (en incluant le "/"), jusqu'au "?" éventuel exclu (par exemple, <code>/docs/index.php</code>).
<code>search</code>	Partie qui suit le "?" (inclus) jusqu'à la fin (par exemple, <code>?name=value&id=p3</code>).
<code>path</code>	Concaténation de <code>pathname + search</code> (par exemple, <code>/docs/index.php?name=value&id=p3</code>).
<code>query</code>	Équivaut à <code>search</code> , mais sans le "?" du début (par exemple, <code>"name=value&id=p3"</code>). Toutefois, si le paramètre <code>parseQueryString</code> vaut <code>true</code> , <code>query</code> ne vaut plus une chaîne de caractères mais plutôt un objet décomposé au format JSON.
<code>hash</code>	Partie qui suit l'éventuel "#" inclus (par exemple, <code>#label1</code>).

Voyons maintenant comment utiliser ces méthodes dans nos programmes.

Utiliser `url.parse()`

La méthode `url.parse(urlStr, parseQueryString)` permet de décomposer une URL en divers composants tels que ceux décrits dans le tableau 4-7.

Grâce au programme suivant, on récupère chaque composant des URL transmises dans un tableau, que l'on affiche sur l'écran.

Décomposer les parties d'une URL à l'aide de `url.parse()`

```
var url = require("url");
var r1 = url.parse("http://ericsarrion.fr:80/index.php?toto=titi#label1");
var r2 = url.parse("http://ericsarrion.fr/index.php");
[r1, r2].forEach(function(r) {
  console.log("\n");
  console.log("href = " + r.href);
  console.log("protocol = " + r.protocol);
```

```

        console.log("host = " + r.host);
        console.log("hostname = " + r.hostname);
        console.log("port = " + r.port);
        console.log("pathname = " + r.pathname);
        console.log("search = " + r.search);
        console.log("path = " + r.path);
        console.log("query = " + r.query);
        console.log("hash = " + r.hash);
    });
}

```

Figure 4–13

Décomposition du contenu de l'URL

```

C:\Users\Eric\Documents\Node.js>node test.js

href = http://ericsarrion.fr:80/index.php?toto=titi#label1
protocol = http:
host = ericsarrion.fr:80
hostname = ericsarrion.fr
port = 80
pathname = /index.php
search = ?toto=titi
path = /index.php?toto=titi
query = toto=titi
hash = #label1

href = http://ericsarrion.fr/index.php
protocol = http:
host = ericsarrion.fr
hostname = ericsarrion.fr
port = null
pathname = /index.php
search = null
path = /index.php
query = null
hash = null

```

Si un élément n'est pas renseigné (par exemple, le port, comme ici), il est positionné à la valeur `null` dans l'objet retourné par `url.parse()`.

Utiliser url.resolve()

La méthode `url.resolve(from, to)` permet de fabriquer une URL à partir des paramètres `from` et `to`. Le plus simple pour comprendre son fonctionnement est de voir le résultat obtenu grâce à quelques exemples.

Afficher les URL produites par url.resolve()

```

var url = require("url");
console.log("%s => \n%s\n", "url.resolve('/one/two/three', 'four')",
           url.resolve('/one/two/three', 'four'));
console.log("%s => \n%s\n", "url.resolve('/one/two/three', '/four')",
           url.resolve('/one/two/three', '/four'));

```

```

console.log("%s => \n%s\n", "url.resolve('http://example.com', '/one')",
           url.resolve('http://example.com', '/one'));
console.log("%s => \n%s\n", "url.resolve('http://example.com/one''', '/two')",
           url.resolve('http://example.com/one', '/two'));
console.log("%s => \n%s\n", "url.resolve('http://example.com/one/'', 'two')",
           url.resolve('http://example.com/one/', 'two'));
console.log("%s => \n%s\n", "url.resolve('http://example.com/one''', 'two')",
           url.resolve('http://example.com/one', 'two'));
console.log("%s => \n%s\n", "url.resolve('http://example.com/one/'', '/two')",
           url.resolve('http://example.com/one/', 'two'));

```

Ce programme se contente d'effectuer quelques appels à la méthode `url.resolve()`, en affichant l'appel suivi du résultat obtenu.

Figure 4-14

Création d'URL

```

C:\Users\Eric\Documents\Node.js>node test.js
url.resolve('/one/two/three', 'four') =>
/one/two/four

url.resolve('/one/two/three', '/four') =>
/four

url.resolve('http://example.com', '/one') =>
http://example.com/one

url.resolve('http://example.com/one', '/two') =>
http://example.com/two

url.resolve('http://example.com/one''', 'two') =>
http://example.com/one/two

url.resolve('http://example.com/one''', '/two') =>
http://example.com/two

url.resolve('http://example.com/one''', '/two') =>
http://example.com/one/two

C:\Users\Eric\Documents\Node.js>_

```

On voit que selon que le paramètre `from` comporte ou non le caractère "/" à la fin, la concaténation des deux paramètres `from` et `to` ne produit pas le même résultat final. En revanche, la présence du caractère "/" au début du paramètre `to` n'a pas d'incidence sur le résultat final.

Gestion des requêtes : module `querystring`

Le module `querystring` permet de manipuler la partie de l'URL correspondant à la requête (*query* en anglais), c'est-à-dire celle située à la suite du "?". Ce module comporte la méthode `querystring.stringify(obj)` qui permet de créer la chaîne de la requête à partir de l'objet fourni, et la méthode `querystring.parse(str)` qui analyse la chaîne transmise et retourne un objet lui correspondant.

Tableau 4–8 Méthodes du module querystring

Méthode	Signification
<code>querystring.stringify(obj, [sep], [eq])</code>	Retourne la chaîne <code>query</code> correspondant à l'objet transmis, en utilisant les caractères optionnels <code>sep</code> comme séparateur (par défaut, " <code>&</code> ") et <code>eq</code> comme signe d'égalité (par défaut, " <code>=</code> ").
<code>querystring.parse(str, [sep], [eq])</code>	Retourne un objet décomposé à partir de la chaîne correspondant à <code>str</code> . C'est l'opération inverse de <code>querystring.stringify()</code> . Le paramètre <code>sep</code> indique le caractère de séparation (par défaut, " <code>&</code> "), tandis que le paramètre <code>eq</code> indique le caractère d'égalité (par défaut, " <code>=</code> ").

Voici un programme montrant l'utilisation de ces deux méthodes.

Utilisation des méthodes du module querystring

```
var querystring = require("querystring");
var query = querystring.stringify( {
    attr1 : "value1",
    attr2 : "value2"
});

console.log("Utilisation de querystring.stringify()");
console.log("La query vaut : " + query);
console.log("\n");

console.log("Utilisation de querystring.parse()");
var obj = querystring.parse(query);
console.dir(obj);
```

Figure 4–15
Méthodes `stringify()` et `parse()`
du module `querystring`

```
C:\> Invité de commandes
C:\Users\Eric\Documents\Node.js>node test.js
Utilisation de querystring.stringify()
La query vaut : attr1=value1&attr2=value2

Utilisation de querystring.parse()
{ attr1: 'value1', attr2: 'value2' }

C:\Users\Eric\Documents\Node.js>
```

Gestion des chemins : module path

Le module `path` permet de manipuler les noms de fichiers et les chemins associés (c'est-à-dire les répertoires). Les méthodes de ce module ne font aucune vérification de l'existence ou non des chemins ou fichiers indiqués. Elles ne font que des manipulations de chaînes de caractères.

Tableau 4–9 Méthodes du module path

Méthode	Signification
<code>path.normalize(p)</code>	Retourne une chaîne de caractères correspondant au nom de fichier ou de répertoire transmis, mais épurée des caractères ".." et "... éventuels. De plus, la chaîne renournée est mise en forme selon les conventions du système d'exploitation (par exemple, on remplace "/" par "\\" sous Windows).
<code>path.join([path1], [path2], [...])</code>	Construit un chemin à partir des différents chemins indiqués en paramètres, dans l'ordre où ils sont indiqués, en ajoutant les séparateurs si nécessaire. Le caractère séparateur pour le système est indiqué par <code>path.sep</code> ("\\" sous Windows, "/" sous Unix).
<code>path.dirname(p)</code>	Retourne le chemin associé à <code>p</code> (<code>p</code> étant un chemin ou un fichier).
<code>path.basename(p)</code>	Retourne le dernier composant du chemin, soit le nom du fichier avec extension, soit le nom du dernier répertoire.
<code>path.extname(p)</code>	Retourne l'extension du fichier précédée de ".", sinon une chaîne vide.
<code>path.sep</code>	Caractère séparateur des répertoires pour le système d'exploitation ("\\" sous Windows, "/" sous Unix).

Voici un programme permettant de manipuler ces différentes méthodes.

Utilisation des méthodes du module path

```
var path = require("path");
console.log("\n%s \n=> %s",
    'path.normalize("/un/deux//trois/quatre/cinq/../")',
    path.normalize("/un/deux//trois/quatre/cinq/../"));
console.log("\n%s \n=> %s",
    'path.normalize("/un/deux//trois/quatre/cinq/../index.php")',
    path.normalize("/un/deux//trois/quatre/cinq/../index.php"));

console.log("\n%s \n=> %s",
    'path.join("/un", "deux", "trois/quatre", "cinq", "..")',
    path.join("/un", "deux", "trois/quatre", "cinq", ".."));

console.log("\n%s \n=> %s",
    'path.join("/un", "deux", "trois/quatre/", "/cinq", "..")',
    path.join("/un", "deux", "trois/quatre/", "/cinq", ".."));

console.log("\n%s \n=> %s",
    'path.dirname("/un/deux/trois/index.php")',
    path.dirname("/un/deux/trois/index.php"));

console.log("\n%s \n=> %s",
    'path.basename("/un/deux/trois/index.php")',
    path.basename("/un/deux/trois/index.php"));
```

```
console.log("\n%s \n=> %s",
    'path.basename("/un/deux/trois/")',
    path.basename("/un/deux/trois/"));

console.log("\n%s \n=> %s",
    'path.extname("/un/deux/trois/index.php")',
    path.extname("/un/deux/trois/index.php"));

console.log("\n%s \n=> %s",
    'path.extname("/un/deux/trois/")',
    path.extname("/un/deux/trois/"));
```

Figure 4–16
Création de chemins

```
C:\Users\Eric\Documents\Node.js>node test.js
path.normalize("/un/deux//trois/quatre/cinq/..")
=> '\un\deux\trois\quatre'

path.normalize("/un/deux//trois/quatre/cinq/..index.php")
=> '\un\deux\trois\quatre\index.php'

path.join("/un", "deux", "trois/quatre", "cinq", "..")
=> '\un\deux\trois\quatre'

path.join("/un", "deux", "trois/quatre/", "cinq", "..")
=> '\un\deux\trois\quatre'

path.dirname('/un/deux/trois/index.php')
=> '/un/deux/trois'

path.basename('/un/deux/trois/index.php')
=> index.php

path.basename('/un/deux/trois/')
=> trois

path.extname('/un/deux/trois/index.php')
=> .php

path.extname('/un/deux/trois/')
=>

C:\Users\Eric\Documents\Node.js>
```

La gestion des fichiers sur disque sera étudiée au chapitre 6, « Gestion des fichiers ».

Gestion d'octets : classe Buffer

JavaScript permet de gérer facilement les chaînes de caractères, mais pas les buffers d'octets. En tant que serveur, Node doit permettre de gérer les flux d'octets reçus ou envoyés aux utilisateurs. Il a donc créé une classe **Buffer** permettant de faire cela.

Créer un objet de classe Buffer

La classe `Buffer` est accessible directement, sans insérer un module particulier. On crée un objet de cette classe au moyen de `new Buffer(string)` ou `new Buffer(size)`. Dans les deux cas, la taille du buffer sera fixe et correspondra à la longueur de la chaîne ou à la taille indiquée, et ne pourra pas être modifiée.

Créer un objet Buffer à partir d'une chaîne de caractères

```
var buf1 = new Buffer("Bonjour Eric");
console.log(buf1);
console.log("Taille du buffer = " + buf1.length);
console.log("\n");

var buf2 = new Buffer("Bonjour éric");
console.log(buf2);
console.log("Taille du buffer = " + buf2.length);
```

Nous créons ici deux buffers à partir de deux chaînes de caractères, la seconde contenant un caractère accentué. Puis, nous affichons les buffers correspondants, ainsi que leur taille en octets.

Figure 4-17

Affichage de buffers d'octets

```
C:\Users\Eric\Documents\Node.js>node test.js
<Buffer 42 6f 6e 6a 6f 75 72 20 45 72 69 63>
Taille du buffer = 12

<Buffer 42 6f 6e 6a 6f 75 72 20 c3 a9 72 69 63>
Taille du buffer = 13
C:\Users\Eric\Documents\Node.js>
```

Le second buffer, bien qu'il ait le même nombre de caractères que le premier, contient un octet de plus, car le caractère `é` s'écrit avec deux octets (encodage UTF-8 utilisé par défaut).

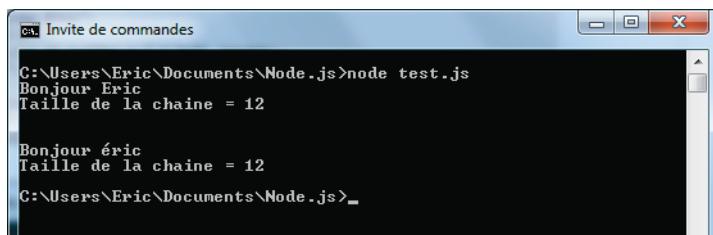
Si on utilise uniquement des chaînes de caractères, sans passer par les buffers proposés par Node, la taille de chacune des chaînes est identique.

Utiliser uniquement des chaînes au lieu de buffers

```
var s1 = "Bonjour Eric";
console.log(s1);
console.log("Taille de la chaine = " + s1.length);
console.log("\n");

var s2 = "Bonjour éric";
console.log(s2);
console.log("Taille de la chaine = " + s2.length);
```

Figure 4-18
Tailles des chaînes identiques



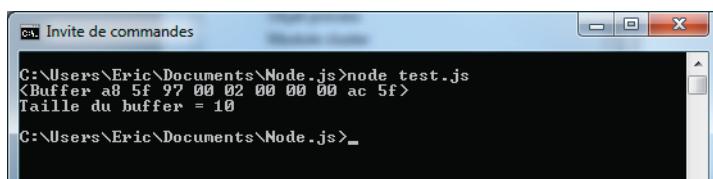
La seconde façon de créer un objet de classe **Buffer** est d'indiquer le nombre d'octets maximal que l'on souhaite réserver lors de sa création. Comme indiqué précédemment, ce nombre d'octets ne pourra pas être modifié par la suite. Donc, l'agrandissement d'un buffer passera par la création d'un nouveau buffer.

Créer un objet Buffer en indiquant sa taille

```
var buf1 = new Buffer(10);
console.log(buf1);
console.log("Taille du buffer = " + buf1.length);
```

Nous créons le buffer en indiquant son nombre maximal d'octets (ici, 10). Puis nous affichons son contenu ainsi que sa longueur.

Figure 4-19
Taille du buffer d'octets



On peut voir que le buffer contient déjà des valeurs, qui sont en fait des valeurs aléatoires correspondant au contenu de la mémoire à cet instant. Cela signifie que le buffer est simplement alloué en mémoire, mais doit être initialisé ensuite.

Modifier un buffer

Une fois créé, le buffer peut être initialisé et modifié. Pour cela, on accède au buffer comme à un tableau d'octets, au moyen de **buf[i]** pour lire ou écrire dans l'indice **i** du buffer (**i** commençant à 0).

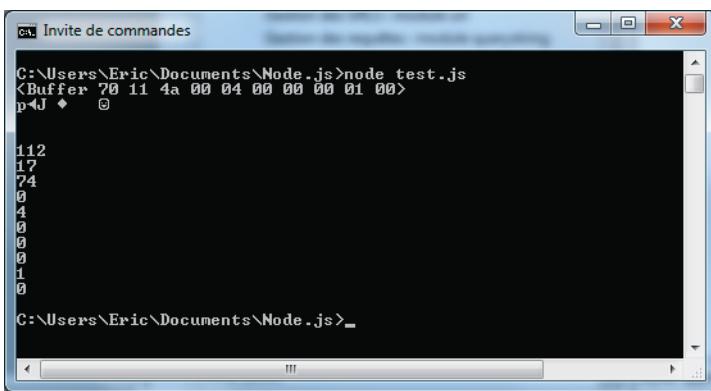
Lire le contenu d'un buffer, octet par octet

```
var buf = new Buffer(10);
console.log(buf);
console.log(buf.toString());

console.log("\n");

for (var i = 0; i < buf.length; i++) {
  console.log(buf[i]);
}
```

Figure 4–20
Caractères dans un buffer



L'exemple qui suit montre comment on peut modifier le contenu d'un buffer, octet par octet. On souhaite ici inscrire, par une boucle, les caractères alphabétiques qui suivent la lettre "A" dans un buffer, sur sa longueur. Ainsi, si le buffer fait dix caractères, on doit inscrire "ABCDEFGHIJ" qui correspondent aux dix premières lettres de l'alphabet.

Inscrire les lettres de A à J dans un buffer au moyen d'une boucle

```
var buf = new Buffer(10);
var from = new Buffer("A");
for (var i = 0; i < buf.length; i++) {
  buf[i] = from[0];
  from[0] = from[0] + 1; // passer à la lettre suivante
}
console.log(buf.toString());
```

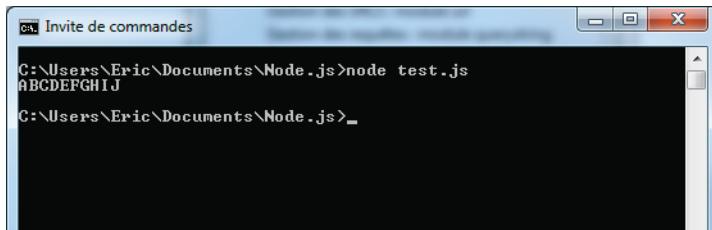
Nous utilisons ici deux buffers.

- Le buffer `buf` contiendra le résultat escompté.
- Le buffer `from` contient la lettre à inscrire dans le buffer résultat. Le fait de passer par un buffer au lieu d'une simple variable sous forme de chaîne permet de mani-

puler le contenu binaire du buffer, et ainsi d'incrémenter le code ASCII de 1 pour passer à la lettre suivante.

Figure 4-21

Buffer sous forme de chaîne de caractères



Il existe un second moyen permettant de modifier un buffer en mémoire, qui consiste à lui affecter certains caractères (ou la totalité) d'une chaîne de caractères. Pour cela, on emploie la méthode `write()` utilisée sur l'objet `buf` correspondant au buffer.

Tableau 4-10 Écriture d'une chaîne dans un buffer buf

Méthode	Signification
<code>buf.write(string, offset, length, encoding)</code>	Écrit dans le buffer <code>buf</code> la chaîne <code>string</code> , à partir de l'offset spécifié dans le buffer (par défaut, 0), et pour la longueur <code>length</code> indiquée pour la chaîne (par défaut, toute la chaîne). Cette dernière doit être encodée selon l'encodage utilisé ("utf8" par défaut). Les principaux encodages sont : <ul style="list-style-type: none"> - "utf8" (par défaut) ; - "base64" ; - "hex".

Un exemple d'utilisation de la méthode `write()` serait le suivant.

Utiliser write() pour écrire une chaîne dans un buffer

```
var buf = new Buffer(10);
console.log(buf);
console.log(buf.toString());

console.log("\n");

buf.write("Bonjour", 1, 3);
console.log(buf);
console.log(buf.toString());
```

On crée d'abord un buffer initialisé avec des valeurs aléatoires, que l'on affiche sous la forme d'octets, puis sous la forme de chaîne de caractères. On affecte ensuite dans ce buffer la chaîne "Bonjour", en la plaçant à l'offset 1 du buffer, et en ne copiant que les trois premiers caractères de la chaîne (ici, "Bon").

Figure 4–22
Modification d'octets
dans un buffer

```
C:\Users\Eric\Documents\Node.js>node test.js
<Buffer c8 81 55 00 94 44 2a 01 00 91>
?U ?D*D? ?

<Buffer c8 42 6f 6e 94 44 2a 01 00 91>
?Bon?D*D? ?

C:\Users\Eric\Documents\Node.js>
```

On peut voir que les caractères "Bon" ont remplacé les trois octets du buffer à partir de l'indice 1, les autres octets du buffer étant inchangés.

Copier et découper un buffer

Afin de manipuler les buffers sous forme d'octets, Node a créé des méthodes spécialisées. Certaines d'entre elles sont des méthodes de classe qui s'utilisent directement sur la classe `Buffer`, tandis que d'autres sont des méthodes d'instance qui s'utilisent sur un objet de classe `Buffer` (ici, représenté par la variable `buf`).

Tableau 4–11 Méthodes de gestion de buffers

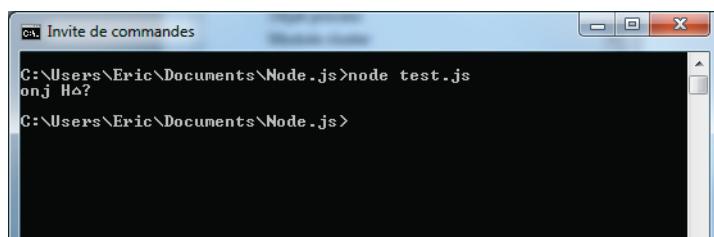
Méthode	Signification
<code>Buffer.concat(list)</code>	Concatène les buffers indiqués dans le tableau <code>list</code> et retourne un nouveau buffer résultat.
<code>buf.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd])</code>	Recopie une partie du buffer <code>buf</code> (comprise entre les indices <code>sourceStart</code> et <code>sourceEnd</code>) dans <code>targetBuffer</code> (à partir de l'indice <code>targetStart</code>). Les valeurs par défaut (optionnelles) sont les suivantes : <ul style="list-style-type: none"> - <code>targetBuffer</code> : <code>0</code>. On recopie au début du buffer destination. - <code>sourceStart</code> : <code>0</code>. On recopie à partir du début du buffer source. - <code>sourceEnd</code> : <code>buf.length</code>. On recopie la totalité du buffer source.
<code>buf.slice([start], [end])</code>	Retourne un nouveau buffer correspondant aux octets situés entre les indices <code>start</code> et <code>end</code> du buffer <code>buf</code> . Le nouveau buffer pointe en réalité vers l'ancien buffer <code>buf</code> (il n'y a pas de nouvelle allocation mémoire pour le nouveau buffer). Les valeurs par défaut (optionnelles) sont les suivantes : <ul style="list-style-type: none"> - <code>start</code> : <code>0</code>. On commence à partir du début du buffer source. - <code>end</code> : <code>buf.length</code>. La fin correspond à la fin du buffer source.
<code>Buffer.byteLength(string, [encoding])</code>	Retourne le nombre d'octets utilisés par la chaîne <code>string</code> dans l'encodage indiqué ("utf8" par défaut).

Utiliser la méthode copy() afin de copier une partie de buffer dans un autre

```
var buf = new Buffer("Bonjour Eric");
var buf2 = new Buffer(10);
buf.copy(buf2, 0, 1, 4);
console.log(buf2.toString());
```

Nous copions dans un buffer `buf2` un extrait d'un premier buffer contenant "Bonjour Eric". La recopie s'effectue à l'indice 0 du buffer `buf2`, tandis que les indices des octets copiés sont compris entre 1 et 4 du buffer d'origine. Seule la chaîne "onj" sera donc copiée dans le second buffer.

Figure 4–23
Copie de buffer



Changer l'encodage d'une chaîne au moyen d'un buffer

Node permet de gérer plusieurs encodages pour les chaînes de caractères, en particulier UTF-8 (utilisé par défaut), mais aussi base 64 ou hexadécimal.

La méthode `buf.toString(encoding)` permet de récupérer une chaîne de caractères correspondant au buffer, mais dans l'encodage indiqué en paramètres.

Tableau 4–12 Récupérer dans un encodage donné un buffer buf

Méthode	Signification
<code>buf.toString(encoding)</code>	Retourne une chaîne de caractères correspondant au buffer <code>buf</code> , dans l'encodage indiqué (par défaut, "utf8"). Les principaux encodages sont : - "utf8" (par défaut) ; - "base64" ; - "hex".

Dans l'exemple qui suit, nous utilisons une chaîne de caractères encodée en UTF-8, que nous mettons dans un buffer. La chaîne correspondant à ce buffer est ensuite affichée en base 64, puis en hexadécimal. L'utilisation du buffer permet ici de passer d'un encodage à l'autre très facilement.

Gérer plusieurs encodages pour une chaîne, au moyen d'un buffer

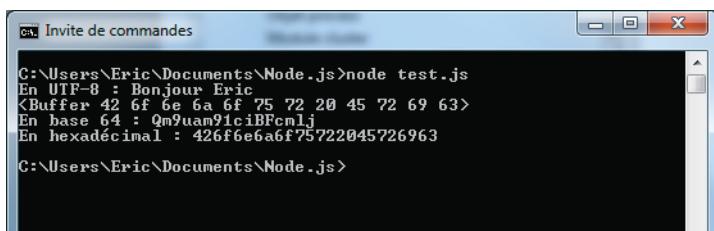
```
var utf8String = "Bonjour Eric";
console.log("En UTF-8 : " + utf8String);

var buf = new Buffer(utf8String);
console.log(buf);

var base64String = buf.toString("base64");
console.log("En base 64 : " + base64String);

var hexaString = buf.toString("hex");
console.log("En hexadécimal : " + hexaString);
```

Figure 4-24
Encodage d'un buffer



On voit donc que le même buffer peut s'afficher de diverses manières selon l'encodage utilisé. Mais les octets correspondants au buffer sont les mêmes quel que soit l'encodage, ce n'est que la représentation sous forme de chaîne de caractères qui diffère.

Connaître la taille d'une chaîne de caractères en octets

On a vu que le nombre de caractères d'une chaîne de caractères n'était pas forcément égal au nombre d'octets qui la constitue. Afin d'avoir facilement accès au nombre d'octets d'une chaîne de caractères, Node a créé la méthode `Buffer.byteLength(string)` qui retourne le nombre d'octets de la chaîne passée en paramètre.

Tableau 4-13 Connaître le nombre d'octets d'une chaîne de caractères

Méthode	Signification
<code>Buffer.byteLength(string, [encoding])</code>	Retourne le nombre d'octets de la chaîne indiquée. Le nombre de caractères est obtenu par <code>string.length</code> (propriété <code>length</code> de la classe <code>String</code>). L'encodage est par défaut "utf8".

Voici un exemple d'utilisation de cette méthode montrant la différence avec la propriété `length` de la classe `String`.

Utilisation de la méthode Buffer.byteLength()

```
var str = "Bonjour éric";
console.log("str = " + str);
console.log("str.length = " + str.length);
console.log("Buffer.byteLength(str) = " + Buffer.byteLength(str));
```

La chaîne analysée contient volontairement un caractère accentué afin que les deux valeurs rentrées ne soient pas identiques.

Figure 4–25
Taille d'un buffer

```
C:\Users\Eric\Documents\Node.js>node test.js
str = Bonjour éric
str.length = 12
Buffer.byteLength(str) = 13
C:\Users\Eric\Documents\Node.js>
```

Bien que la chaîne comporte douze caractères, elle tient sur un buffer de treize octets.

Gestion des timers

La gestion des timers est classique en JavaScript et s'effectue au moyen des méthodes `setTimeout()` et `setInterval()`. Ces méthodes sont accessibles sans passer par un module intermédiaire.

Tableau 4–14 Méthodes de gestion des timers

Méthode	Signification
<code>setTimeout(cb, ms)</code>	Positionne un timer qui se déclenchera une seule fois au bout de <code>ms</code> millisecondes. La fonction de callback indiquée dans le paramètre <code>cb</code> sera déclenchée. La méthode retourne un identifiant de timer <code>t</code> pouvant être utilisé dans <code>clearTimeout(t)</code> .
<code>clearTimeout(t)</code>	Supprime le timer <code>t</code> indiqué et empêche son déclenchement s'il ne s'était pas encore déclenché.
<code>setInterval(cb, ms)</code>	Positionne un timer qui se déclenchera à intervalle fixe toutes les <code>ms</code> millisecondes. La fonction de callback indiquée dans le paramètre <code>cb</code> sera déclenchée à chaque fois. La méthode retourne un identifiant de timer <code>t</code> pouvant être utilisé dans <code>clearInterval(t)</code> .
<code>clearInterval(t)</code>	Supprime le timer <code>t</code> indiqué et empêche son déclenchement futur.

Utilisons la méthode `setInterval()` afin de donner un exemple d'utilisation de ces méthodes. Nous affichons l'heure système toutes les secondes.

Afficher l'heure système toutes les secondes

```
setInterval(function() {  
    var d = new Date();  
    console.log(d);  
}, 1000);
```

Figure 4–26

Affichage périodique de l'heure

```
C:\Users\Eric\Documents\Node.js>node test.js  
Fri May 31 2013 16:22:18 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:19 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:20 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:21 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:22 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:23 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:24 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:25 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:26 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:27 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:28 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:29 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:30 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:31 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:32 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:33 GMT+02:00 <Paris, Madrid <heure d'été>>  
Fri May 31 2013 16:22:34 GMT+02:00 <Paris, Madrid <heure d'été>>
```

5

Gestion des streams

Les *streams* (flux en français) sont une notion importante dans Node car ils permettent d'échanger des informations. Par exemple, un utilisateur qui se connecte à un serveur via son navigateur HTTP, initie un stream qui permet l'échange des informations entre le client et le serveur. Mais un programme qui lit ou écrit dans un fichier sur disque initie également un autre stream représenté par la lecture ou l'écriture du fichier.

Un autre exemple de stream très simple concerne la gestion du clavier et l'affichage sur la console. La récupération des caractères saisis au clavier représente un stream utilisé en lecture, tandis que l'affichage sur la console correspond à un autre stream, cette fois en écriture.

Un stream est donc une abstraction permettant de représenter la lecture ou l'écriture de flux d'octets.

On voit donc que certains streams ne concerneront que des lectures (streams dits « en lecture »), que d'autres ne concerneront que des écritures (streams dits « en écriture »), et enfin que certains seront bidirectionnels (lecture et écriture). Dans ce chapitre, nous étudierons ces trois types de streams.

Créer un stream en lecture

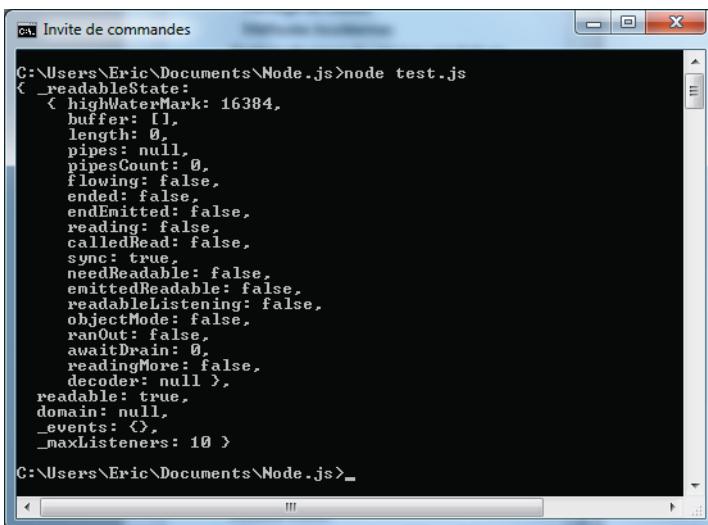
Un stream est utilisable à travers le module `stream`. Pour créer un stream en lecture, on utilise la classe `stream.Readable`.

Créer un objet `stream.Readable`

```
var stream = require("stream");
var readable = new stream.Readable();
console.dir(readable);
```

Figure 5-1

Contenu d'un objet de classe
`stream.Readable`



The screenshot shows a Windows Command Prompt window with the title "Invite de commandes". The command entered is "C:\Users\Eric\Documents\Node.js>node test.js". The output displays the properties of a newly created `stream.Readable` object, including its state and various boolean flags like `highWaterMark`, `buffer`, `length`, `pipes`, `pipesCount`, `flowing`, `ended`, `endEmitted`, `reading`, `calledRead`, `sync`, `needReadable`, `emittedReadable`, `readableListening`, `objectMode`, `ranOut`, `awaitDrain`, `readingMore`, `decoder`, `readable`, `domain`, `_events`, and `_maxListeners`. The output ends with "C:\Users\Eric\Documents\Node.js>".

Utiliser l'événement `data` sur le stream

Un stream étant une entité ayant la capacité d'émettre et/ou de recevoir des informations, il est aussi de classe `events.EventEmitter`.

Vérifier qu'un stream est de classe `events.EventEmitter`

```
var stream = require("stream");
var events = require("events");
var readable = new stream.Readable();
console.log(readable instanceof events.EventEmitter); // true
```

Étant un objet capable de recevoir et d'émettre des événements, un stream créé en lecture pourra recevoir l'événement `data` qui signifie que des données sont présentes, prêtes à être lues.

Gérer la réception de données sur le stream en lecture

```
var stream = require("stream");
var readable = new stream.Readable();

readable.on("data", function(chunk) {
  console.log(chunk);
});
```

Le paramètre `chunk` transmis dans l'événement `data` correspond au paquet d'octets reçus. Plusieurs événements `data` peuvent être reçus si le stream reçoit plusieurs paquets de données.

Figure 5–2
Erreur sur le stream en lecture

```
C:\Users\Eric\Documents\Node.js>node test.js
events.js:72
  throw er; // Unhandled 'error' event
          ^
Error: not implemented
    at Readable._read (<stream_readable.js:430:22)
    at Readable.read (<stream_readable.js:304:10)
    at Readable.<anonymous> (<stream_readable.js:719:45)
    at Readable.EventEmitter.emit (events.js:92:17)
    at emitDataEvents (<stream_readable.js:745:10)
    at Readable.on (<stream_readable.js:666:5)
    at Object.<anonymous> (C:\Users\Eric\Documents\Node.js\test.js:1:10)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:314:12)
    at Module.require (module.js:367:10)
    at require (internal/module.js:16:11)

C:\Users\Eric\Documents\Node.js>_
```

Une erreur se produit, indiquant que la méthode `_read()` n'est pas définie sur le stream. En effet, étant à l'écoute de l'événement `data` sur le stream, celui-ci appelle régulièrement la méthode interne `_read()` qui doit retourner les octets lus sur le stream. Il faut donc définir cette méthode sur l'objet associé au stream.

Définir la méthode `_read()` sur l'objet associé au stream

```
var stream = require("stream");
var readable = new stream.Readable();

readable._read = function(size) {
  console.log("Méthode _read() appelée");
};

readable.on("data", function(chunk) {
  console.log(chunk);
});
```

Le paramètre `size` indique la taille en octets du buffer pouvant être lu lors de l'appel de la méthode `_read()`. Ce paramètre est facultatif et peut être omis lors de l'écriture de la méthode.

Figure 5-3
Stream en lecture fonctionnel

```
C:\Users\Eric\Documents\Node.js>node test.js
Méthode _read() appelée
C:\Users\Eric\Documents\Node.js>
```

L'exécution précédente montre que le positionnement de l'événement `data` sur le stream provoque l'appel de la méthode `_read()`. Toutefois, aucune donnée n'est, pour l'instant, lue sur le stream, c'est pourquoi la lecture sur le stream s'arrête au bout d'un seul appel de la méthode `_read()`. Pour que cette lecture soit effectuée une autre fois, il faudrait qu'un paquet de données soit émis par le stream. Pour cela, on utilise la méthode `readable.push(octets)` qui permet d'indiquer les octets à lire sur le stream.

Émettre un paquet de données sur le stream en lecture

```
var stream = require("stream");
var readable = new stream.Readable();

readable._read = function(size) {
    console.log("Méthode _read() appelée");
};

readable.on("data", function(chunk) {
    console.log(chunk);
});

readable.push("Bonjour1");
```

Figure 5-4
Envoi d'octets sur le stream
en lecture

```
C:\Users\Eric\Documents\Node.js>node test.js
Méthode _read() appelée
Méthode _read() appelée
<Buffer 42 6f 6e 6a 6f 75 72 31>
C:\Users\Eric\Documents\Node.js>
```

La méthode `_read()` est appelée une première fois de façon automatique en raison du positionnement de l'événement `data` sur le stream, puis une seconde fois du fait de l'envoi du paquet correspondant à la chaîne "`Bonjour1`". Le buffer d'octets affichés à la suite correspond aux octets reçus lors du traitement de l'événement `data`.

Envoyons un second paquet "Bonjour2" à la suite du premier.

Envoi de deux paquets de données sur le stream

```
var stream = require("stream");
var readable = new stream.Readable();

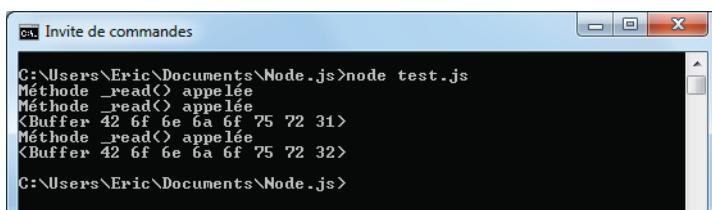
readable._read = function(size) {
    console.log("Méthode _read() appelée");
};

readable.on("data", function(chunk) {
    console.log(chunk);
});

readable.push("Bonjour1");
readable.push("Bonjour2");
```

Figure 5–5

Envoyer de plusieurs paquets sur le stream en lecture



La méthode `_read()` est à nouveau appelée, pour chacun des paquets émis. L'événement `data` est alors déclenché pour chacun de ces paquets.

En résumé, le positionnement de l'événement `data` sur le stream en lecture provoque l'appel de la méthode `_read()` interne au stream. Les appels à la méthode `_read()` suivants seront effectués grâce à l'envoi des paquets par `readable.push(octets)`, ce qui déclenche l'événement `data` positionné sur le stream.

Définir la méthode `_read()` sur le stream

Tout d'abord, le fait que la méthode `_read()` commence par le caractère `_` signifie qu'il s'agit d'une méthode privée du module `stream`, indiquant qu'elle ne doit pas être appellée directement par notre programme. En effet, ce sont les méthodes internes de Node qui l'appellent, comme nous l'avons vu précédemment.

Pour l'instant, la méthode `_read()` ne fait rien d'autre qu'afficher un message indiquant qu'elle est appelée. Mais le rôle de cette méthode est de fournir les octets qui seront reçus par l'événement `data` positionné sur le stream.

Au lieu d'utiliser la méthode `push()` en dehors de la méthode `_read()` comme précédemment, inscrivons ces instructions dans celle-ci.

La méthode `_read()` effectue l'écriture des données à lire sur le stream

```
var stream = require("stream");
var readable = new stream.Readable();

readable._read = function(size) {
  readable.push("Bonjour1");
  readable.push("Bonjour2");
};

readable.on("data", function(chunk) {
  console.log(chunk);
});
```

Figure 5–6
Boucle infinie sur le stream
en lecture

On voit une suite ininterrompue de paquets d'octets, chaque paquet contenant la chaîne "Bonjour1Bonjour2". La question qui se pose est : « Pourquoi cette suite est-elle sans fin ? ».

La raison est la suivante : lors du positionnement de l'événement `data`, la méthode `_read()` est appelée. Cette méthode fournissant des octets à lire sur le stream (par la méthode `push()`), cela provoque le déclenchement de l'événement `data`, mais également un nouvel appel à la méthode `_read()`. D'où la récursivité des appels qui provoque un affichage ininterrompu.

L'écriture de la méthode `_read()` doit donc veiller à ne pas provoquer de récursivité, et par conséquent elle doit contenir une condition de sortie. Par exemple, supposons que l'on souhaite que le stream en lecture soit associé au contenu d'un fichier. La méthode `_read()` retournerait alors le contenu de ce fichier.

Associer la méthode `_read()` à la lecture du contenu d'un fichier

```
var fs = require("fs");
var stream = require("stream");
var readable = new stream.Readable();

var content = fs.readFileSync("stream.txt");

readable._read = function(size) {
  console.log("Méthode _read() appelée");
  if (content) readable.push(content);
  content = "";
};

readable.on("data", function(chunk) {
  console.log(chunk);
});
```

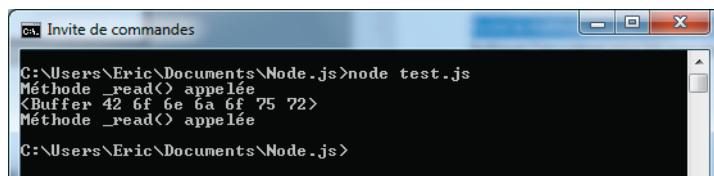
On utilise le module `fs` permettant de lire le contenu d'un fichier (voir chapitre 6). La méthode `fs.readFileSync(name)` permet de lire le fichier représenté par `name` et de retourner le contenu de ce fichier sous forme de buffer d'octets.

Remarquez comment on supprime la récursivité de la méthode `_read()`. Nous utilisons la variable `content` qui contient le contenu du fichier, que nous effaçons après le premier appel à `push()`.

Le positionnement de l'événement `data` provoque donc le premier appel à la fonction `_read()`, qui effectue l'appel à `push()`, qui déclenche l'événement `data`. Cet appel à `push()` provoque un nouvel appel à la méthode `_read()`, mais ce dernier ne fait plus aucun `push()` donc la récursivité s'arrête.

Figure 5–7

Suppression de la boucle infinie sur le stream en lecture



Le fichier lu `stream.txt` est supposé ici contenir la simple chaîne de caractères "Bonjour". Elle correspond au buffer d'octets affiché ici. On voit également que le dernier appel à `_read()` est bien effectué, mais s'arrête aussitôt.

Fichier `stream.txt`

Bonjour

Indiquer l'encodage pour les paquets lus sur le stream

Les octets reçus par l'événement `data` sont affichés sous forme de `Buffer`, donc en hexadécimal. Il est possible d'indiquer l'encodage à utiliser, et donc d'afficher ces octets sous forme de chaîne de caractères, par exemple en UTF-8.

La méthode `readable.setEncoding("utf8")` permet d'indiquer l'encodage des chaînes de caractères en UTF-8. Tous les octets utilisés par le stream seront affichés à l'écran en UTF-8.

Tableau 5-1 Indiquer l'encodage pour les octets lus sur le stream

Méthode	Signification
<code>readable.setEncoding(encoding)</code>	Définit l'encodage des octets lus sur le stream. Par défaut, les octets sont lus sous forme de <code>Buffer</code> . Le paramètre <code>encoding</code> peut valoir : - "utf8" pour un encodage en UTF-8 ; - "hex" pour un encodage en hexadécimal.

Afficher les octets lus sur le stream en UTF-8

```
var fs = require("fs");
var stream = require("stream");
var readable = new stream.Readable();

var content = fs.readFileSync("stream.txt");

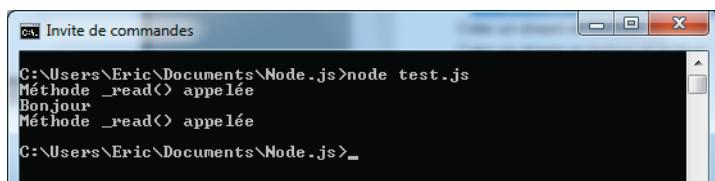
readable.setEncoding("utf8");

readable._read = function(size) {
  console.log("Méthode _read() appelée");
  if (content) readable.push(content);
  content = "";
};

readable.on("data", function(chunk) {
  console.log(chunk);
});
```

Figure 5-8

Encodage UTF-8 sur le stream en lecture



Utiliser la méthode `read()` sur le stream plutôt que l'événement `data`

L'événement `data` permet d'être prévenu que des octets sont disponibles en lecture sur le stream. Au lieu d'utiliser cette technique, on peut aussi décider de lire le flux d'octets lorsqu'on le souhaite. On utilise pour cela la méthode `readable.read(size)` (sans underscore (`_`) dans la méthode `read()`) qui permet de récupérer le nombre d'octets indiqué dans `size`. Si `size` n'est pas précisé, l'appel de cette méthode récupère l'ensemble des octets actuellement disponibles (ou retourne `null` si aucun octet). De plus, si on essaie de récupérer plus d'octets que ceux qui sont disponibles, cette méthode retourne également `null`.

Tableau 5–2 Méthode de lecture des octets disponibles sur le stream en lecture

Méthode	Signification
<code>readable.read([size])</code>	Lit <code>size</code> octets sur le stream. Si <code>size</code> n'est pas indiqué (optionnel), lit tous les octets actuellement disponibles. Retourne un buffer contenant les octets lus (ou une chaîne si l'encodage est UTF-8), ou <code>null</code> si <code>size</code> octets n'ont pas pu être lus.

Node permet d'être prévenu que des octets sont disponibles et prêts à être lus. Pour ce faire, le stream reçoit l'événement `readable`, dans lequel on peut appeler la méthode `readable.read()`. L'événement `readable` ne sera alors déclenché la fois suivante que si les octets présents sur le stream ont tous été lus la fois précédente. Ainsi, lors de l'utilisation de l'événement `readable`, il est conseillé d'utiliser la méthode `read()` sans indiquer le paramètre `size` (de façon à tout lire à chaque fois sans bloquer le déclenchement de l'événement).

Lorsque l'événement `readable` est utilisé, sa présence déclenche automatiquement l'appel de la méthode `_read()`, tout comme la présence de l'événement `data` la déclencheait.

Réécrivons le précédent programme en utilisant la méthode `read()` au lieu de l'événement `data`.

Utiliser la méthode `readable.read()`

```
var fs = require("fs");
var stream = require("stream");
var readable = new stream.Readable();

var content = fs.readFileSync("stream.txt");

readable._read = function(size) {
  console.log("Méthode _read() appelée");
  if (content) readable.push(content);
}
```

```

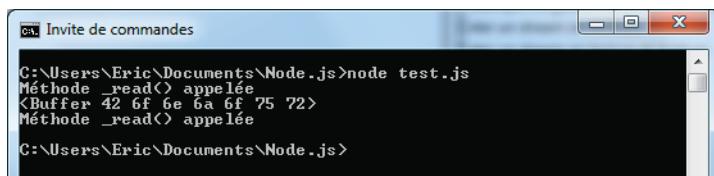
        content = "";
};

var buf = readable.read();
console.log(buf);

```

Cette exécution montre que `readable.read()` appelle la méthode `_read()`, puis le contenu du buffer lu est affiché par `console.log(buf)`. La méthode `_read()` est appelée une seconde fois suite à l'exécution de la méthode `push()`.

Figure 5-9
Utilisation de la méthode `read()`



Écrivons une variante de ce programme en utilisant simultanément la méthode `read()` et l'événement `readable`.

Utiliser la méthode `readable.read()` et l'événement `readable`

```

var fs = require("fs");
var stream = require("stream");
var readable = new stream.Readable();

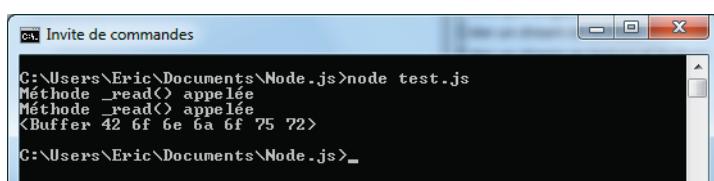
var content = fs.readFileSync("stream.txt");

readable._read = function(size) {
  console.log("Méthode _read() appelée");
  if (content) readable.push(content);
  content = "";
};

readable.on("readable", function() {
  var buf = readable.read();
  console.log(buf);
});

```

Figure 5-10
Utilisation de l'événement
`readable`



Le premier appel de la méthode `_read()` est effectué grâce au positionnement de l'événement `readable`. Puis, l'appel de la méthode `push()` entraîne alors le déclenchement de l'événement `readable` qui effectue l'affichage du buffer lu sur le stream.

Connaître la fin du stream en lecture avec l'événement end

Un stream en lecture peut recevoir de multiples événements `data` signalant la présence de données à lire sur le stream. Lorsque plus aucune donnée n'est à lire (par exemple, fin d'un fichier), le stream émet l'événement `end`, signalant ainsi la fin du stream.

Exemple de gestion d'un stream en lecture

On souhaite montrer un exemple de gestion d'un stream en lecture, permettant d'afficher les chaînes de caractères saisies au clavier.

Gérer les caractères lus au clavier dans un stream en lecture

```
var stream = require("stream");
var readable = new stream.Readable();

readable._read = function(size) {
  process.stdin.removeAllListeners("data").on("data", function(chunk) {
    readable.push(chunk);
  });
};

readable.on("data", function(chunk) {
  console.log(chunk);
});
```

Le stream gérant les caractères saisis au clavier est géré par Node dans l'objet `process.stdin`. On récupère les paquets de données entrées au clavier grâce à l'événement `data` géré sur ce stream. Ce stream étant géré dans la méthode `_read()` appelée après chaque `push()`, il faut supprimer tous les gestionnaires précédemment positionnés au moyen de `process.stdin.removeAllListeners("data")`.

Voici un exemple d'exécution après avoir saisi quelques chaînes de caractères, séparées par des retours à la ligne.

Figure 5–11
Capture de caractères
au clavier

```
C:\Users\Eric\Documents\Node.js>node test.js
bonjour
<Buffer 62 6f 6e 6a 6f 75 72 0d 0a>
hello
<Buffer 68 65 6c 6c 6f 0d 0a>
Je vais bien
<Buffer 6a 65 20 76 61 69 73 20 62 69 65 6e 0d 0a>

<Buffer 0d 0a>
OK
<Buffer 4f 4b 0d 0a>
```

En utilisant l'événement `readable` et la méthode `read()`, le programme précédent peut aussi s'écrire :

Utiliser l'événement readable et la méthode read()

```
var stream = require("stream");
var readable = new stream.Readable();

readable._read = function(size) {
  process.stdin.removeAllListeners("data").on("data", function(chunk) {
    readable.push(chunk);
  });
};

readable.on("readable", function() {
  console.log(readable.read());
});
```

Utiliser les méthodes pause() et resume()

On peut mettre en pause la lecture d'octets sur un stream par `readable.pause()`, et lui demander de reprendre la lecture au moyen de `readable.resume()`. Remarquez que seule la lecture est mise en pause, mais les octets peuvent continuer à arriver. Ils seront lus après que l'instruction `readable.resume()` soit exécutée.

Tableau 5–3 Méthodes de gestion du stream en lecture

Méthode	Signification
<code>readable.pause()</code>	Met en attente la lecture d'octets sur le stream. La lecture ne pourra reprendre qu'après <code>stream.resume()</code> .
<code>readable.resume()</code>	Reprend la lecture d'octets sur le stream.

Utilisons ces méthodes pour gérer les caractères tapés au clavier. On souhaite introduire des caractères qui seront affichés sur la ligne du dessous après avoir appuyé sur la touche Entrée. Il est également possible de quitter la saisie en tapant la chaîne "exit".

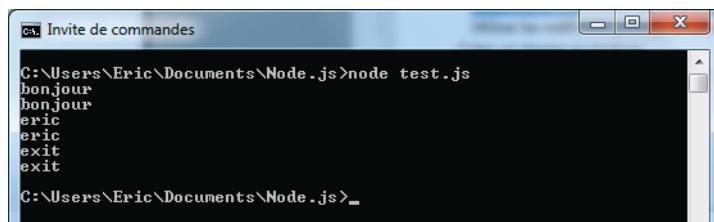
Gérer la saisie de caractères et quitter lorsque l'utilisateur tape "exit"

```
process.stdin.on("data", function(chunk) {
  chunk = chunk.toString().replace(/[\r\n]/g, "");
  console.log(chunk);
  if (chunk == "exit") this.pause();
});
```

Le stream associé à la gestion du clavier est `process.stdin`. Nous récupérons le buffer d'octets `chunk` que nous transformons en chaîne grâce à la méthode `toString()`. Nous éliminons les caractères "`\r`" et "`\n`" de retour à la ligne, inclus dans les octets du buffer récupéré. Si la chaîne saisie correspond à "`exit`", le stream est mis en pause, ce qui fait sortir de la saisie.

Figure 5-12

Sortie du stream en lecture par "exit"



Les caractères saisis sont reproduits sur la ligne du dessous, et dès que l'on reconnaît la chaîne "`exit`", la saisie s'interrompt.

Modifions le programme afin d'utiliser la méthode `resume()`. Nous souhaitons qu'après être sorti de la saisie, on y retourne au bout de deux secondes.

Revenir en saisie au bout de deux secondes

```
process.stdin.on("data", function(chunk) {
  chunk = chunk.toString().replace(/[\r\n]/g, "");
  console.log(chunk);
  if (chunk == "exit") {
    this.pause();
    setTimeout(function() {
      process.stdin.resume();
    }, 2000);
  }
});
```

À la fin du timer, on exécute l'instruction `process.stdin.resume()`, qui permet de continuer à lire le flux d'octets reçus du clavier. Remarquez que les caractères tapés entre les instructions `pause()` et `resume()` sont bufferisés et transmis dès l'ouverture du stream.

Créer un stream en écriture

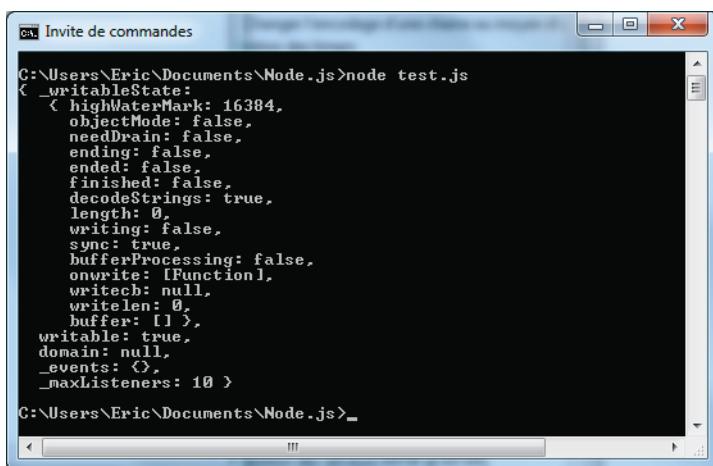
On peut également créer un stream en écriture. Pour cela, on utilise la classe `stream.Writable` du module `stream`.

Créer un objet `stream.Writable`

```
var stream = require("stream");
var writable = new stream.Writable();
console.log(writable);
```

Figure 5-13

Contenu d'un objet de classe `stream.Writable`



```
cd\Users\Eric\Documents\Node.js>node test.js
C:\Users\Eric\Documents\Node.js>{
  _writableState:
    {
      highWaterMark: 16384,
      objectMode: false,
      needDrain: false,
      ending: false,
      ended: false,
      finished: false,
      decodeStrings: true,
      length: 0,
      writing: false,
      sync: true,
      bufferProcessing: false,
      onwrite: [Function],
      writecb: null,
      writelen: 0,
      buffer: [ ],
      writable: true,
      domain: null,
      _events: {},
      _maxListeners: 10
    }
}
C:\Users\Eric\Documents\Node.js>
```

L'objet représenté par le stream en écriture est, comme pour un stream en lecture, également de classe `events.EventEmitter`. Il pourra donc émettre et recevoir des événements.

Nous avons ici utilisé le constructeur de la classe `stream.Writable` sans lui spécifier de paramètres. Toutefois, on peut lui indiquer un objet en paramètres dont la propriété `highWaterMark` représente le nombre d'octets que Node peut mémoriser en interne lorsque l'instruction `write()`, qui permet d'écrire sur le stream, retourne `false`. Il est alors préférable d'attendre que tous les octets écrits sur le stream aient été « absorbés » avant de continuer à écrire sur celui-ci, pour ne pas risquer d'atteindre la capacité maximale du buffer temporaire. Les octets qui n'ont pas pu être écrits le seront dès que possible par Node.

Dans la section suivante, nous allons voir comment utiliser la fonction `write()`.

Utiliser la méthode `write()` sur le stream

Un stream en écriture s'utilise par l'intermédiaire de la méthode `write()` permettant d'écrire des octets sur le stream.

Tableau 5–4 Méthode d'écriture des octets sur le stream en écriture

Méthode	Signification
<code>writable.write(chunk, [encoding], [callback])</code>	Écrit le buffer d'octets indiqué par <code>chunk</code> sur le stream. Si le buffer est représenté par une chaîne de caractères, le paramètre facultatif <code>encoding</code> permet de préciser l'encodage utilisé ("utf8" par défaut). Le paramètre <code>callback</code> représente une éventuelle fonction de callback qui sera appelée à la fin de l'exécution de l'instruction. Le retour de la fonction <code>write()</code> est <code>true</code> si le buffer a pu être écrit, <code>false</code> dans le cas contraire. Si la méthode retourne <code>false</code> , le buffer est mis dans une mémoire temporaire dont la taille maximale est indiquée par la propriété <code>highWaterMark</code> . Il sera écrit sur le stream aussitôt que possible, de façon transparente par Node.

Définir la méthode `_write()` sur le stream

Comme pour les streams en lecture, Node demande à ce que la méthode interne d'écriture soit définie, car elle sera appelée en interne lors de toute écriture d'octets sur le stream lors de l'instruction `write()`. Cette méthode interne est nommée `_write(chunk, encoding, callback)`. Elle utilise trois paramètres, tous obligatoires.

- Le paramètre `chunk` indique le buffer à écrire sur le stream.
- Le paramètre `encoding` précise l'encodage dans le cas où le buffer est une chaîne de caractères ("utf8" par défaut).
- Le paramètre `callback` est une fonction de callback que l'on devra appeler dans la méthode `_write()` pour indiquer la fin de l'écriture sur le stream (que le buffer soit écrit sur le stream ou simplement bufferisé temporairement).

Dans l'exemple ci-dessous, nous définissons la méthode `_write()` sur le stream en écriture, puis nous effectuons trois écritures sur ce stream.

Définir la méthode `_write()` et effectuer trois écritures sur le stream

```
var stream = require("stream");
var writable = new stream.Writable();

writable._write = function(chunk, encoding, callback) {
  console.log("Appel _write");
  callback();
};

var ret1 = writable.write("1234567890", function() {
  console.log("Fin write1");
});
console.log("Retour write1 = " + ret1);
```

```

var ret2 = writable.write("2", function() {
    console.log("Fin write2");
});
console.log("Retour write2 = " + ret2);

var ret3 = writable.write("3");
console.log("Retour write3 = " + ret3);

```

Attention de ne pas confondre la méthode `_write()` et la méthode `write()`. La méthode `_write()` est appelée en interne par Node à chaque écriture sur le stream (chaque écriture étant effectuée par un appel à la méthode `write()` dans notre programme).

Nous avons donc défini la méthode `_write()` sur le stream en écriture. Elle fait un simple affichage indiquant son appel, puis elle appelle la fonction de callback indiquée en paramètres.

Ensuite, nous effectuons trois appels à la méthode `write()` sur l'objet `stream` représenté par `writable`. Nous indiquons une fonction de callback pour les deux premiers appels, mais pas pour le dernier. Nous affichons le retour de la méthode `write()` pour chacun des appels (`true` ou `false`).

Figure 5-14
Utilisation de la méthode
`_write()` sur le stream
en écriture

```

C:\Users\Eric\Documents\Node.js>node test.js
Appel _write
Retour write1 = true
Appel _write
Retour write2 = true
Appel _write
Retour write3 = true
Fin write1
Fin write2

C:\Users\Eric\Documents\Node.js>_

```

Nous voyons les trois appels à la méthode `write()`, qui se traduisent par des appels vers la méthode `_write()`. Chacun des appels à la méthode `write()` retourne `true`. Les fonctions de callback indiquées lors de l'appel sont appelées lorsque le buffer est traité par le stream, bien après le retour de l'instruction `write()`.

Indiquer la fin d'un stream en écriture

Il est possible de fermer un stream en écriture. Pour cela, on utilise la méthode `writable.end()` sur le stream. Les appels suivants `writable.write()` produiront une erreur, car l'écriture est alors impossible. Pour éviter de générer une erreur qui arrête le programme, on gère l'événement `error` sur le stream. Le programme ne provoque plus d'erreur fatale, même si on écrit sur le stream fermé.

Tableau 5–5 Méthode de fermeture d'un stream en écriture

Méthode	Signification
writable.end([chunk], [encoding], [callback])	Ferme le stream. Le buffer <code>chunk</code> , si indiqué, sera écrit sur le stream avant sa fermeture. Les paramètres <code>chunk</code> , <code>encoding</code> et <code>callback</code> sont facultatifs et ont la même signification que dans la méthode <code>write(chunk, encoding, callback)</code> .

Utiliser la méthode end() pour fermer le stream

```
var stream = require("stream");
var writable = new stream.Writable();

writable._write = function(chunk, encoding, callback) {
  console.log("Appel _write");
  callback();
};

writable.on("error", function(error) {
  console.log(error);
});

var ret1 = writable.write("1234567890", function() {
  console.log("Fin write1");
});
console.log("Retour write1 = " + ret1);

writable.end();

var ret2 = writable.write("2", function() {
  console.log("Fin write2");
});
console.log("Retour write2 = " + ret2);

var ret3 = writable.write("3");
console.log("Retour write3 = " + ret3);
```

Ce programme est similaire au précédent. Nous avons simplement fermé le stream suite à la première écriture sur celui-ci, et nous gérons l'événement `error` pour ne pas provoquer d'erreur d'exécution.

Figure 5–15

Fermeture d'un stream
en écriture

```
C:\Users\Eric\Documents\Node.js>node test.js
Appel _write
[Error: write after end]
Retour write1 = true
Fin write1
Retour write2 = false
Fin write2
Retour write3 = false
Fin write3
C:\Users\Eric\Documents\Node.js>
```

Le premier appel à la méthode `write()` s'effectue correctement, tandis que les deux appels suivants provoquent une erreur "Error : write after end". Toutefois, les fonctions de callback situées dans les méthodes `write()` sont appelées, même si la méthode `write()` retourne `false`.

Connecter un stream en lecture sur un stream en écriture

Il est fréquent d'avoir besoin de connecter un stream en lecture sur un stream en écriture. Tous les octets lus sur le stream en lecture sont redirigés vers le stream en écriture. Pour connecter les deux streams, on utilise la méthode `readable.pipe(writable)`.

Tableau 5–6 Méthodes de connexion d'un stream en lecture vers un stream en écriture

Méthode	Signification
<code>readable.pipe(writable)</code>	Connecte le stream en lecture <code>readable</code> vers le stream en écriture <code>writable</code> . Tous les octets lus sur le stream en lecture sont transmis automatiquement vers le stream en écriture.
<code>readable.unpipe([writable])</code>	Annule l'instruction <code>readable.pipe(writable)</code> précédemment effectuée. Si le stream <code>writable</code> n'est pas indiqué, annule toutes les instructions <code>readable.pipe(writable)</code> effectuées, quel que soit le stream en écriture utilisé.

Utiliser la méthode `pipe()` pour connecter un stream en lecture et un stream en écriture

```
var stream = require("stream");
var writable = new stream.Writable();

writable._write = function(chunk, encoding, callback) {
  console.log("Appel _write : " + chunk);
  callback();
};

var readable = new stream.Readable();

readable._read = function(size) {
  process.stdin.removeAllListeners("data").on("data", function(chunk) {
    readable.push(chunk);
  });
};

readable.on("data", function(chunk) {
  console.log(chunk);
});

readable.pipe(writable);
```

Nous définissons d'abord le stream en écriture de façon classique en écrivant sa méthode interne `_write()`. Puis nous définissons le stream en lecture en écrivant sa méthode interne `_read()`. Les deux streams sont connectés au moyen de `readable.pipe(writable)`.

Figure 5-16

Connexion d'un stream en lecture sur un stream en écriture

```
C:\> Invite de commandes - node test.js
C:\Users\Eric\Documents\Node.js>node test.js
bonjour
<Buffer 62 6f 6e 6a 6f 75 72 0d 0a>
Appel _write : bonjour

c'est moi
<Buffer 63 27 65 73 74 20 6d 6f 69 0d 0a>
Appel _write : c'est moi

eric
<Buffer 65 72 69 63 0d 0a>
Appel _write : eric
```

Exemple de stream en écriture

On désire montrer un exemple de stream en écriture faisant appel aux notions vues précédemment. Il s'agit d'utiliser la méthode `console.log()`, qui affiche normalement des messages sur la console, mais ici elle insérera le message dans un fichier `logs.txt`. Cela permettra de conserver dans un fichier tous les affichages effectués au moyen de `console.log()`.

Les caractères écrits sur l'écran au moyen de `console.log()` sont en réalité redirigés par Node vers le stream associé à `process.stdout`. On redéfinit la méthode `_write()` sur ce stream afin qu'elle écrive dans un fichier `logs.txt`.

Rediriger les caractères écrits sur process.stdout vers un fichier logs.txt

```
var fs = require("fs");
var f = fs.openSync("logs.txt", "a");
process.stdout._write = function(chunk, encoding, callback) {
  fs.writeSync(f, chunk);
  callback();
};

console.log("Exécution en date du " + (new Date()));
console.log("Message1");
console.log("Message2");
console.log("Message3");
```

Nous utilisons le module `fs` permettant d'écrire dans un fichier (voir chapitre 6). La méthode `_write()` est redéfinie sur le stream `process.stdout`. Les instructions `console.log()` qui suivent permettent de vérifier la bonne exécution du programme.

Une fois le programme exécuté, le fichier `logs.txt` est créé et contient les données transmises dans les instructions `console.log()`.

Fichier logs.txt suite à une exécution du programme précédent

```
Exécution en date du Thu Jun 13 2013 17:25:36 GMT+0200 (Paris, Madrid (heure  
d'été))  
Message1  
Message2  
Message3
```

Une autre façon de procéder, permettant le même comportement et n'utilisant pas les streams, consisterait à réécrire la méthode `console.log()`. Le programme pourrait alors s'écrire :

Réécriture de la méthode `console.log()` afin d'écrire dans le fichier logs.txt

```
var fs = require("fs");  
var f = fs.openSync("logs.txt", "a");  
console.log = function(chunk) {  
    fs.writeSync(f, chunk + "\n");  
};  
  
console.log("Exécution en date du " + (new Date()));  
console.log("Message1");  
console.log("Message2");  
console.log("Message3");
```

Créer un stream en lecture et écriture

Jusqu'à présent, nous avons défini les streams soit en lecture, soit en écriture. Mais un stream peut être défini en lecture et en écriture, ce qui permet de le lire mais aussi de l'écrire.

Node a défini la classe `stream.Duplex` dans le module `stream` permettant de créer des streams en lecture et en écriture. Les objets de cette classe devront implémenter les méthodes `_read()` et `_write()` vues précédemment, afin que le stream correspondant puisse s'utiliser en lecture et en écriture.

Dans l'exemple suivant, nous lisons des chaînes de caractères au clavier, puis nous transmettons à un stream `duplex` utilisé en lecture et en écriture.

Créer un stream en lecture et en écriture

```
var stream = require("stream");
var duplex = new stream.Duplex();

duplex._write = function(chunk, encoding, callback) {
  console.log("Écrit sur le stream : " + chunk);
  callback();
};

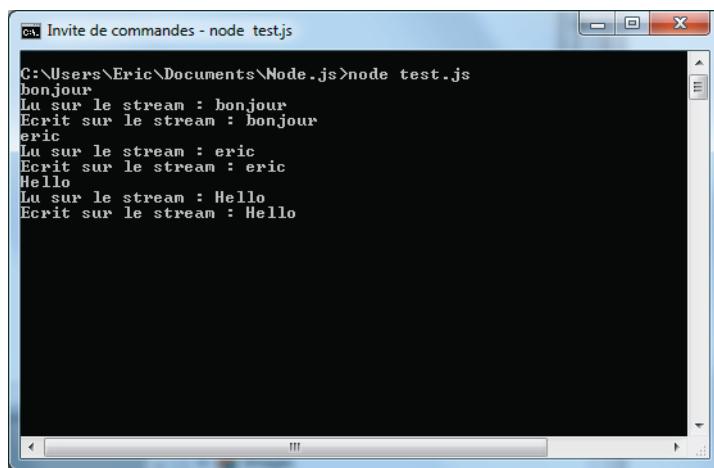
duplex._read = function(size) {
  process.stdin.removeAllListeners("data").on("data", function(chunk) {
    chunk = chunk.toString().replace(/[\d\n]/g, "");
    duplex.push(chunk);
    duplex.write(chunk);
  });
};

duplex.on("data", function(chunk) {
  console.log("Lu sur le stream : " + chunk);
});
```

Au fur et à mesure que les chaînes de caractères sont récupérées au clavier, elles sont transmises au stream en lecture au moyen de l'instruction `duplex.push(chunk)`. Elles sont également transmises au stream en écriture au moyen de l'instruction `duplex.write(chunk)`.

Figure 5-17

Stream en lecture et écriture



6

Gestion des fichiers

Les fichiers sont des streams particuliers, tels que ceux étudiés dans le chapitre précédent. Mais on peut également leur adjoindre une API spécialisée qui est définie dans le module `fs` de Node (`fs` pour *FileSystem*). Cette API permet les opérations traditionnelles sur les fichiers, telles que lire ou écrire des fichiers.

Gestion synchrone et gestion asynchrone

Node permet une gestion de fichiers en mode synchrone ou asynchrone. La plupart du temps, on utilisera la gestion asynchrone, mais il est utile de connaître les deux modes de fonctionnement.

Gestion synchrone

La gestion synchrone des fichiers correspond à celle fréquemment utilisée sur les serveurs autres que Node. Le résultat de l'opération (une lecture, une écriture) est obtenu à la fin de l'exécution de l'instruction. Par exemple, on lit un fichier, et le résultat (le contenu du fichier) est utilisé dans l'instruction qui suit. Ce mode de fonctionnement est classique avec les serveurs traditionnels et c'est aussi le cas dans Node. Par exemple, la lecture d'un fichier en mode synchrone s'effectue au moyen de l'instruction `fs.readFileSync(filename)`. Le retour de l'instruction correspond aux octets lus dans le fichier.

Lire un fichier en mode synchrone

```
var fs = require("fs");
var data = fs.readFileSync("fichier.txt");
console.log("Contenu du fichier fichier.txt : ");
console.log(data);
```

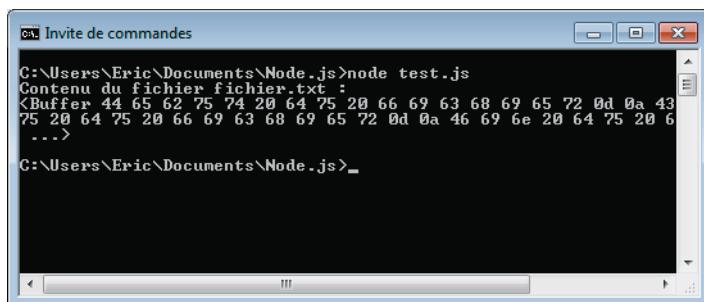
Nous souhaitons ici afficher le contenu du fichier nommé `fichier.txt`. On suppose que ce fichier est situé dans le même répertoire que celui du programme Node. Le fichier contient par exemple :

Fichier fichier.txt

```
Début du fichier
Contenu du fichier
Fin du fichier
```

Figure 6-1

Lecture d'un fichier en mode synchrone



On voit que le contenu du fichier est obtenu directement après l'exécution de l'instruction. Il s'affiche sous forme de buffer d'octets, mais il peut également s'afficher sous forme de texte, comme on le verra par la suite.

Gestion asynchrone

La gestion asynchrone des fichiers est particulière à Node. Elle permet de ne pas bloquer le programme du serveur lors des opérations de lecture ou d'écriture des fichiers. Ces opérations sont longues (d'un point de vue informatique) et l'attente du résultat (le contenu d'un fichier, par exemple, lors d'une lecture) est pénalisante pour les utilisateurs du serveur qui ne peuvent pas accéder au serveur durant ce laps de temps.

Node permet donc d'effectuer les accès aux fichiers de manière asynchrone, à l'aide des fonctions de callback passées en paramètres des méthodes. La méthode `fs.readFile(filename, callback)` est similaire à la version synchrone de la

méthode `fs.readFileSync(filename)`, mais restitue le résultat dans la fonction de callback plutôt qu'en retour de méthode.

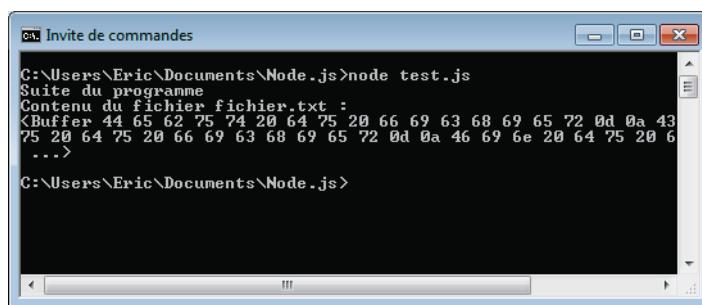
Lire le fichier en mode asynchrone

```
var fs = require("fs");
fs.readFile("fichier.txt", function(err, data){
  console.log("Contenu du fichier fichier.txt : ");
  console.log(data);
});
console.log("Suite du programme");
```

Le contenu du fichier lu correspond ici au paramètre `data` de la fonction de callback passée en paramètres. La suite du programme, qui exploite le contenu du fichier, doit se situer dans la fonction de callback, et non pas à la suite de l'appel de la méthode `fs.readFile()`.

Figure 6–2

Lecture d'un fichier en mode asynchrone



Le résultat est le même que pour la version synchrone de la méthode, seule la manière d'obtenir le résultat diffère.

De plus, on voit que la suite du programme est exécutée avant la fonction de callback, ce qui montre le caractère asynchrone de la méthode `fs.readFile()`.

Quelle gestion (synchrone ou asynchrone) choisir ?

On voit que deux solutions sont possibles pour effectuer les opérations classiques telles que la lecture d'un fichier. La version synchrone d'une méthode bloquera l'exécution de la méthode jusqu'à l'obtention du résultat, tandis que la version asynchrone ne la bloquera pas.

Le choix de l'une ou l'autre des solutions serait sans incidence si Node était un serveur multithread, c'est-à-dire s'il instanciait un nouveau thread pour chaque utilisateur qui se connecte et utilise le serveur. Le blocage des instructions serait uniquement au niveau du thread de l'utilisateur, laissant les autres threads non bloqués.

Mais comme Node possède une seule boucle de traitement pour tous les utilisateurs, la version synchrone des méthodes est rédhibitoire dans ce contexte.

Si la version synchrone des méthodes est présente dans Node, c'est pour faciliter l'écriture des programmes lorsque la version synchrone peut être utilisée sans incidence. C'est le cas, par exemple, lorsque le serveur démarre, alors que les utilisateurs ne sont pas encore en train d'utiliser le serveur.

En résumé, on utilisera presque toujours la version asynchrone des méthodes de gestion de fichiers, et parfois la version synchrone lorsqu'on est certain que cela ne bloquera pas les utilisateurs du serveur. Dans le module `fs` de Node, les méthodes comportant le mot "Sync" dans leur nom sont synchrones, et toutes celles qui ne contiennent pas ce mot sont asynchrones.

Ouvrir et fermer un fichier

Node permet l'opération classique d'ouverture et de fermeture de fichier. L'ouverture d'un fichier permet de récupérer un descripteur de fichier (*file descriptor*, abrégé en `fd`). Les accès suivants au fichier sont alors plus rapides car le fichier est ouvert et peut être accédé de multiples fois avant sa fermeture.

La plupart des méthodes gérant les fichiers ont deux formes :

- la première utilise le nom du fichier, indiqué sous forme de chaîne de caractères ;
- la seconde utilise le descripteur de fichier (`fd`), obtenu lors de l'ouverture du fichier.

Ouvrir un fichier

Regardons en détail les méthodes permettant d'ouvrir et de fermer les fichiers dans Node.

Tableau 6–1 Méthodes d'ouverture de fichier

Méthode	Signification
<code>fs.open(path, flags, callback)</code>	Ouvre le fichier selon le chemin indiqué (<code>path</code>). La fonction de callback est de la forme <code>function (err, fd)</code> dans laquelle <code>fd</code> est le descripteur du fichier ouvert s'il n'y a pas eu d'erreur. Le paramètre <code>flags</code> est une chaîne de caractères indiquant le mode d'ouverture du fichier (lecture, écriture, ajout, etc.). Ces caractères sont décrits à la suite.
<code>fs.openSync(path, flags)</code>	Version synchrone de <code>fs.open()</code> . Le retour de la méthode est le descripteur de fichier obtenu (<code>fd</code>).

Les flags utilisés pour ouvrir un fichier sont indiqués dans le tableau 6–2.

Tableau 6–2 Flags utilisés lors de l'ouverture de fichier

Flag	Signification
r	Ouverture du fichier en lecture seule. Le fichier doit exister.
r+	Ouverture du fichier en lecture et écriture. Le fichier doit exister.
w	Ouverture du fichier en écriture seule. Le fichier est créé (s'il n'existe pas) ou écrasé (s'il existe).
w+	Ouverture du fichier en lecture et écriture. Le fichier est créé (s'il n'existe pas) ou écrasé (s'il existe).
a	Ouverture du fichier en mode ajout. Le fichier est créé (s'il n'existe pas) ou modifié (s'il existe).
a+	Ouverture du fichier en lecture et ajout. Le fichier est créé (s'il n'existe pas) ou modifié (s'il existe).

Fermer un fichier

Une fois le fichier ouvert, puis lu et/ou écrit, il faut le fermer. Ceci peut se faire grâce à l'une des méthodes suivantes. Ces deux méthodes utilisent le descripteur de fichier `fd` obtenu lors de l'ouverture du fichier.

Un fichier est ouvert en utilisant son chemin (`path`), mais est fermé en utilisant son descripteur de fichier (`fd`), obtenu lors de l'ouverture du fichier.

Tableau 6–3 Méthodes de fermeture de fichier

Méthode	Signification
<code>fs.close(fd, callback)</code>	Ferme le fichier représenté par son descripteur de fichier <code>fd</code> . La fonction de callback est appelée à l'issue de la fermeture.
<code>fs.closeSync(fd)</code>	Version synchrone de <code>fs.close()</code> .

Remarquons qu'un fichier peut être ouvert en utilisant le mode synchrone et fermé dans le mode asynchrone (ou l'inverse).

Lire un fichier

Il est possible de lire un fichier soit en totalité, soit partiellement.

- La lecture du fichier en totalité s'effectue en indiquant le nom du fichier aux méthodes `fs.readFile()` et `fs.readFileSync()`.
- La lecture partielle du fichier s'effectue en indiquant le descripteur de fichier aux méthodes `fs.read()` et `fs.readSync()`.

Lecture du fichier en totalité

Nous avons précédemment utilisé les méthodes `fs.readFile()` et `fs.readFileSync()` pour lire le contenu d'un fichier. Le descriptif de ces deux méthodes est donné dans le tableau 6-4.

Tableau 6-4 Méthodes de lecture de fichier à partir du nom

Méthode	Signification
<code>fs.readFile(filename, [options], callback)</code>	<p>Lit le fichier indiqué, puis appelle la fonction de callback de la forme <code>function(err, data)</code>, dans laquelle le paramètre <code>data</code> représente le contenu du fichier lu.</p> <p>Le paramètre <code>options</code> permet de préciser le mode d'ouverture (<code>flag</code>) du fichier et son encodage (<code>encoding</code>).</p> <ul style="list-style-type: none"> - L'option <code>flag</code> peut valoir "<code>r</code>", "<code>r+</code>", "<code>w</code>", "<code>w+</code>", "<code>a</code>" ou "<code>a+</code>" (voir tableau 6-2). Par défaut, elle a la valeur "<code>r</code>" (lecture seule). - L'option <code>encoding</code> peut valoir "<code>utf8</code>" ou <code>null</code>. Par défaut, elle vaut <code>null</code>, ce qui retourne le contenu du fichier sous forme de buffer d'octets.
<code>fs.readFileSync(filename, [options])</code>	Version synchrone de <code>fs.readFile()</code> . Le contenu du fichier est retourné par la fonction.

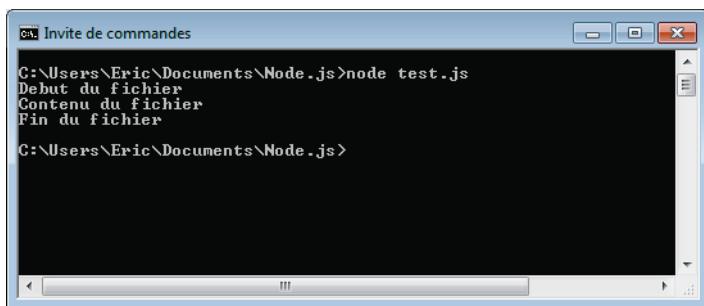
Par exemple, pour lire le contenu du fichier `fichier.txt` et l'afficher sous forme de texte, on utilise l'option `{ encoding : "utf8" }` et on écrit :

Lire le contenu du fichier sous forme de texte

```
var fs = require("fs");
fs.readFile("fichier.txt", { encoding : "utf8" }, function(err, data){
  console.log(data);
});
```

Figure 6-3

Lecture d'un fichier en mode asynchrone



Le contenu du fichier s'affiche maintenant sous forme de texte.

Lecture partielle du fichier

Lorsque l'on souhaite lire seulement une portion de fichier (au lieu de l'intégralité comme précédemment), on utilise les méthodes `fs.read()` et `fs.readSync()`.

Tableau 6–5 Méthodes de lecture de fichier à partir du descripteur de fichier

Méthode	Signification
<code>fs.read(fd, buffer, offset, length, position, callback)</code>	<p>Lit le fichier correspondant au descripteur <code>fd</code> (le fichier a été précédemment ouvert).</p> <p>Le paramètre <code>buffer</code> correspond à un buffer d'octets précédemment alloué (<code>new Buffer(size)</code>) qui contiendra les octets lus dans le fichier.</p> <p>Le paramètre <code>offset</code> indique où placer (dans le buffer) les octets lus. Indiquer <code>0</code> pour commencer au début du buffer.</p> <p>Le paramètre <code>length</code> indique le nombre d'octets à lire dans le fichier. Le nombre d'octets effectivement lus peut être moindre.</p> <p>Le paramètre <code>position</code> indique (à partir de <code>0</code>) l'indice à partir duquel les octets sont lus dans le fichier. Indiquer <code>null</code> pour lire à partir de la position courante.</p> <p>Le paramètre <code>callback</code> est une fonction de callback (appelée lorsque le fichier a été lu) de la forme <code>function(err, bytesRead, data)</code> dans laquelle <code>bytesRead</code> est le nombre d'octets effectivement lus, et <code>data</code> est le buffer lu (il correspond au paramètre <code>buffer</code>).</p>
<code>fs.readSync(fd, buffer, offset, length, position)</code>	<p>Version synchrone de <code>fs.read()</code>. Le contenu du fichier est retourné dans le buffer passé en paramètre.</p> <p>Le retour de la fonction est le nombre d'octets lus dans le fichier.</p>

Le nombre d'octets lus ne doit pas excéder le nombre d'octets restants dans le buffer, sinon une erreur se produit.

On souhaite lire le fichier `fichier.txt` précédent en utilisant la méthode `fs.read()`. Le contenu du fichier sera inséré dans un buffer de 100 octets alloué dynamiquement.

Utiliser la méthode `fs.read()`

```
var fs = require("fs");
fs.open("fichier.txt", "r", function(err, fd) {
  var buffer = new Buffer(100);
  fs.read(fd, buffer, 0, buffer.length, 0, function(err, bytesRead, data){
    console.log("Le nombre d'octets lus est : " + bytesRead);
    console.log(data);
  });
});
```

Le fichier est ouvert en mode lecture seule, puis son contenu est lu et placé dans un buffer réservé de 100 octets.

Figure 6–4

Lecture d'un fichier sous forme d'octets

```
C:\ Invite de commandes
C:\Users\Eric\Documents\Node.js>node test.js
Le nombre d'octets lus est : 52
<Buffer 44 65 62 75 74 20 64 75 20 66 69 63 68 69 65 72 0d 0a 43
75 20 64 75 20 66 69 63 68 69 65 72 0d 0a 46 69 6e 20 64 75 20 6
...
C:\Users\Eric\Documents\Node.js>
```

Seuls 52 octets ont été lus dans le fichier, ce qui correspond à sa taille actuelle.

Pour afficher le contenu du fichier sous forme de chaînes de caractères, il suffit d'utiliser la méthode `toString()` sur le buffer d'octets.

Afficher le buffer sous forme de chaînes de caractères

```
var fs = require("fs");
fs.open("fichier.txt", "r", function(err, fd) {
  var buffer = new Buffer(100);
  fs.read(fd, buffer, 0, buffer.length, 0, function(err, bytesRead, data){
    console.log("Le nombre d'octets lus est : " + bytesRead);
    console.log(data.toString());
  });
});
```

La méthode `data.toString()` transforme le buffer d'octets `data` en une chaîne de caractères.

Figure 6–5

Lecture d'un fichier sous forme de caractères

```
C:\ Invite de commandes
C:\Users\Eric\Documents\Node.js>node test.js
Le nombre d'octets lus est : 52
<Buffer 44 65 62 75 74 20 64 75 20 66 69 63 68 69 65 72 0d 0a 43
75 20 64 75 20 66 69 63 68 69 65 72 0d 0a 46 69 6e 20 64 75 20 6
...
C:\Users\Eric\Documents\Node.js>node test.js
Le nombre d'octets lus est : 52
Debut du fichier
Contenu du fichier
Fin du fichier,bF@??@      pcF?'@      XbF@j8?~@
C:\Users\Eric\Documents\Node.js>
```

Les caractères affichés en dehors de ceux lus dans le fichier correspondent au buffer initial qui comporte des octets aléatoires.

Écrire dans un fichier

Il existe différentes façons d'écrire dans un fichier :

- soit on écrase son précédent contenu, s'il existe ;
- soit on ajoute en fin de fichier de nouveaux octets.

Examinons ces deux possibilités.

Écraser le contenu du fichier

Les méthodes `fs.writeFile()` et `fs.writeFileSync()` permettent de remplacer l'intégralité d'un fichier par un nouveau contenu, ou de créer un nouveau fichier s'il n'existe pas. Le fichier est identifié par son nom uniquement (pas de descripteur de fichier).

Tableau 6–6 Méthodes d'écriture dans un fichier (érasrement)

Méthode	Signification
<code>fs.writeFile(filename, data, callback)</code>	Remplace le contenu du fichier par les octets (<code>Buffer</code> ou <code>String</code>) indiqués dans le paramètre <code>data</code> . Si <code>data</code> représente une chaîne, elle est encodée en UTF-8. La fonction de callback est de la forme <code>function(err)</code> et elle est appelée à la fin de l'écriture dans le fichier. Le fichier est créé s'il n'existe pas, sinon il est écrasé.
<code>fs.writeFileSync(filename, data)</code>	Version synchrone de <code>fs.writeFile()</code> .

Dans l'exemple qui suit, on écrit dans le fichier `fichier.txt` que l'on relit pour afficher son contenu.

Écrire dans un fichier, puis le relire

```
var fs = require("fs");
fs.writeFile("fichier.txt", "Bonjour éric", function(err) {
  fs.readFile("fichier.txt", { flag : "r", encoding : "utf8" }, function(err, data) {
    console.log(data);
  });
});
```

Figure 6–6
Écriture, puis lecture
d'un fichier



Le précédent contenu a été remplacé par le nouveau.

Ajouter des octets à la fin du fichier

L'ajout en fin de fichier s'effectue au moyen des méthodes `fs.appendFile()` et `fs.appendFileSync()`. Le fichier est créé s'il n'existe pas. Le fichier est identifié par son nom uniquement (pas de descripteur de fichier).

Tableau 6–7 Méthodes d'ajout en fin de fichier

Méthode	Signification
<code>fs.appendFile(filename, data, callback)</code>	Ajoute en fin de fichier les octets (<code>Buffer</code> ou <code>String</code>) représentés par le paramètre <code>data</code> . Si <code>data</code> représente une chaîne, elle est encodée en UTF-8. La fonction de callback est de la forme <code>function(err)</code> et elle est appelée à la fin de l'écriture dans le fichier. Le fichier est créé s'il n'existe pas.
<code>fs.appendFileSync(filename, data)</code>	Version synchrone de <code>fs.appendFile()</code> .

On utilise la méthode `fs.appendFile()` pour ajouter une chaîne de caractères à la fin du précédent fichier `fichier.txt`, puis on affiche le nouveau contenu du fichier.

Ajouter en fin de fichier, puis le relire

```
var fs = require("fs");
fs.appendFile("fichier.txt", " et bonjour tout le monde !", function(err) {
  fs.readFile("fichier.txt", { flag : "r", encoding : "utf8" }, function(err, data) {
    console.log(data);
  });
});
```

Figure 6–7
Ajout en fin de fichier

```
C:\ Invité de commandes
C:\Users\Eric\Documents\Node.js>node test.js
Bonjour éric et bonjour tout le monde !
C:\Users\Eric\Documents\Node.js>_
```

La chaîne "et bonjour tout le monde" est venue s'ajouter en fin de fichier.

Dupliquer un fichier

Il est possible de copier un fichier à un nouvel emplacement. Pour ce faire, on utilise les méthodes `fs.link()` ou `fs.linkSync()`.

Une fois la copie réalisée, les deux fichiers existent simultanément.

Tableau 6–8 Méthodes de copie de fichier à un nouvel emplacement

Méthode	Signification
<code>fs.link(srcpath, destpath, callback)</code>	Copie le fichier de l'emplacement source vers l'emplacement destination. La fonction de callback est de la forme <code>function(err)</code> dans laquelle <code>err</code> vaut <code>null</code> si le fichier a été correctement copié. Le fichier destination ne doit pas déjà exister avant la copie, sinon une erreur se produit.
<code>fs.linkSync(srcpath, destpath)</code>	Version synchrone de <code>fs.link()</code> . Provoque une exception si le fichier ne peut pas être copié.

Utilisons la méthode `fs.link()` pour dupliquer le fichier `fichier.txt` en `fichier2.txt`.

Dupliquer le fichier `fichier.txt` en `fichier2.txt`

```
var fs = require("fs");
fs.link("fichier.txt", "fichier2.txt", function(err) {
  if (err) console.log(err);
  else console.log("Fichier copié");
});
```

Supprimer un fichier

Une fois qu'on sait créer des fichiers et les mettre à jour, il reste à savoir les supprimer. Ceci est possible au moyen des méthodes `fs.unlink()` et `fs.unlinkSync()`.

Tableau 6–9 Méthodes de suppression de fichier

Méthode	Signification
<code>fs.unlink(path, callback)</code>	Supprime le fichier associé au chemin indiqué. La fonction de callback est de la forme <code>function(err)</code> dans laquelle <code>err</code> vaut <code>null</code> si le fichier a été correctement supprimé.
<code>fs.unlinkSync(path)</code>	Version synchrone de <code>fs.unlink()</code> . Provoque une exception si le fichier ne peut pas être supprimé.

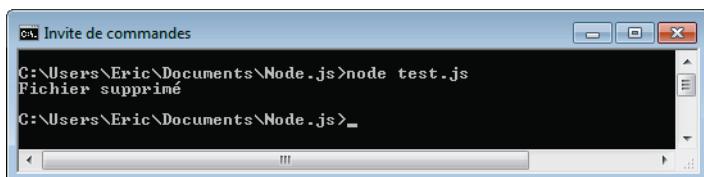
Utilisons la méthode `fs.unlink()` pour supprimer le fichier `fichier.txt`.

Supprimer le fichier fichier.txt

```
var fs = require("fs");
fs.unlink("fichier.txt", function(err) {
  if (err) console.log(err);
  else console.log("Fichier supprimé");
});
```

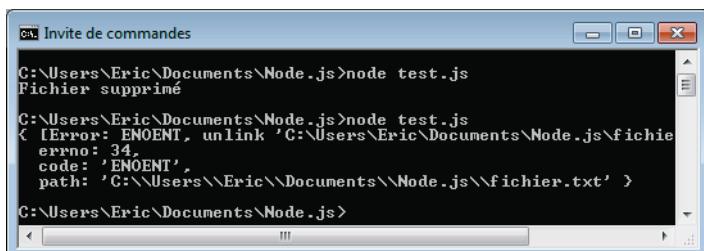
Lors de la première exécution de ce programme, le fichier est supprimé.

Figure 6–8
Suppression de fichier



Lors d'un second appel au programme, une erreur est affichée car le fichier n'existe plus.

Figure 6–9
Erreur lors d'une suppression
de fichier inexistant



Si on utilise la version synchrone de la méthode `fs.unlinkSync()`, il faut capturer l'exception.

Utiliser la version synchrone et capturer l'exception

```
var fs = require("fs");
try {
  fs.unlinkSync("fichier.txt");
  console.log("Fichier supprimé");
} catch(e) {
  console.log(e);
}
```

Dès qu'une exception se produit dans le bloc `try`, le bloc `catch` s'exécute et affiche l'exception produite.

Renommer un fichier ou un répertoire

Pour renommer un fichier ou un répertoire, on utilisera les méthodes `fs.rename()` et `fs.renameSync()`.

Tableau 6–10 Méthodes de modification d'un nom de fichier ou de répertoire

Méthode	Signification
<code>fs.rename(oldpath, newpath, callback)</code>	Renomme le chemin (fichier ou répertoire) <code>oldpath</code> en <code>newpath</code> . La fonction de callback est de la forme <code>function(err)</code> dans laquelle <code>err</code> vaut <code>null</code> si la modification de nom est effectuée. Si le chemin d'origine correspond à un fichier, et si le fichier de destination existe, ce dernier est écrasé. Si le chemin d'origine correspond à un répertoire, et si le répertoire de destination existe, rien n'est modifié.
<code>fs.renameSync(oldpath, newpath)</code>	Version synchrone de <code>fs.rename()</code> . Provoque une exception si la modification de nom ne peut être effectuée.

Renommer le fichier `fichier.txt` en `fichier2.txt`

```
var fs = require("fs");
fs.rename("fichier.txt", "fichier2.txt", function(err) {
  console.log("Fichier renommé");
});
```

Créer ou supprimer un répertoire

Il est possible de créer et de supprimer des répertoires.

Créer un répertoire

La création d'un nouveau répertoire s'effectue au moyen des méthodes `fs.mkdir()` ou `fs.mkdirSync()`.

Tableau 6–11 Méthodes de création d'un répertoire

Méthode	Signification
<code>fs.mkdir(path, callback)</code>	Crée le répertoire indiqué. La fonction de callback est de la forme <code>function(err)</code> dans laquelle <code>err</code> vaut <code>null</code> si le répertoire a été correctement créé.
<code>fs.mkdirSync(path)</code>	Version synchrone de <code>fs.mkdir()</code> . Provoque une exception si le répertoire ne peut pas être créé.

On peut créer un seul répertoire à la fois, mais pas un enchaînement de sous-répertoires.

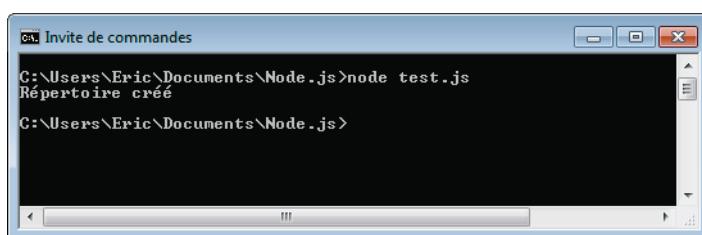
Dans l'exemple ci-dessous, on utilise ces méthodes pour créer le répertoire `newdir` dans le répertoire courant.

Créer le répertoire newdir dans le répertoire courant

```
var fs = require("fs");
fs.mkdir("newdir", function(err) {
  if (err) console.log(err);
  else console.log("Répertoire créé");
});
```

Une première exécution du programme crée le répertoire.

Figure 6–10
Création de répertoire



Une seconde exécution du programme provoque une erreur car le répertoire existe déjà.

Figure 6–11

Erreur lors de la création d'un répertoire déjà existant

```
C:\Users\Eric\Documents\Node.js>node test.js
Répertoire créé
C:\Users\Eric\Documents\Node.js>node test.js
C:\Error: EEXIST, mkdir 'C:\Users\Eric\Documents\Node.js\newdir',
errno: 42,
code: 'EEXIST',
path: 'C:\Users\Eric\Documents\Node.js\newdir' '
C:\Users\Eric\Documents\Node.js>
```

Supprimer un répertoire

On peut supprimer un répertoire à l'aide des méthodes `fs.rmdir()` et `fs.rmdirSync()`. Le répertoire doit être vide (ni fichier ni répertoire) sinon une erreur se produit.

Tableau 6–12 Méthodes de suppression d'un répertoire

Méthode	Signification
<code>fs.rmdir(path, callback)</code>	Supprime le répertoire indiqué. La fonction de callback est de la forme <code>function(err)</code> dans laquelle <code>err</code> vaut <code>null</code> si le répertoire a été correctement supprimé. Attention : le répertoire doit être vide (ni fichier ni répertoire), sinon il ne sera pas supprimé.
<code>fs.rmdirSync(path)</code>	Version synchrone de <code>fs.rmdir()</code> . Provoque une exception si le répertoire ne peut pas être supprimé.

Supprimer le répertoire newdir précédemment créé

```
var fs = require("fs");
fs.rmdir("newdir", function(err) {
  if (err) console.log(err);
  else console.log("Répertoire supprimé");
});
```

Figure 6–12

Suppression d'un répertoire

```
C:\Users\Eric\Documents\Node.js>node test.js
Répertoire supprimé
C:\Users\Eric\Documents\Node.js>
```

Lister tous les fichiers d'un répertoire

Il est possible de retourner la liste de tous les fichiers et répertoires d'un répertoire donné, en utilisant les méthodes `fs.readdir()` et `fs.readdirSync()`.

Tableau 6–13 Méthodes recherchant la liste des fichiers et répertoires d'un chemin donné

Méthode	Signification
<code>fs.readdir(path, callback)</code>	Appelle la fonction de callback de la forme <code>function(err, files)</code> dans laquelle <code>files</code> est un tableau des noms de fichiers et de répertoires du chemin indiqué.
<code>fs.readdirSync(path)</code>	Version synchrone de <code>fs.readdir()</code> . Le tableau <code>files</code> est retourné par la méthode.

Afficher la liste de tous les fichiers et répertoires du répertoire courant

```
var fs = require("fs");
fs.readdir(".", function(err, files) {
  console.log(files);
});
```

Le répertoire courant est symbolisé par "`..`", son parent par "`... .`".

Figure 6–13

Afficher les fichiers d'un répertoire

```
C:\Users\Eric\Documents\Node.js>node test.js
[ '404.html',
  'client.js',
  'client1.js',
  'code',
  'copy.js',
  'data',
  'eric.txt',
  'errorhandler.js',
  'favicon.ico',
  'fichier.txt',
  'fichier2.txt',
  'forms',
  'hello.js',
  'html',
  'index.html',
  'index2.html',
  'logger.js',
  'logo.png',
  'logo1.png',
  'logs.txt',
  'module0.js',
  'module1.js']
```

Tester l'existence d'un fichier ou d'un répertoire

Il est possible de tester l'existence d'un fichier ou d'un répertoire, à l'aide des méthodes `fs.exists()` et `fs.existsSync()`. Le fichier ou répertoire est identifié par son nom.

Tableau 6–14 Méthodes permettant de tester l'existence d'un fichier ou d'un répertoire

Méthode	Signification
<code>fs.exists(path, callback)</code>	Teste l'existence du fichier ou du répertoire indiqué, et appelle la fonction de callback de la forme <code>function(exists)</code> dans laquelle <code>exists</code> vaut <code>true</code> si le fichier ou répertoire existe, sinon <code>false</code> . Attention : la fonction de callback ne possède pas le paramètre <code>err</code> traditionnel.
<code>fs.existsSync(path)</code>	Version synchrone de <code>fs.exists()</code> . Retourne <code>true</code> si le chemin <code>path</code> existe, <code>false</code> sinon.

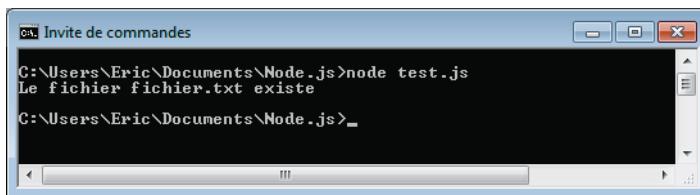
Utilisons ces méthodes pour vérifier si le fichier `fichier.txt` existe.

Indiquer si un fichier existe

```
var fs = require("fs");
fs.exists("fichier.txt", function(exists) {
  if (exists) console.log("Le fichier fichier.txt existe");
  else console.log("Le fichier fichier.txt n'existe pas");
});
```

Figure 6–14

Test de l'existence d'un fichier



Obtenir des informations sur un fichier ou un répertoire

Il est possible d'obtenir des informations sur un fichier ou un répertoire, par exemple :

- la taille du fichier ;
- si le chemin correspond à un répertoire ou à un fichier ;
- sa date de création et de dernière modification.

Pour cela, Node fournit les méthodes `fs.stat()` et `fs.statSync()` qui prennent en paramètres le nom du fichier ou du répertoire à analyser.

Tableau 6–15 Méthodes permettant d'obtenir des informations sur un fichier ou un répertoire

Méthode	Signification
<code>fs.stat(path, callback)</code>	Analyse le chemin indiqué (fichier ou répertoire) et appelle la fonction de callback de la forme <code>function(err, stat)</code> dans laquelle <code>stat</code> est un objet contenant les propriétés ou méthodes suivantes : - <code>stat.size</code> : taille en octets du fichier ; - <code>stat.ctime</code> : date de création ; - <code>stat.mtime</code> : date de dernière modification ; - <code>stat.isFile()</code> : retourne <code>true</code> si le <code>path</code> représente un fichier, sinon <code>false</code> ; - <code>stat.isDirectory()</code> : retourne <code>true</code> si le <code>path</code> représente un répertoire, sinon <code>false</code> .
<code>fs.statSync(path)</code>	Version synchrone de <code>fs.stat()</code> . Retourne l'objet <code>stat</code> .

Utilisons cette méthode pour obtenir des informations sur le fichier `fichier.txt`.

Afficher les informations du fichier fichier.txt

```
var fs = require("fs");
fs.stat("fichier.txt", function(err, stat) {
  console.log(stat);
  if (stat.isFile()) console.log("C'est un fichier");
  else console.log("C'est un répertoire");

});
```

Figure 6–15
Affichage d'informations sur un fichier

```
C:\Users\Eric\Documents\Node.js>node test.js
{ dev: 0,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  ino: 0,
  size: 31,
  atime: Tue Apr 08 2014 16:31:42 GMT+0200 (Paris, Madrid C\u00f4te d'Ivoire),
  mtime: Tue Apr 08 2014 16:32:30 GMT+0200 (Paris, Madrid C\u00f4te d'Ivoire),
  ctime: Tue Apr 08 2014 16:31:42 GMT+0200 (Paris, Madrid C\u00f4te d'Ivoire)
C'est un fichier

C:\Users\Eric\Documents\Node.js>_
```

Relier les fichiers et les streams

Un fichier peut être lu et écrit, tout comme un stream. Node associe un fichier à un stream en lecture et à un stream en écriture.

Node fournit les méthodes `fs.createReadStream(path)` et `fs.createWriteStream(path)` qui permettent de créer les streams en lecture et en écriture associés aux fichiers indiqués.

Tableau 6–16 Méthodes permettant d'obtenir des streams associés à un fichier

Méthode	Signification
<code>fs.createReadStream(path)</code>	Retourne le stream en lecture associé au fichier indiqué.
<code>fs.createWriteStream(path)</code>	Retourne le stream en écriture associé au fichier indiqué.

Une fois les streams obtenus à partir du nom de fichier, les opérations sur les streams étudiées au chapitre 5 sont disponibles.

Voyons comment les utiliser avec deux exemples.

Copier un fichier dans un autre au moyen d'un stream

On peut utiliser les streams pour copier un fichier dans un autre. Le fichier d'origine est associé au stream en lecture, tandis que le fichier destination est associé au stream en écriture. La méthode `readable.pipe(writable)` permet de copier un stream dans un autre.

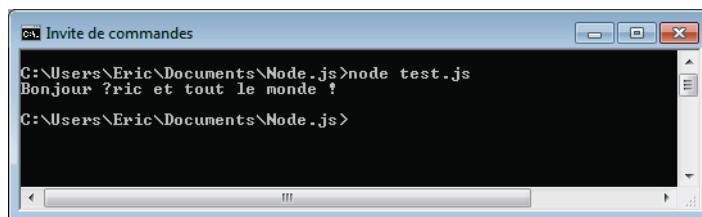
Copier le fichier fichier.txt dans fichier2.txt

```
var fs = require("fs");
var readstream = fs.createReadStream("fichier.txt");
var writestream = fs.createWriteStream("fichier2.txt");
readstream.pipe(writestream);

fs.readFile("fichier2.txt", function(err, data) {
  console.log(data.toString());
});
```

Figure 6–16

Associer un stream et un fichier



Copier les caractères entrés au clavier dans un fichier

Comme précédemment, on peut considérer que le stream en lecture est maintenant associé au clavier, tandis que le stream en écriture est le fichier destination. Ceci permet d'entrer des caractères au clavier qui sont directement transmis dans le fichier destination.

Dans l'exemple qui suit, on tape des caractères au clavier jusqu'à ce que l'utilisateur saisisse la chaîne "exit". Toutes les chaînes tapées sont insérées dans le fichier `fichier2.txt`, dont le contenu est affiché à la fin.

Copier les caractères entrés au clavier dans le fichier `fichier2.txt`

```
var fs = require("fs");
var readstream = process.stdin;
var writestream = fs.createWriteStream("fichier2.txt");
readstream.pipe(writestream);

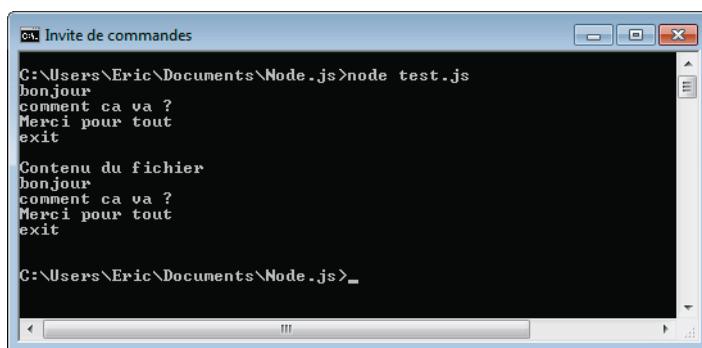
readstream.on("data", function(chunk) {
    chunk = chunk.toString().replace(/\r\n/g, "");
    if (chunk == "exit") {
        fs.readFile("fichier2.txt", function(err, data) {
            console.log("\nContenu du fichier");
            console.log(data.toString());
            writestream.end(); // fermer le stream en écriture
            readstream.pause(); // puis celui en lecture (dans cet ordre)
        });
    }
});
```

L'instruction `readstream.pause()` permet de quitter le mode de saisie. Mais elle ne peut fonctionner que si l'autre côté du stream (le stream en écriture, associé au fichier) est également fermé (au moyen de l'instruction `writestream.end()`).

Voyons un exemple d'utilisation. Lançons le programme et entrons des chaînes de caractères au clavier, puis tapons la chaîne "exit" pour terminer.

Figure 6-17

Sauvegarde des caractères tapés au clavier dans un fichier



Gestion des processus

Un programme Node consiste en l'exécution d'un seul processus, qui est la boucle de traitement des événements reçus. Tout traitement effectué dans le programme doit être le plus rapide possible, de façon à ne pas bloquer les autres événements qui surviennent en même temps.

Les traitements trop longs sont pénalisants pour le système et pour les utilisateurs. Il faut donc les décentraliser dans des processus externes, ce qui est possible dans Node grâce au module `child_process`.

Exécuter un processus grâce à la méthode `exec()` du module `child_process`

Le module `child_process` permet de gérer les processus dans Node. L'exécution d'un nouveau processus consiste à lancer une nouvelle commande qui s'exécutera dans le système. Deux sortes de commandes peuvent s'exécuter :

- les commandes système, telles que `dir` pour afficher la liste des fichiers du répertoire courant (`dir` sous Windows, `ls` dans un système Unix) ;
- les fichiers Node écrits par nous-mêmes ou d'autres développeurs. Ces commandes commencent par la commande `node` suivie du nom du fichier JavaScript à exécuter.

Le plus simple pour exécuter ces deux types de commandes est d'utiliser la méthode `exec()` du module `child_process`. D'autres méthodes existent (par exemple, la méthode `spawn()` apportant plus de flexibilité) et seront décrites dans une prochaine section.

Examinons les différentes manières d'utiliser la méthode `exec()`.

Exécuter une commande système en tant que nouveau processus

La méthode `exec()` du module `child_process` permet d'exécuter une commande système quelconque, dans un nouveau processus. Le résultat est récupéré dans une fonction de callback passée en paramètre de la méthode `exec()`.

Tableau 7-1 Méthode exec() d'exécution d'un nouveau processus

Méthode	Signification
<code>child_process.exec(cmd, callback)</code>	Exécute la commande <code>cmd</code> . À l'issue de l'exécution de la commande, la fonction de callback de la forme <code>function(err, stdout, stderr)</code> est appelée, dans laquelle : <ul style="list-style-type: none"> - le paramètre <code>err</code> indique une erreur éventuelle (<code>null</code> sinon) ; - le paramètre <code>stdout</code> représente une chaîne de caractères contenant la réponse lorsque la commande s'est bien exécutée ; - le paramètre <code>stderr</code> représente une chaîne de caractères contenant le message d'erreur lorsque la commande ne s'est pas exécutée correctement.

Utilisons la méthode `exec()` pour lister les fichiers d'extension `.js` du répertoire courant. La commande système correspondante est `dir *.js` sous un système Windows ou `ls *.js` sous un système Unix.

Lister les fichiers d'extension .js dans un second processus

```
var child_process = require("child_process");
child_process.exec("dir *.js", function(err, stdout, stderr){
  if (err) console.log(stderr);
  else console.log(stdout);
});
```

Figure 7-1

Exécution de la commande `dir *.js` dans un processus

```
C:\> Invite de commandes
C:\Users\Eric\Documents\Node.js>node test.js
Le volume dans le lecteur C s'appelle OS
Le num?ro de s?rie du volume est DAF9-41D1
R?pertoire de C:\Users\Eric\Documents\Node.js

20/07/2013 13:36      616 client.js
24/06/2013 13:18     1?111 client1.js
27/05/2013 14:58      0 copy.js
09/08/2013 19:02     305 errorhandler.js
06/08/2013 17:51      91 hello.js
09/08/2013 18:12     248 logger.js
08/08/2013 16:02     375 module0.js
08/08/2013 13:34     144 module1.js
08/08/2013 13:37     186 module2.js
22/07/2013 15:48     511 module3.js
22/07/2013 16:39     174 module4.js
22/07/2013 16:40     325 module5.js
06/04/2014 14:22     741 server.js
24/06/2013 13:21     898 server1.js
09/04/2014 17:00     209 test.js
15 fichier(s)          57934 octets
0 R?p(s)    4?920?070?144 octets libres
```

Si une erreur est survenue, on affiche le texte du message d'erreur contenu dans `stderr`, sinon on affiche le résultat de la commande contenu dans `stdout`.

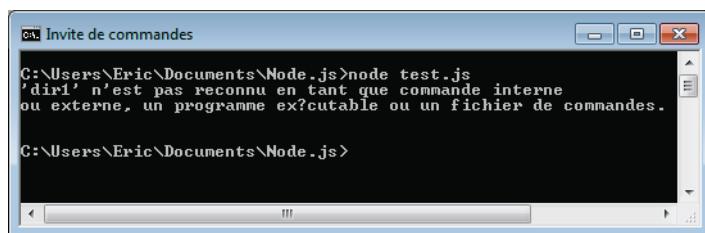
Si on exécute une commande inconnue (par exemple, `dir1` au lieu de `dir`), le texte du message d'erreur contenu dans `stderr` s'affiche.

Exécuter une commande inconnue (par exemple, `dir1` au lieu de `dir`)

```
var child_process = require("child_process");
child_process.exec("dir1 *.js", function(err, stdout, stderr){
    if (err) console.log(stderr);
    else console.log(stdout);
});
```

Figure 7–2

Exécution d'une commande inconnue dans un processus



Exécuter un fichier Node en tant que nouveau processus

Lorsque la commande indiquée dans la méthode `exec()` est `node`, cela permet d'exécuter un fichier Node en tant que nouveau processus.

Nous utilisons ici deux modules Node.

- Le fichier `test.js` est le module principal qui est lancé depuis la console du serveur.
- Le fichier `test2.js` est un nouveau processus lancé depuis le module principal. Il permet de ne pas bloquer le processus principal pendant son exécution.

Fichier `test.js` (module principal)

```
var child_process = require("child_process");
console.log("Début du processus principal");
child_process.exec("node test2.js", function(err, stdout, stderr){
    if (err) console.log(stderr);
    else console.log(stdout);
});
console.log("Fin du processus principal");
```

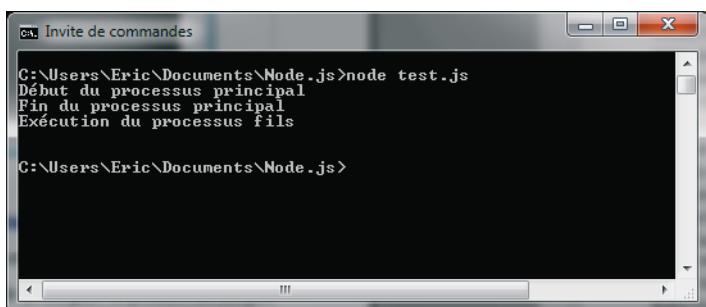
Le processus principal s'exécute, puis un processus fils est lancé par `child_process.exec()`. Ce processus fils s'exécutera indépendamment du processus père.

Fichier test2.js (processus fils)

```
console.log("Exécution du processus fils");
```

Le code des deux processus est volontairement simple afin de montrer l'enchaînement d'exécution des deux processus et leur indépendance.

Figure 7–3
Exécuter un fichier Node en tant que processus



Exécuter un processus grâce à la méthode `spawn()` du module `child_process`

La méthode `child_process.spawn()` est similaire à la méthode `exec()` vue précédemment, mais offre plus de souplesse. En effet, la méthode `exec()` récupère le résultat du processus fils en un seul bloc (dans les paramètres `stdout` ou `stderr` de la fonction de callback transmise en paramètre), alors que la méthode `spawn()` permet d'utiliser le résultat sous forme de stream et donc de fluidifier son utilisation.

De plus, la méthode `spawn()` permet des interactions entre le processus père et le processus fils, telles que l'échange d'informations entre eux ou l'arrêt du processus fils à la demande du père.

Remarquons que la méthode `spawn()` ne permet pas d'exécuter des commandes système (telles que `dir`, `ls`, etc.), mais elle peut exécuter la commande `node` pour démarrer un processus fils. C'est dans ce but que cette méthode sera utilisée ici.

Exécuter un fichier Node en tant que nouveau processus

Le principe est similaire à celui expliqué pour la méthode `exec()`. Toutefois, le résultat du processus fils est maintenant obtenu via un stream, au lieu d'être récupéré dans les paramètres de la fonction de callback.

Tableau 7-2 Méthode `spawn()` d'exécution d'un nouveau processus

Méthode	Signification
<code>child_process.spawn("node", args)</code>	Exécute la commande <code>node filename</code> afin d'exécuter le fichier Node en tant que processus fils. Le fichier à exécuter est indiqué dans le tableau <code>args</code> . Un objet <code>child</code> de classe <code>ProcessChild</code> est retourné par la méthode, permettant d'accéder aux streams en lecture <code>child.stdout</code> et <code>child.stderr</code> .

L'objet `child` retourné par la méthode `spawn()` permet d'accéder aux deux streams `child.stdout` et `child.stderr` qui contiendront les résultats du processus fils :

- le stream `child.stdout` contiendra la réponse du processus fils en cas de réussite ;
- le stream `child.stderr` contiendra la réponse du processus fils en cas d'erreur.

Utilisons la méthode `spawn()` afin d'exécuter un nouveau processus fils. Le processus père sera dans le fichier `test.js`, tandis que le fils sera dans le fichier `test2.js`.

Fichier test.js (processus père)

```
var child_process = require("child_process");
console.log("Début du processus principal");
var child = child_process.spawn("node", ["test2.js"]);
console.log("Fin du processus principal");

child.stdout.on("data", function(data) {
  console.log("Données : " + data);
});
```

Remarquez que la commande `node test2.js` est écrite en deux parties, car la méthode `spawn()` utilise le tableau `args` qui contient les arguments de la commande.

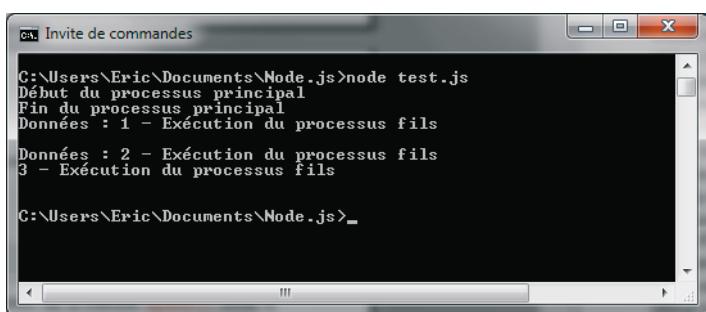
Le stream `child.stdout` reçoit les événements `data` envoyés sur le stream par le processus fils.

Fichier test2.js (processus fils)

```
console.log("1 - Exécution du processus fils");
console.log("2 - Exécution du processus fils");
console.log("3 - Exécution du processus fils");
```

Le processus fils effectue trois affichages, puis se termine. L'exécution est la suivante.

Figure 7–4
Utiliser la méthode `spawn()`



Le stream `child.stdout` reçoit les événements `data` qui permettent d'afficher les données reçues. Ces données sont émises par le processus fils au moyen des instructions `console.log()`. On voit sur l'exécution précédente que trois appels à la méthode `console.log()` provoquent le déclenchement de seulement deux événements `data` (d'où les deux blocs de données affichées).

Gérer les cas d'erreur dans le processus fils

Le processus fils précédent ne produit aucune erreur, donc le résultat de son exécution est dirigé vers le stream `child.stdout`. Toutefois, les cas d'erreur sont dirigés vers le stream `child.stderr`.

Produisons une erreur dans le processus fils et gérons l'événement `data` sur le stream `child.stderr`.

Fichier test.js (processus père)

```
var child_process = require("child_process");
console.log("Début du processus principal");
var child = child_process.spawn("node", ["test2.js"]);
console.log("Fin du processus principal");

child.stdout.on("data", function(data) {
  console.log("Données : " + data);
});

child.stderr.on("data", function(data) {
  console.log("Erreur : " + data);
});
```

On a simplement implémenté l'événement `data` sur le stream `child.stderr`.

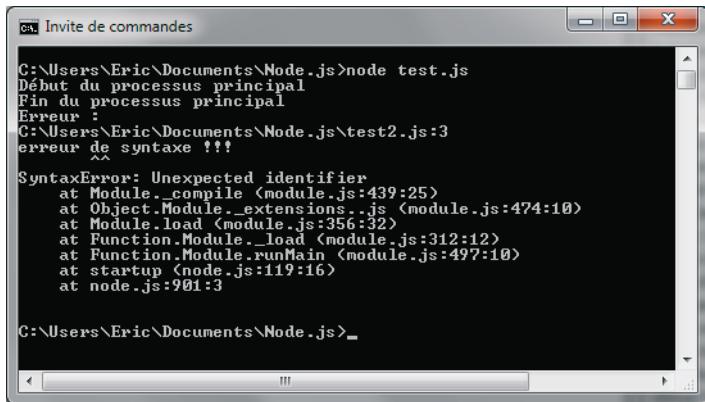
Fichier test2.js (processus fils)

```
console.log("1 - Exécution du processus fils");
console.log("2 - Exécution du processus fils");
erreur de syntaxe !!!
console.log("3 - Exécution du processus fils");
```

Une erreur de syntaxe est introduite dans le processus fils afin de provoquer une erreur et observer comment elle est prise en compte.

Figure 7–5

Erreur dans le processus fils



```
C:\Users\Eric\Documents\Node.js>node test.js
Début du processus principal
Fin du processus principal
Erreur :
C:\Users\Eric\Documents\Node.js\test2.js:3
erreur de syntaxe !!!
SyntaxError: Unexpected identifier
    at Module._compile (module.js:439:25)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3

C:\Users\Eric\Documents\Node.js>
```

Communication entre le processus père et le processus fils

La méthode `spawn()` permet de faire communiquer un processus père et un processus fils par l'intermédiaire des streams. Chaque processus (père ou fils) possède les streams `sdtin`, `stdout` et `stderr`, en lecture ou en écriture. Ces streams sont accessibles au moyen des objets `process` et `child`.

- L'objet `process` correspond au processus courant, c'est-à-dire soit le père (si on est dans le fichier JavaScript associé au père), soit le fils (si on est dans le fichier JavaScript associé au fils). Le stream `process.stdin` est en lecture, tandis que les streams `process.stdout` et `process.stderr` sont en écriture.
- L'objet `child` correspond au processus fils, c'est-à-dire celui créé par la méthode `spawn()` dans le fichier JavaScript du père. Le stream `child.stdin` est en écriture, tandis que les streams `child.stdout` et `child.stderr` sont en lecture (c'est l'inverse de l'objet `process`).

Utilisons ces streams pour communiquer entre le père et le fils. Le père transmet un nom au fils, par exemple "Eric", qui retourne alors la chaîne "Bonjour Eric" affichée par le père.

Fichier test.js (processus père)

```
var child_process = require("child_process");
var child = child_process.spawn("node", ["test2.js"]);

// envoi au fils de la chaîne "Eric"
child.stdin.write("Eric");

// récupération des éléments reçus du fils
child.stdout.on("data", function(data) {
    console.log(data.toString());
});

// traitement des erreurs (obligatoire)
child.stdout.on("error", function(data) {
    console.log(data.toString());
});
```

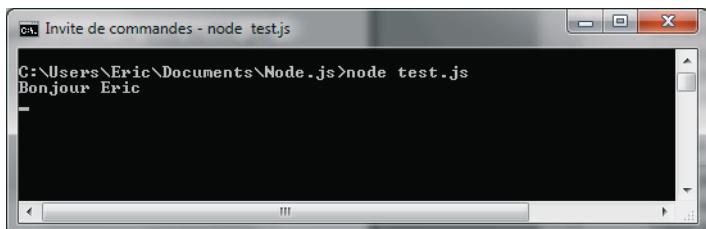
Fichier test2.js (processus fils)

```
// attente des données envoyées par le père
process.stdin.on("data", function(data) {
    var nom = data.toString();
    // insertion de la chaîne résultat dans le stream en écriture,
    // sur lequel le père est en attente
    process.stdout.write("Bonjour " + nom);
});
```

Dans le processus père, le fils reçoit les données du père sur `child.stdin`, et envoie des données au père via `child.stdout`.

Dans le processus fils, le fils reçoit les données du père sur `process.stdin`, et envoie les données au père via `process.stdout`.

Figure 7–6
Communication d'un processus père avec son fils



La chaîne "Eric" est envoyée au processus fils, qui la retourne en la préfixant de "Bonjour".

8

Gestion des connexions TCP

Ce chapitre aborde les connexions TCP(*Transmission Control Protocol*), qui permettent d'associer un stream à un port du serveur. Un port correspond à un numéro quelconque et sert à identifier de façon unique le programme qui sera associé à ce port sur ce serveur. Par exemple, si on associe le port 3000 du serveur à un programme Node, l'accès des utilisateurs au port 3000 activera des parties de code de notre programme, car celui-ci est à l'écoute des connexions des utilisateurs sur ce port.

On voit donc qu'une connexion TCP est un système plus élaboré qu'un stream. En effet, une connexion TCP peut être vue comme un stream associé à un port du serveur, alors qu'un simple stream est une abstraction plus générale, qui n'est associée à aucun port.

Enfin, lorsque des paquets de données sont transmis via une liaison TCP, il est garanti par le protocole TCP que ces paquets seront reçus dans l'ordre où ils ont été envoyés, sans perte d'informations.

Principe d'une connexion TCP

Une connexion TCP va consister en deux streams, l'un appelé « client » et l'autre appelé « serveur ». Chaque stream est en lecture et en écriture. La liaison entre ces deux streams correspond à la connexion TCP. Node permet de créer des streams en lecture et en écriture au moyen de la classe `stream.Duplex`. Toutefois, pour faciliter la gestion des connexions TCP, Node met à notre disposition le module `net` qui contient un ensemble de méthodes spécifiques permettant de gérer les clients TCP ou les serveurs TCP.

Bien que le serveur et le client soient chacun associés à un stream, Node fournit l'accès par programme au stream associé au client, mais pas à celui du serveur. Nous verrons que cela est amplement suffisant pour les besoins les plus courants.

La première étape lorsqu'on utilise une connexion TCP consistera à créer le programme du serveur, appelé ici « serveur TCP ». Le client TCP pourra lui-aussi être créé par programme, mais dans un premier temps nous utiliserons le programme Telnet qui fait office de client TCP.

Le programme Telnet

Le programme Telnet permet d'accéder aux ports TCP sur un serveur et ainsi d'exécuter les traitements disponibles sur chacun des ports TCP sur le serveur. Le traitement associé à un port correspond au programme du serveur TCP et sera étudié dans la prochaine section.

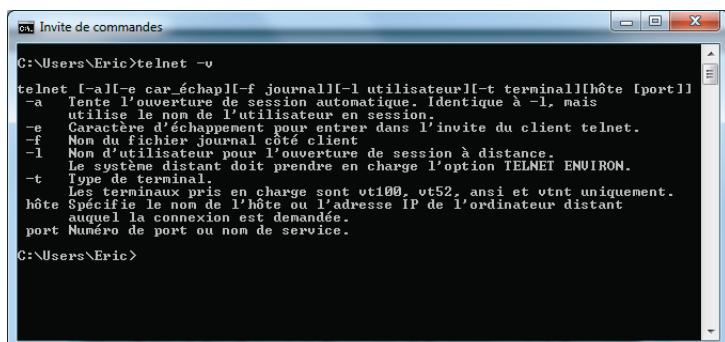
Le programme Telnet est accessible via la commande `telnet` tapée dans un interpréteur de commandes, disponible sous tous les systèmes d'exploitation. Toutefois, sous Windows, il est possible (voire probable) que celle-ci ne soit pas installée. Dans ce cas, il suffit, dans un interpréteur de commandes, de saisir les deux commandes suivantes permettant d'installer le programme Telnet correspondant.

Commandes Windows pour installer le programme Telnet

```
c:\> pkgmgr /iu:TelnetClient
c:\> pkgmgr /iu:TelnetServer
```

Une fois ces instructions tapées, il faudra redémarrer votre PC. La commande `telnet` sera alors accessible dans un interpréteur de commandes, par exemple en tapant l'instruction `telnet -v`.

Figure 8–1
Aide sur la commande telnet



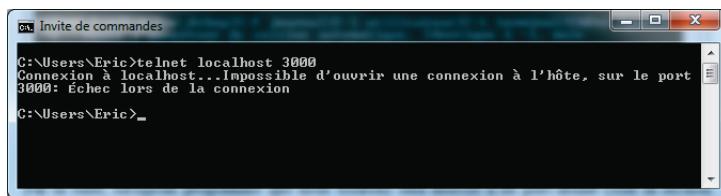
Par la suite, lorsqu'un programme que nous écrirons sera associé à un port donné (voir la section suivante), nous pourrons accéder au port TCP (et donc effectuer le traitement associé) en tapant simplement la commande `telnet` suivante dans un interpréteur de commandes (ici pour le port 3000).

Commande `telnet` exécutée sur le port 3000 du serveur local

```
telnet localhost 3000
```

La commande `telnet` précédente active le programme associé au port 3000 du serveur. Si aucun programme n'est associé à ce port sur le serveur (ce qui est le cas pour l'instant), un message d'erreur est affiché sur la console du serveur.

Figure 8–2
Erreur d'exécution avec
la commande `telnet`



Créer un serveur TCP

La création d'un serveur TCP nécessite le développement d'un programme qui sera exécuté lorsqu'un client se connectera sur ce port TCP.

Utiliser la méthode `net.createServer()`

Node fournit la méthode `net.createServer()` du module `net` permettant de créer le serveur TCP.

Tableau 8–1 Méthodes de création d'un serveur TCP

Méthode	Signification
<code>net.createServer([callback])</code>	Crée un serveur TCP (objet <code>server</code>), qui devra se mettre en attente des connexions des clients TCP sur un port du serveur, au moyen de l'instruction <code>server.listen(port)</code> . La fonction de callback indiquée en paramètre est optionnelle et elle est activée à chaque connexion d'un client TCP sur ce serveur. L'événement <code>connection</code> est également déclenché à chaque connexion de client TCP.

Tableau 8-1 Méthodes de création d'un serveur TCP (suite)

Méthode	Signification
<code>server.listen(port, [callback])</code>	Met le serveur TCP en attente de connexions d'éventuels clients, sur le port indiqué. La fonction de callback est optionnelle, elle est activée par Node lorsque le serveur est prêt à recevoir des connexions d'utilisateurs. L'événement <code>listening</code> est déclenché (une seule fois) dès que le serveur est prêt à recevoir ces connexions.

Création d'un serveur TCP à l'écoute du port 3000

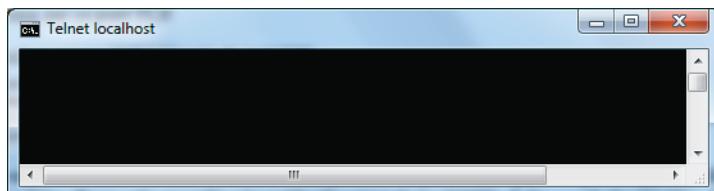
```
var net = require("net");
var server = net.createServer();
server.on("connection", function(socket) {
  console.log("Un client vient de se connecter");
});
server.listen(3000);
```

Nous insérons le module `net` contenant les méthodes gérant les connexions TCP. Le serveur TCP est créé par la méthode `net.createServer()`. L'objet `server` retourné par cette méthode est de classe `net.Server`, qui est elle-même une classe dérivant de `events.EventEmitter`. L'objet `server` pourra donc recevoir des événements, en particulier l'événement `connection` indiquant qu'un client TCP s'est connecté sur ce serveur TCP. Le traitement lors de la connexion d'un utilisateur va consister ici à afficher un message dans la console, indiquant qu'un client s'est connecté.

Enfin, le serveur doit être mis à l'écoute d'un port, ici le port 3000 que l'on peut supposer libre sur cette machine. Pour cela, on utilise la méthode `server.listen(3000)`. À l'issue de l'exécution de cette instruction, le programme se met en attente d'éventuelles connexions TCP sur ce port.

Pour tester ce programme, il faut d'abord lancer le serveur grâce à la commande `node test.js` saisie dans un interpréteur de commandes. Le serveur se met en attente de connexions de la part de clients, qui vont être simulés par le programme Telnet. On tape alors la commande `telnet localhost 3000` dans un nouvel interpréteur de commandes. La commande entrée n'apparaît plus à l'écran, qui reste vide (figure 8-3).

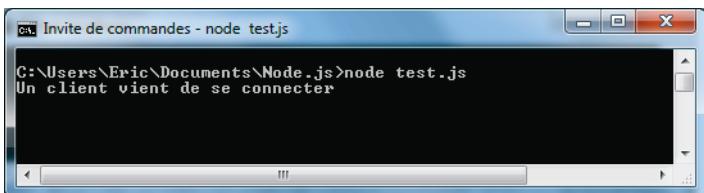
Figure 8-3
Lancement de la commande telnet



La fenêtre contenant le serveur affiche maintenant le texte indiquant qu'un client s'est connecté.

Figure 8–4

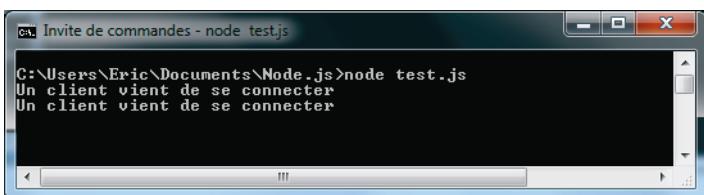
Connexion d'un client au serveur TCP



Si d'autres clients TCP se connectent (à partir d'autres interpréteurs de commandes en utilisant le programme Telnet), de nouveaux messages s'inscrivent dans la console du serveur indiquant ces connexions. Par exemple, si un second client se connecte :

Figure 8–5

Plusieurs connexions de clients au serveur TCP



Autre forme de la méthode `net.createServer()`

L'utilisation conjointe de la méthode `net.createServer()`, puis de l'événement `connection`, indiquant la connexion d'un utilisateur, est très fréquente car le traitement principal du serveur sera effectué lors de la gestion de cet événement.

Plutôt que d'écrire le code en deux parties comme précédemment (une partie pour la création du serveur, puis la gestion de l'événement `connection`), Node permet de regrouper ceci en une seule instruction. On utilise pour cela `net.createServer(callback)`, dans laquelle la fonction de callback correspond au traitement de l'événement `connection`.

Le code précédent pourrait alors s'écrire :

Gérer l'événement `connection` sous forme de callback

```
var net = require("net");
var server = net.createServer(function(socket) {
  console.log("Un client vient de se connecter");
});
server.listen(3000);
```

Remarquez que l'événement `connection` est encore déclenché et pourrait continuer à être géré dans notre programme.

Utiliser le paramètre `socket` pour indiquer au client qu'il est connecté

Pour l'instant, lorsqu'un client se connecte, aucun message n'apparaît. Il serait bon d'afficher un texte lui confirmant qu'il est effectivement connecté. Pour cela, Node fournit l'objet `socket` en paramètre de la fonction de traitement de l'événement `connection`. L'objet `socket` est un stream utilisable en lecture et en écriture, correspondant au stream associé au client. L'instruction `socket.write()` permet donc d'écrire sur ce stream.

Prévenir le client qu'il est connecté

```
var net = require("net");
var server = net.createServer();
server.on("connection", function(socket) {
  console.log("Un client vient de se connecter");
  socket.write("Connexion au serveur réussie");
});
server.listen(3000);
```

Nous avons simplement utilisé l'instruction `socket.write()`, exécutée lors de la connexion d'un client sur le serveur. Lorsqu'un client se connecte au serveur au moyen de Telnet, un message s'affiche maintenant dans la fenêtre.

Figure 8–6
Affichage d'une information de connexion



Utiliser l'événement `listening` pour indiquer que le serveur a démarré

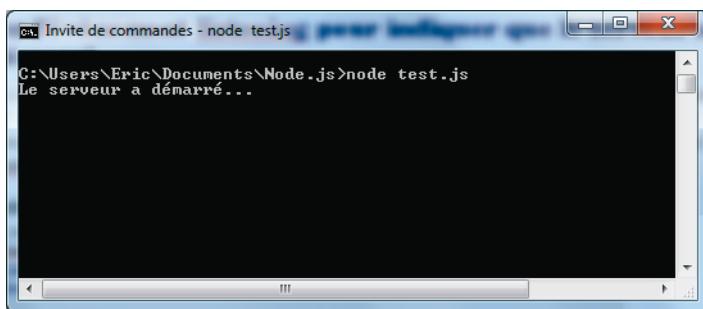
Lorsqu'on utilise l'instruction `server.listen(3000)`, on demande à démarrer le serveur en écoutant le port 3000. Cette instruction de démarrage prend un certain temps (quelques millisecondes) avant que le serveur soit réellement démarré. On est prévenu de son démarrage effectif en recevant l'événement `listening` sur l'objet `server`.

Utiliser l'événement listening

```
var net = require("net");
var server = net.createServer();
server.on("connection", function(socket) {
  console.log("Un client vient de se connecter");
  socket.write("Connexion au serveur réussie");
});
server.on("listening", function() {
  console.log("Le serveur a démarré...");
});
server.listen(3000);
```

Figure 8–7

Affichage d'une information de démarrage du serveur



Remarquons que la méthode `listen(port, callback)` possède un paramètre `callback` qui peut être utilisé comme l'événement `listening`. La fonction de callback est appelée par Node lorsque le serveur est prêt à recevoir des connexions des utilisateurs, ce qui est le cas lorsque l'événement `listening` est déclenché.

On pourrait donc écrire le code précédent de la façon suivante.

Utiliser une fonction de callback dans la méthode `listen()`

```
var net = require("net");
var server = net.createServer();
server.on("connection", function(socket) {
  console.log("Un client vient de se connecter");
  socket.write("Connexion au serveur réussie");
});
server.listen(3000, function() {
  console.log("Le serveur a démarré...");
});
```

Méthodes associées à un serveur TCP

L'objet `server` retourné par `net.createServer()` possède un certain nombre de méthodes utilitaires, comme la méthode `server.listen()` vue précédemment. Les principales méthodes sont listées dans le tableau 8-2.

Tableau 8-2 Méthodes de gestion du serveur TCP

Méthode	Signification
<code>server.listen(port, [callback])</code>	Met le serveur en attente sur le port indiqué. La fonction de callback est facultative et permet d'effectuer un traitement lorsque le serveur est prêt à recevoir les requêtes des clients TCP. L'événement <code>listening</code> est également envoyé sur l'objet <code>server</code> pour indiquer que le serveur est prêt.
<code>server.close([callback])</code>	Permet de ne plus traiter les requêtes des nouveaux clients TCP qui veulent se connecter au serveur. Les clients connectés préalablement restent connectés. La fonction de callback est facultative et elle est appelée lorsque toutes les connexions sont fermées des deux côtés (client et serveur). La connexion est fermée côté client au moyen de <code>socket.end()</code> (voir la section « Crée un client TCP »). L'événement <code>close</code> est émis au même moment que lorsque la fonction de callback est exécutée.
<code>server.address()</code>	Retourne un objet <code>{ port, family, address }</code> contenant entre autres le port et l'adresse IP du serveur. Remarquons que cet objet n'est disponible qu'après l'émission de l'événement <code>listening</code> indiquant que le serveur est prêt.
<code>server.getConnections(callback)</code>	La fonction de callback est ici obligatoire, elle est de la forme <code>function(err, count)</code> . Le paramètre <code>count</code> indique le nombre de clients TCP actuellement connectés sur le serveur.
<code>server.maxConnections</code>	Propriété indiquant le nombre maximal de clients TCP autorisés en même temps. La valeur par défaut est <code>undefined</code> , signifiant pas de limite. Il faut modifier cette propriété pour limiter l'accès au serveur.

Dans l'exemple qui suit, on affiche les connexions sur le serveur. Dès que deux connexions sont effectuées, le serveur est fermé. Toute autre connexion ultérieure est inopérante. Les connexions existantes ne sont pas perdues.

Fermer le serveur lorsque deux clients sont connectés

```
var net = require("net");
var server = net.createServer();
server.on("connection", function(socket) {
  console.log("Un client vient de se connecter");
  socket.write("Connexion au serveur réussie");
```

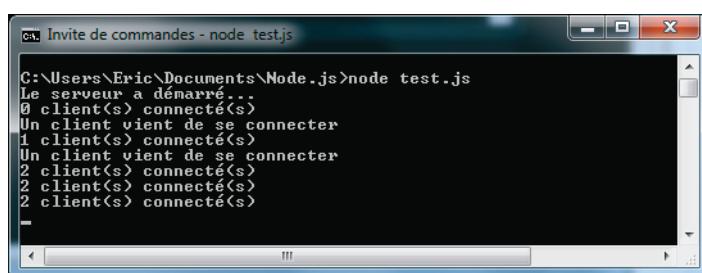
```
});  
server.on("listening", function() {  
    console.log("Le serveur a démarré...");  
});  
server.listen(3000);  
  
var server_closed= false;  
  
setInterval(function() {  
    server.getConnections(function(err, count) {  
        console.log(count + " client(s) connecté(s)");  
        if (count >= 2) {  
            if (!server_closed) {  
                server_closed = true;  
                server.close(function() {  
                    console.log("Le serveur est arrêté...");  
                });  
            }  
        }  
    },  
}, 5000);
```

La méthode `setInterval()` permet de récupérer régulièrement les informations de connexion sur le serveur. Au bout de deux connexions, on ferme le serveur, qui ne peut plus recevoir de nouvelles connexions. Les deux connexions en cours ne sont pas perdues.

La variable interne `server_closed` sert à mémoriser si le serveur a effectué l'instruction `server.close()`, de façon à ne pas la réexécuter, sinon une erreur se produit.

Figure 8–8

Connexion et déconnexion
de clients au serveur



Remarquons que lorsque deux clients sont connectés, ils le restent tant qu'ils n'ont pas effectué eux-mêmes l'opération de déconnexion. Ceci peut se faire via la méthode `socket.end()`, qui ferme la socket. Si toutes les sockets sont fermées, la fonction de callback indiquée dans la méthode `server.close(callback)` est appelée et l'événement `close` est déclenché.

Modifions le précédent programme pour prendre en compte ces instructions.

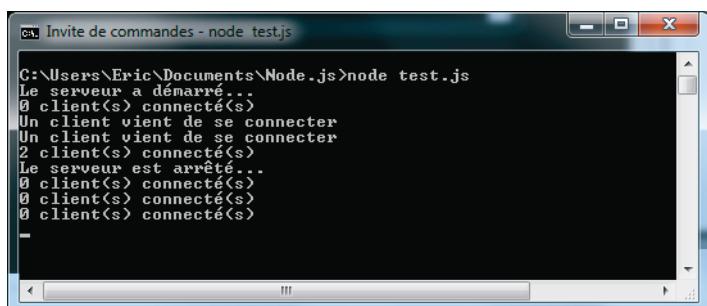
Fermer les sockets lorsque deux clients se sont inscrits

```
var net = require("net");
var server = net.createServer();
var sockets = [];
server.on("connection", function(socket) {
  console.log("Un client vient de se connecter");
  socket.write("Connexion au serveur réussie");
  sockets.push(socket);
});
server.on("listening", function() {
  console.log("Le serveur a démarré... ");
});
server.listen(3000);

var server_closed = false;

setInterval(function() {
  server.getConnections(function(err, count) {
    console.log(count + " client(s) connecté(s)");
    if (count >= 2) {
      if (!server_closed) {
        server_closed = true;
        server.close(function() {
          console.log("Le serveur est arrêté... ");
        });
        sockets.forEach(function(socket, i) {
          socket.write("\n\rFin de la connexion");
          socket.end();
        });
      }
    }
  });
}, 5000);
```

Figure 8–9
Arrêt provoqué du serveur



Lors de chaque événement `connection`, on mémorise la socket dans un tableau `sockets`, de façon à l'utiliser par la suite. Lorsque le serveur exécute `server.close()`, les sockets sont donc également clôturées par `socket.end()`.

Les clients TCP ayant fermé leur socket respective, ils ne sont plus vus connectés par le serveur. De plus, si on essaie de se connecter, le serveur va refuser car il est fermé.

Événements associés à un serveur TCP

Selon le cycle de vie du serveur TCP, différents événements sont produits sur l'objet `server` retourné par `net.createServer()`.

Tableau 8–3 Événements gérés par le serveur TCP

Événement	Signification
<code>listening</code>	Le serveur est en attente des connexions des clients TCP.
<code>connection</code>	Une connexion au serveur vient d'être effectuée par un client TCP.
<code>close</code>	Le serveur a effectué <code>server.close()</code> et chacun des clients a également clôturé sa connexion avec lui.
<code>error</code>	Une erreur s'est produite (par exemple, si on essaie de se connecter sur un port déjà occupé par un autre programme).

Dans les pages précédentes, nous avons utilisé les événements `listening`, `connection` et `close`. Utilisons maintenant l'événement `error` afin de montrer sa fonction.

Supposons que nous souhaitions créer deux serveurs dans notre programme, associés au même port 3000. Le second serveur produit une erreur du fait qu'un serveur est déjà attaché au port 3000.

Le programme, sans utiliser l'événement `error`, s'écrit de la façon classique.

Créer deux serveurs associés au port 3000 sans gérer les erreurs

```
var net = require("net");
var server = net.createServer();
server.on("listening", function() {
  console.log("Le serveur a démarré...");
});
server.listen(3000);

var server2 = net.createServer();
```

```

server2.on("listening", function() {
  console.log("Le serveur a démarré...");
});
server2.listen(3000);

```

Figure 8-10

Erreur lors du démarrage d'un serveur

```

C:\Users\Eric\Documents\Node.js>node test.js
Le serveur a démarré...

events.js:72
  throw er; // Unhandled 'error' event
          ^
Error: listen EADDRINUSE
    at errnoException (net.js:884:11)
    at Server._listen2 (net.js:1022:14)
    at listen (net.js:1044:10)
    at Server.listen (net.js:1110:5)
    at Object.<anonymous> (C:\Users\Eric\Documents\Node.js\test.js:1:10)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)

C:\Users\Eric\Documents\Node.js>

```

Le premier serveur démarre, mais le second produit une erreur qui termine le programme. Le premier serveur, qui avait pourtant démarré, est bien sûr arrêté vu que le programme qui le contient s'est arrêté.

Améliorons le programme afin de gérer les cas d'erreurs et de ne pas produire d'erreur pouvant terminer le programme.

Utiliser l'événement error dans le programme

```

var net = require("net");
var server = net.createServer();
server.on("listening", function() {
  console.log("Le serveur a démarré...");
});
server.on("error", function() {
  console.log("Le premier serveur a produit une erreur");
});
server.listen(3000);

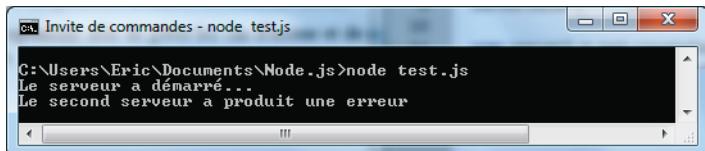
var server2 = net.createServer();
server2.on("listening", function() {
  console.log("Le serveur a démarré...");
});
server2.on("error", function() {
  console.log("Le second serveur a produit une erreur");
});
server2.listen(3000);

```

Nous gérons l'événement `error` sur chacun des deux serveurs, même si dans notre programme c'est le second serveur qui produit l'erreur.

Figure 8–11

Prise en compte de l'erreur lors du démarrage du serveur



Grâce à la gestion de l'événement `error`, le programme n'est plus arrêté. Bien que le second serveur ait produit une erreur, le premier serveur reste opérationnel.

Créer un client TCP

Nous avons vu dans les pages précédentes comment créer un serveur TCP. Nous avons également utilisé un ou plusieurs clients TCP en nous servant du programme Telnet fourni sur les principaux systèmes d'exploitation.

Le but ici va être de créer nous-mêmes le programme du client TCP, sans nous servir de Telnet. Node fournit différentes méthodes permettant de créer facilement le programme du client TCP que l'on souhaite.

Utiliser la méthode `net.connect(port)`

Node fournit la méthode `net.connect(port)` du module `net` permettant de créer un client TCP connecté à un port du serveur. Le serveur TCP doit avoir été démarré sur ce port. Si l'on cherche à se connecter à un serveur qui n'est pas en attente sur ce port, une erreur se produit.

Tableau 8–4 Méthode de création d'un client TCP

Méthode	Signification
<code>net.connect(port, [host], [callback])</code>	Crée un client TCP associé au port indiqué sur le serveur <code>host</code> . Le serveur TCP doit préalablement exister et être en attente sur ce port. Le paramètre <code>host</code> est facultatif et représente le nom ou l'adresse IP du serveur. S'il n'est pas indiqué, il vaut " <code>localhost</code> ". La fonction de callback indiquée en paramètre est optionnelle et activée à la fin de la connexion du client TCP. La méthode <code>net.connect()</code> retourne un objet de classe <code>net.Socket</code> , correspondant au stream associé au client TCP, utilisable en lecture et en écriture.

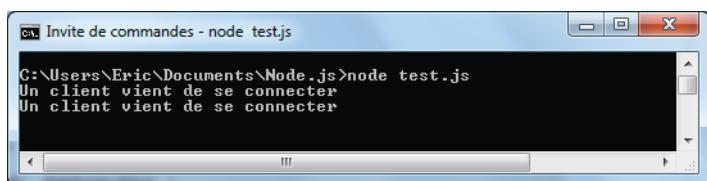
Création d'un serveur et d'un client TCP sur le port 3000

```
var net = require("net");
var server = net.createServer(function(socket) {
  console.log("Un client vient de se connecter");
});
server.listen(3000);

var client1 = net.connect(3000);
var client2 = net.connect(3000);
```

Le serveur est préalablement créé avant les clients. Nous créons ici deux clients TCP, qui se connectent sur le port 3000 du serveur.

Figure 8–12
Connexions de clients
au serveur



Le serveur reçoit ces deux connexions et active la fonction de callback pour chacune d'elles. Il est également possible de se connecter via Telnet, en plus des deux clients TCP créés dans notre programme. La connexion par Telnet sera automatiquement reçue par le programme du serveur.

Dissocier le programme du client et celui du serveur

Le code précédent est opérationnel, mais dans la réalité le code du client et celui du serveur sont dans des fichiers différents. On peut le faire ici en créant deux fichiers, l'un `server.js` et l'autre `client.js`, qui seront tous deux activés par la commande `node`.

Fichier server.js

```
var net = require("net");
var server = net.createServer(function(socket) {
  console.log("Un client vient de se connecter");
});
server.listen(3000);
```

Fichier client.js

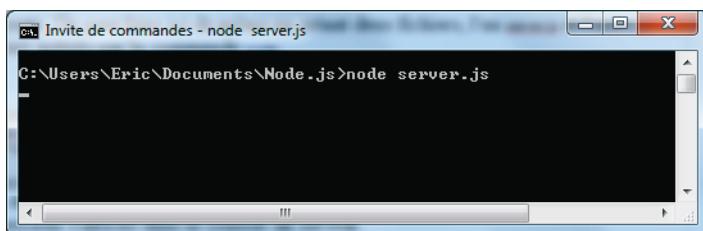
```
var net = require('net');
var client = net.connect(3000);
```

Pour tester ces deux programmes, il suffit de lancer le programme du serveur, puis celui du client. Pour ce faire, on utilise deux interpréteurs de commandes contenant chacun un des programmes. Si le programme du client est lancé plusieurs fois (dans des interpréteurs de commandes différents), chacune des connexions s'affiche dans la console du serveur.

Lançons en premier le serveur.

Figure 8–13

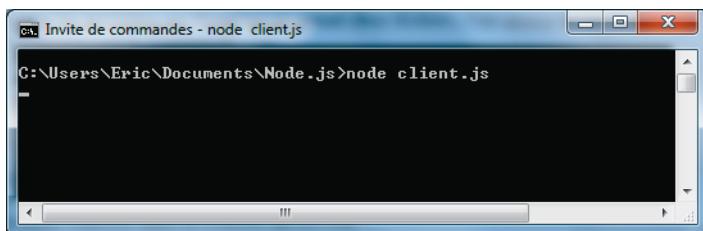
Un serveur autonome dans un seul fichier Node



Puis connectons-nous en lançant le client.

Figure 8–14

Un client autonome dans un seul fichier Node



Suite à cette connexion, le serveur affiche maintenant ce message.

Figure 8–15

Connexion d'un client sur le serveur



Ceci montre que le serveur et le client peuvent communiquer par TCP, même dans des programmes situés dans des fichiers différents. Si on lance plusieurs autres clients, le serveur les affiche dans sa console.

Gérer la déconnexion d'un client TCP au serveur

Arrêtons maintenant le programme du client en fermant sa fenêtre. La liaison avec le serveur s'interrompt, mais produit une erreur du serveur, l'arrêtant définitivement en perdant la connexion avec tous les autres clients.

Figure 8–16

Erreur lors de la déconnexion d'un client

```
C:\ Invite de commandes
C:\Users\Eric\Documents\Node.js>node server.js
Un client vient de se connecter
Un client vient de se connecter
events.js:72
    throw er; // Unhandled 'error' event
Error: read ECONNRESET
    at errnoException (net.js:884:11)
    at TCP.onread (net.js:539:19)
C:\Users\Eric\Documents\Node.js>
```

Ceci est évidemment rédhibitoire dans un cas réel, car un serveur ne peut pas s'arrêter dès qu'un client s'interrompt. Pour cela, il suffit de gérer l'événement `error` sur la socket associée au client. Le programme du serveur devient :

Gérer l'erreur sur une socket dans le fichier server.js

```
var net = require("net");
var server = net.createServer(function(socket) {
  console.log("Un client vient de se connecter");
  socket.on("error", function() {
    console.log("error");
  });
});
server.listen(3000);
```

La gestion de l'événement `error` ne peut pas être réalisée dans le fichier `client.js` car on suppose qu'on ferme la fenêtre du client pour clôturer la connexion. Ainsi, le traitement d'erreur qui serait inséré dans ce fichier ne serait jamais pris en compte, vu que ce programme est terminé.

L'arrêt d'un client ne provoque plus l'arrêt brutal du serveur comme précédemment, mais plutôt l'affichage d'un message d'erreur (ici, "error").

Figure 8–17

Gestion d'erreur lors de la déconnexion d'un client

```
C:\ Invite de commandes - node server.js
C:\Users\Eric\Documents\Node.js>node server.js
Un client vient de se connecter
error
Un client vient de se connecter
-
```

Le serveur peut donc continuer à recevoir de nouvelles connexions, même après l'arrêt d'un client.

Remarquons que lorsque l'événement `error` survient sur la socket, il est immédiatement suivi de l'événement `close` qui indique la terminaison du stream associé à celle-ci.

Communication entre le serveur TCP et un client TCP

Un client TCP correspond à un stream accessible en lecture et en écriture. Il peut donc être lu ou écrit autant que souhaité. Il peut également être fermé, comme tout stream en écriture, et être mis en pause, comme tout stream en lecture (voir chapitre 5, « Gestion des streams »).

Gérer le flux entre le serveur et le client

Pour illustrer ces principes, nous souhaitons écrire un serveur et un client qui communiquent entre eux. Le serveur affiche l'heure de façon périodique sur chacun des clients qui se connectent.

Ainsi, une fois que le serveur a été démarré et qu'un client s'est connecté, la fenêtre du serveur est la suivante.

Figure 8–18

Démarrage du serveur et connexion d'un client

```
c:\> Invite de commandes - node server.js
C:\Users\Eric\Documents\Node.js>node server.js
Le serveur est démarré...
Un client vient de se connecter
```

La fenêtre du client affiche l'heure de façon régulière (ici, toutes les deux secondes).

Figure 8–19

Affichage de l'heure sur le client

```
c:\> Invite de commandes - node client.js
C:\Users\Eric\Documents\Node.js>node client.js
Mon Jun 24 2013 17:08:23 GMT+0200 <Paris, Madrid <heure d'été>>
Mon Jun 24 2013 17:08:25 GMT+0200 <Paris, Madrid <heure d'été>>
Mon Jun 24 2013 17:08:27 GMT+0200 <Paris, Madrid <heure d'été>>
Mon Jun 24 2013 17:08:29 GMT+0200 <Paris, Madrid <heure d'été>>
```

Les programmes du serveur et du client sont mis dans deux fichiers séparés afin de s'exécuter de façon distincte.

Fichier server.js

```
var net = require("net");
var server = net.createServer(function(socket) {
    console.log("Un client vient de se connecter");

    var timer = setInterval(function() {
        socket.write((new Date()).toString());
    }, 2000);

    socket.on("error", function() {
        console.log("error");
        clearInterval(timer);
    });
});

server.on("listening", function() {
    console.log("Le serveur est démarré...");
});

server.listen(3000);
```

Lors de la connexion d'un client, un timer est mis en route de façon périodique grâce à la méthode `setInterval()`. Nous récupérons l'heure actuelle que nous transformons en chaîne de caractères par la méthode `toString()`. Puis nous écrivons cette chaîne sur la socket associée au client, au moyen de l'instruction `socket.write()`.

En cas d'erreur, par exemple par la fermeture d'un client, nous arrêtons le timer associé, mémorisé dans une variable locale `timer`.

Le programme du client est le suivant.

Fichier client.js

```
var net = require("net");
var client = net.connect(3000);
client.setEncoding("utf8");

client.on("data", function(chunk) {
    console.log(chunk);
});
```

Nous nous connectons d'abord au serveur via l'instruction `net.connect()`. Nous indiquons ensuite que le client affichera les chaînes de caractères en UTF-8 au moyen de

l'instruction `client.setEncoding("utf8")`. Puis, le client se met en attente des paquets de données au moyen de l'événement `data`, dont le traitement affiche les données reçues.

Gérer l'arrêt du serveur

L'arrêt d'un client ne provoque pas d'erreur sur le serveur, car l'événement `error` est traité sur la socket associée au client. Toutefois, l'arrêt du serveur provoque l'arrêt de tous les clients, ce qui est normal pour l'instant car ils ont perdu la connexion au serveur.

Pour éviter ce désagrément, il faudrait que les clients puissent se reconnecter au serveur en cas de perte de la connexion. Le client reçoit dans ce cas l'événement `error`. Il suffit lors de la réception de cet événement, que le client tente une nouvelle connexion sur le serveur. Le script du client est légèrement modifié (le script du serveur n'est pas modifié).

Fichier client.js

```
var net = require("net");
function connect() {
    var client = net.connect(3000);
    client.setEncoding("utf8");

    client.on("data", function(chunk) {
        console.log(chunk);
    });

    client.on("error", function() {
        console.log("error on client");
        connect();
    });
}
connect();
```

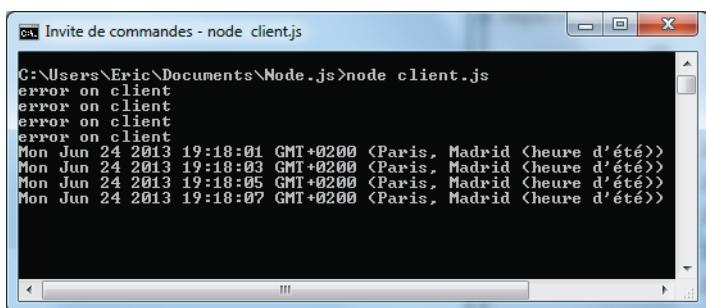
Nous avons défini une fonction `connect()`, appelée immédiatement après sa définition, mais qui pourra également être appelée en cas d'erreur, dans le traitement de l'événement `error`.

Remarquons que l'événement `error` sur la socket est traité à deux endroits : dans le programme du serveur et dans le programme du client. Chacun de ces traitements permet de tenir compte du fait que soit le serveur, soit le client est arrêté, et ainsi de ne pas rester dans une impasse.

Avec ces quelques modifications, on peut même démarrer le client avant le serveur. Le client se connectera automatiquement dès que le serveur sera lancé.

Par exemple, lançons le client, puis le serveur. La console du client s'affiche de la façon suivante.

Figure 8–20
Connexion d'un client
au serveur



Le client tente de se connecter, mais tombe dans le cas d'erreur. Il est alors automatiquement relancé, tant que le cas d'erreur se produit. Lorsque le serveur est lancé, l'erreur n'est plus produite et le traitement normal peut s'effectuer.

Une autre forme d'écriture du fichier `client.js` est possible. Plutôt que d'appeler la fonction `connect()` directement en fin de script, on l'appelle après sa définition, sur la même ligne.

Fichier `client.js`

```
var net = require("net");
(function connect() {
    var client = net.connect(3000);
    client.setEncoding("utf8");

    client.on("data", function(chunk) {
        console.log(chunk);
    });

    client.on("error", function() {
        console.log("error on client");
        connect();
    });
})();
```

Cette façon d'écrire est assez courante et vous pouvez la trouver fréquemment employée dans des bibliothèques JavaScript.

Exemple : communication entre plusieurs clients TCP via un serveur TCP

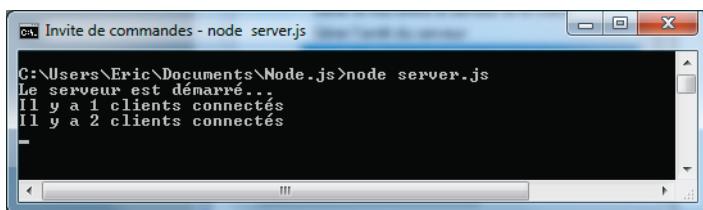
On souhaite maintenant écrire un programme de communication (chat) entre utilisateurs via un serveur. Chaque utilisateur sera associé à un client TCP. Les utilisateurs peuvent saisir des messages qui, une fois validés en appuyant sur la touche Entrée, s'affichent sur les consoles de tous les autres utilisateurs. Un utilisateur peut fermer sa connexion en introduisant la chaîne "exit" ou en clôturant le programme. S'il se reconnecte, il participe de nouveau au système d'échange de messages.

Dès qu'un utilisateur se connecte ou se déconnecte, le serveur affiche le nombre d'utilisateurs connectés à cet instant.

Lorsque le serveur a démarré et que deux utilisateurs se sont connectés, ce message apparaît.

Figure 8–21

Connexion de plusieurs clients au serveur

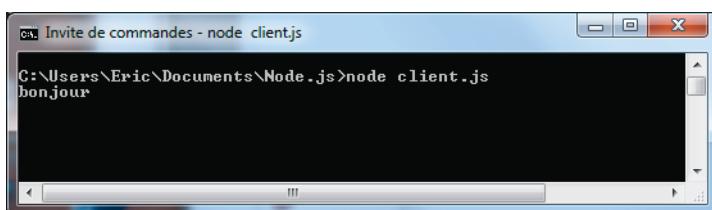


```
C:\Users\Eric\Documents\Node.js>node server.js
Le serveur est démarré...
Il y a 1 clients connectés
Il y a 2 clients connectés
```

Dans la fenêtre du premier utilisateur, on tape la chaîne "Bonjour".

Figure 8–22

Lecture de caractères au clavier par le client

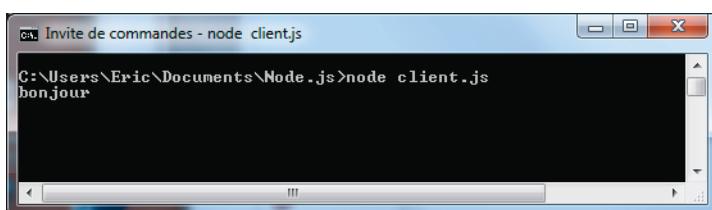


```
C:\Users\Eric\Documents\Node.js>node client.js
bonjour
```

Celle-ci se retrouve dans la fenêtre du second utilisateur (ou de tous les utilisateurs s'il y en a d'autres).

Figure 8–23

Les caractères lus par un client sont transmis aux autres clients.



```
C:\Users\Eric\Documents\Node.js>node client.js
bonjour
```

Cet utilisateur peut répondre de la même manière. Ce qu'il saisira se trouvera inscrit dans la fenêtre de tous les autres utilisateurs.

Si un utilisateur tape "exit", cela ferme sa session et le serveur le voit déconnecté.

Figure 8-24
Déconnexion d'un client

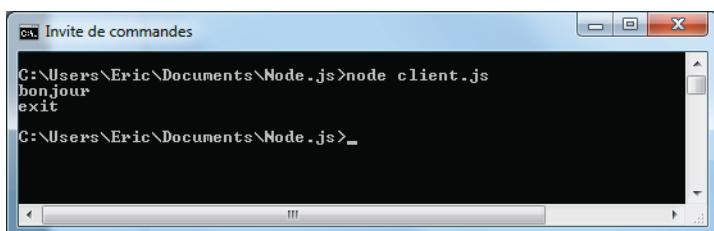


Figure 8-25
Décrémentation du nombre
de clients connectés



Les programmes du client et du serveur sont les suivants.

Fichier client.js

```
var net = require("net");
(function connect() {
    var client = net.connect(3000);
    client.setEncoding("utf8");

    process.stdin.resume();
    process.stdin.pipe(client);

    client.pipe(process.stdout);

    client.on("error", function() {
        // événement error émis lorsque le serveur s'est arrêté (fenêtre fermée)
        console.log("Perte de connexion au serveur...");
        connect();
    });

    client.on("end", function() {
```

```
// événement end émis lorsque le client a tapé "exit"
process.stdin.pause();
});
})();
```

Le programme du client consiste en une fonction qui s'appelle à l'issue de sa déclaration, comme nous l'avons fait dans l'exemple précédent. Cela permet d'appeler automatiquement la fonction en cas d'erreur (dans l'événement `error`).

La partie intéressante concerne les instructions utilisant `process.stdin` et `process.stdout`.

- `process.stdin.resume()` permet de mettre le programme Node du client en attente de saisie de caractères au clavier.
- `process.stdin.pipe(client)` permet de transmettre les caractères lus au clavier vers le stream associé au client, pour qu'il les traite dans l'événement `data`. Cet événement est traité dans le corps du programme serveur (voir fichier `server.js`).
- `client.pipe(process.stdout)` permet que les caractères reçus par le stream associé au client soient directement affichés sur l'écran du client. Ces caractères sont reçus lorsqu'un autre client a tapé sur la touche Entrée qui valide sa saisie (voir fichier `server.js`).

Enfin, les deux événements `error` et `end` sont traités dans le programme du client :

- l'événement `error` est déclenché lorsque le serveur s'est arrêté ;
- l'événement `end` est déclenché lorsque le client a tapé la chaîne "`exit`", puis a appuyé sur la touche Entrée.

Fichier server.js

```
var net = require("net");
var sockets = [];
var server = net.createServer(function(socket) {
  sockets.push(socket);
  console.log("Il y a " + sockets.length + " clients connectés");

  socket.on("data", function(chunk) {
    // événement data émis lorsque la socket reçoit des données (lecture clavier)
    if (chunk.toString().replace(/\r|\n/g, "") == "exit") socket.end();
    else {
      sockets.forEach(function(s) {
        if (s != socket) s.write(chunk);
      });
    }
});
```

```
socket.on("error", function() {
    // événement error émis lorsque le client s'est arrêté (fenêtre fermée)
    socket.emit("end");
});

socket.on("end", function() {
    // événement end émis lorsque le client a tapé "exit"
    sockets.splice(sockets.indexOf(socket), 1);
    console.log("Il reste " + sockets.length + " clients connectés");
});
});

server.on("listening", function() {
    console.log("Le serveur est démarré...");
});

server.listen(3000);
```

Lors de la connexion d'un client, une socket est mise à notre disposition. Nous l'intégrons dans un tableau `sockets` contenant toutes les sockets ouvertes. La longueur du tableau représente le nombre de clients connectés au serveur. Les événements `data`, `error` et `end` sont traités sur la socket associée au client connecté.

- L'événement `data` est déclenché lorsque des caractères sont reçus du clavier (voir le fichier `client.js`). On analyse la chaîne reçue et si elle correspond à `"exit"`, on ferme la socket par `socket.end()`, ce qui déclenchera l'événement `end`. Si la chaîne ne correspond pas à `"exit"`, elle est envoyée (par l'instruction `write()`) à tous les autres clients afin qu'ils l'affichent dans leur console (voir le fichier `client.js`).
- L'événement `error` est déclenché lorsque le client s'est arrêté. On émet alors l'événement `end` car le traitement dans les deux cas est identique (voir événement `end`).
- L'événement `end` est déclenché soit par l'événement `error`, soit parce que le client a tapé `"exit"`, ce qui provoque l'envoi de l'événement `end` par l'instruction `socket.end()` (expliqué dans l'événement `data`). Comme une socket de moins est disponible, il faut l'enlever de la table des sockets en indiquant le nouveau nombre de clients connectés. On enlève la socket du tableau `sockets` au moyen de la méthode `splice()`.

9

Gestion des connexions UDP

Le protocole TCP que nous avons étudié dans le chapitre précédent est un protocole fiable, ce qui signifie que les informations qui transitent sur le réseau sont transmises aux destinataires même si la qualité de la transmission est médiocre. De plus, les informations envoyées sont reçues dans le même ordre, permettant de garantir une chronologie dans les informations reçues. Pour réaliser cela, le protocole TCP met en œuvre des contrôles à toutes les étapes, pour chaque donnée transmise, afin de s'assurer de leur acheminement correct.

Le protocole UDP(*User Datagram Protocol*) est radicalement différent. Il n'effectue aucune vérification afin de s'assurer que les informations envoyées ont été correctement reçues, et les informations peuvent être reçues dans un autre ordre que celui où elles ont été envoyées. Effectuant moins de contrôles que le protocole TCP, le protocole UDP est plus rapide, mais il est aussi moins fiable.

On utilisera donc le protocole UDP lorsque les informations à envoyer ne sont pas d'une importance capitale, car on accepte la perte d'informations. Mais le gain en rapidité d'acheminement est appréciable par rapport à une connexion TCP.

Principe d'une connexion UDP

Comme pour les connexions TCP, une connexion UDP va consister en un dialogue entre deux entités, respectivement nommées client et serveur. Le serveur va se mettre en attente de messages reçus d'éventuels clients, auxquels il pourra également transmettre des informations.

Le serveur sera associé à un numéro de port, ce qui permettra aux clients d'envoyer des messages sur ce port. Le client et le serveur seront représentés par des sockets UDP, dont la gestion est réalisée par le module `dgram` interne à Node (`dgram` étant une abréviation de *datagramme*).

Comme pour TCP, il existe un programme que l'on peut utiliser pour simuler des clients UDP. Il s'agit du programme Netcat que nous décrivons dans la section suivante.

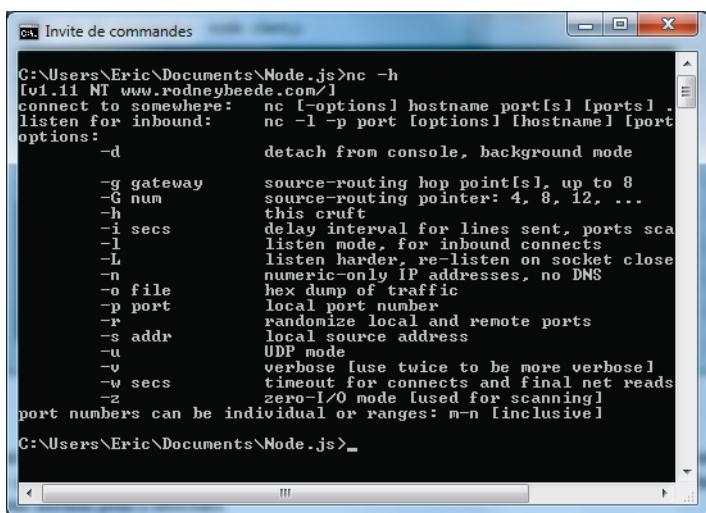
Le programme Netcat

Le programme Netcat permet de simuler un client UDP, qui se connectera sur le serveur UDP associé au port correspondant. Ce programme est accessible via la commande `nc`. Pour l'utiliser sous Windows, il faudra installer un exécutable le permettant, que vous trouverez par exemple à l'adresse <http://joncraton.org/blog/46/netcat-for-windows>.

Une fois que Netcat est installé, on peut vérifier son bon fonctionnement en tapant la commande `nc -h` dans un interpréteur de commandes.

Figure 9-1

Aide sur le programme Netcat



```
C:\> Invite de commandes
C:\Users\Eric\Documents\Node.js>nc -h
[vl.11 NT www.rodneybeede.com]
connect to somewhere: nc [-options] hostname port[s] [ports]
listen for inbound:   nc -l -p port [options] [hostname] [port]
options:
  -d          detach from console, background mode
  -G gateway  source-routing hop point[s], up to 8
  -G num      source-routing pointer: 4, 8, 12, ...
  -h          this cruft
  -i secs     delay interval for lines sent, ports sca
  -l          listen mode, for inbound connects
  -L          listen harder, re-listen on socket close
  -n          numeric-only IP addresses, no DNS
  -o file     hex dump of traffic
  -p port     local port number
  -r          randomize local and remote ports
  -s addr     local source address
  -u          UDP mode
  -v          verbose [use twice to be more verbose]
  -w secs     timeout for connects and final net reads
  -z          zero-I/O mode [used for scanning]
port numbers can be individual or ranges: m-n [inclusive]
C:\Users\Eric\Documents\Node.js>
```

La commande `nc` comporte beaucoup d'options, parmi lesquelles nous n'utiliserons que l'option `-u` permettant de se connecter à un port UDP sur le serveur. Par exemple, pour se connecter sur le port 3000 du serveur local, on tapera la commande suivante.

Commande Netcat pour se connecter sur le port 3000 associé à un serveur UDP

```
| nc -u localhost 3000
```

Une fois cette commande tapée, la fenêtre se met en attente et on peut saisir des chaînes de caractères qui seront envoyées au serveur. Voyons maintenant comment créer un serveur UDP.

Créer un serveur UDP

Comme pour un serveur TCP, la création d'un serveur UDP consiste à créer un programme qui sera exécuté lorsqu'un client se connectera sur ce port UDP. On utilise pour cela la méthode `dgram.createSocket()` définie dans le module `dgram`.

Utiliser la méthode `dgram.createSocket()`

Node fournit la méthode `dgram.createSocket(type)` du module `dgram` permettant de créer le serveur UDP. Deux types de sockets peuvent être créées : "`udp4`" et "`udp6`". Nous utiliserons ici uniquement les sockets de type "`udp4`".

Tableau 9–1 Méthodes de création d'un serveur UDP

Méthode	Signification
<code>dgram.createSocket(type, [callback])</code>	Crée un serveur UDP, qui devra se mettre en attente sur un port du serveur, des messages envoyés par les clients UDP, au moyen de l'instruction <code>server.bind(port)</code> . La fonction de callback indiquée en paramètre est optionnelle et elle est activée à chaque message reçu d'un client UDP sur ce serveur. L'événement <code>message</code> est également déclenché à chaque réception de message.
<code>server.bind(port, [callback])</code>	Met le serveur UDP en attente de messages envoyés par d'éventuels clients, sur le port indiqué. La fonction de callback est optionnelle, elle est activée par Node lorsque le serveur est prêt à recevoir les messages des utilisateurs. L'événement <code>listening</code> est déclenché (une seule fois) dès que le serveur est prêt à recevoir ces messages.

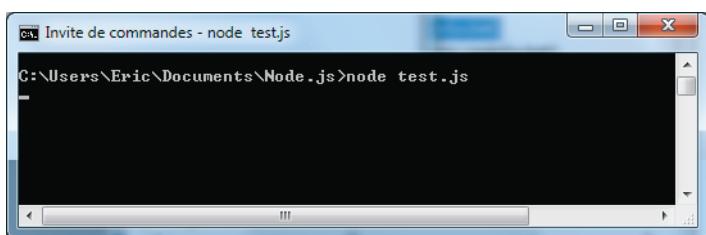
Création d'un serveur UDP

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4");
server.on("message", function(message) {
  console.log("Message reçu : " + message);
});
server.bind(3000);
```

Nous insérons le module `dgram` qui gère les connexions UDP. Le serveur est créé par l'instruction `dgram.createSocket("udp4")`, qui retourne un objet `server` de classe `dgram.Socket`. Le traitement de l'événement `message` sur le serveur permet de traiter les messages que les clients UDP envoient sur ce serveur. Enfin, il faut indiquer quel port est associé à ce traitement du serveur, au moyen de la méthode `server.bind(3000)`, qui associe ce serveur UDP au port 3000.

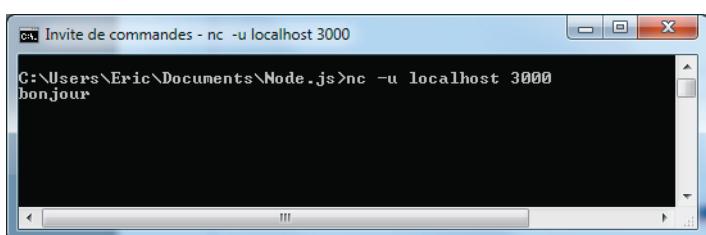
Pour tester ce programme, lançons le serveur au moyen de la commande `node test.js`.

Figure 9–2
Lancement d'un serveur UDP



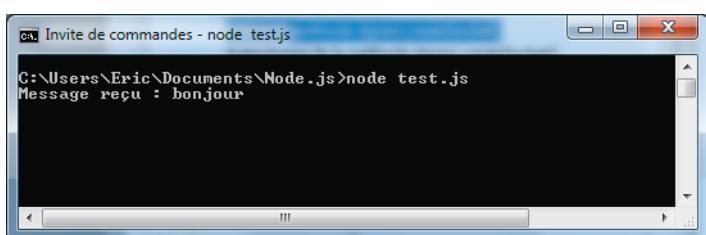
Le serveur est lancé et mis en attente de messages sur le port 3000. Au moyen du programme Netcat, simulons un client UDP qui envoie une chaîne de caractères vers le serveur sur le port 3000.

Figure 9–3
Envoi de caractères au serveur
par UDP



Nous avons lancé la commande `nc`, puis tapé la chaîne "bonjour". Le serveur a reçu cette chaîne et affiche dans sa console le message suivant.

Figure 9–4
Réception de messages
par le serveur UDP



Au fur et à mesure que l'on introduit des chaînes de caractères au moyen du client Netcat, celles-ci sont transmises au serveur qui les affiche.

Autre forme de la méthode `dgram.createSocket()`

Il est possible de spécifier une fonction de callback en paramètres de la méthode `dgram.createSocket()`. Cela permet d'indiquer le traitement à effectuer lorsque l'événement `message` se produit, c'est-à-dire lorsque le serveur UDP reçoit un message des utilisateurs. Cette façon d'écrire est plus rapide car on n'a plus besoin de traiter spécifiquement l'événement `message`.

Le code précédent pourrait alors s'écrire :

Gérer l'événement `message` sous forme de callback

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message) {
  console.log("Message reçu : " + message);
});
server.bind(3000);
```

Le résultat est identique au précédent.

Utiliser l'événement `listening` pour indiquer que le serveur a démarré

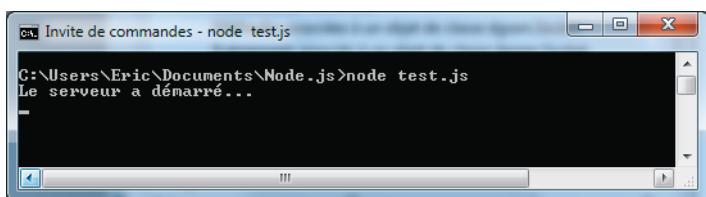
Lorsque la commande `node` a été tapée, le serveur UDP a démarré mais aucune information n'a été affichée afin de l'indiquer. L'événement `listening` déclenché sur l'objet `server` par Node permet d'effectuer un traitement lorsque le serveur est prêt à recevoir des messages des clients.

Traiter l'événement `listening` lors du démarrage du serveur

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message) {
  console.log("Message reçu : " + message);
});
server.bind(3000);

server.on("listening", function() {
  console.log("Le serveur a démarré...");
});
```

Figure 9–5
Utilisation de l'événement
`listening`



Un message s'affiche pour prévenir que le serveur est prêt à recevoir les messages des utilisateurs.

Remarquons que l'on peut également utiliser la fonction de callback indiquée dans les paramètres de la méthode `bind(port, callback)`, à la place de l'événement `listening`. Cette fonction de callback est déclenchée en même temps que l'événement `listening` et peut donc le remplacer.

Le code précédent pourrait alors s'écrire :

Utiliser la fonction de callback de la méthode `bind()`

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message) {
    console.log("Message reçu : " + message);
});
server.bind(3000, function() {
    console.log("Le serveur a démarré...");
});
```

Connexion/déconnexion du client ou du serveur

Au moyen du précédent programme, établissons la connexion entre le client Netcat et le serveur UDP. Si nous stoppons l'un ou l'autre des programmes (client ou serveur), l'autre programme n'est pas impacté et ne produit pas d'erreur. Le programme arrêté peut être redémarré et la communication peut ainsi continuer entre le client et le serveur.

Si nous regardons les résultats que nous avions obtenus dans les mêmes circonstances pour une connexion TCP, il est clair que ce n'est pas la même chose. En effet, l'arrêt d'un client ou du serveur dans une connexion TCP provoquait une rupture de connexion qui devait être gérée dans notre programme afin de ne pas provoquer d'erreur et l'arrêt du programme client ou serveur.

La différence de comportement tient au fait qu'une connexion TCP établit une connexion réelle entre le serveur et ses clients, tandis que la connexion UDP est plus souple car il n'y a pas de véritable connexion entre le serveur et ses clients, qui peuvent ainsi aller et venir à leur guise.

Créer un client UDP

Jusqu'à présent, nous avons simulé un client UDP au moyen du programme Netcat. Nous allons maintenant créer un programme qui remplace Netcat et permettra également d'envoyer des messages au serveur.

Utiliser la méthode `dgram.createSocket()`

La méthode `dgram.createSocket()` que nous avions précédemment utilisée pour créer le serveur sert également à créer un client UDP. Nous indiquons l'argument "`udp4`" en paramètres de la méthode.

Créer un client UDP

```
var client = dgram.createSocket("udp4");
```

L'objet `client` ainsi créé sera de classe `dgram.Socket`, tout comme l'objet `server`. Mais l'objet `server` doit en plus être associé à un port UDP au moyen de la méthode `server.bind(port)`, tandis que l'objet `client` n'a pas besoin d'être affecté à un port particulier. Dans ce cas, Node affecte un port aléatoire à la socket associée au client. Si toutefois on souhaite affecter un port particulier au client, il suffit d'utiliser l'instruction `client.bind(port)`, comme nous l'avons fait pour le serveur.

Dans ce cas, on peut alors se demander quelle est la différence entre le client et le serveur dans une connexion UDP, vu qu'on les crée de la même façon. En fait, les termes client et serveur sont utilisés de manière impropre ici, car l'un et l'autre peuvent être client ou serveur à tour de rôle. On dira que le serveur est celui à qui on a affecté un numéro de port avec `bind()`, sinon il serait impossible de lui envoyer des informations (si on n'a pas de port destinataire), tandis que le client n'a pas besoin d'avoir un port affecté en particulier (même si Node lui en affecte un interne).

Envoyer un message vers le serveur avec la méthode `send()`

Une fois le client UDP créé, il reste à lui permettre de communiquer avec le serveur. La méthode `client.send()` permet de réaliser l'envoi d'un buffer d'octets (constituant un message) vers le serveur. Le buffer d'octets sera créé au moyen de la classe `Buffer` de Node.

Tableau 9–2 Méthode d'envoi d'un message vers un serveur UDP

Méthode	Signification
<code>client.send(buf, offset, length, port, address, [callback])</code>	<p>Envoie un buffer d'octets vers le serveur repéré par son adresse IP (ou "<code>localhost</code>") et son port.</p> <p>Le paramètre <code>buf</code> est le buffer (de classe <code>Buffer</code>) que l'on souhaite envoyer au serveur.</p> <p>Le paramètre <code>offset</code> indique à partir de quel indice dans le buffer on commence.</p> <p>Le paramètre <code>length</code> représente le nombre d'octets que l'on souhaite transmettre.</p> <p>Les paramètres <code>port</code> et <code>address</code> sont respectivement le port et l'adresse IP du serveur (voire "<code>localhost</code>") vers lequel on envoie le buffer.</p> <p>La fonction de callback est optionnelle et appelée par Node lorsque le buffer est transmis au serveur. Le buffer peut alors être réutilisé si nécessaire. Si vous devez modifier ce buffer après son envoi, cela ne pourra être fait que dans cette fonction de callback, afin d'être sûr que Node ne l'utilise plus.</p>

Utilisons la méthode `send()` pour envoyer un message au serveur, qui l'affichera dans sa console.

Utiliser la méthode `send()` pour dialoguer avec le serveur

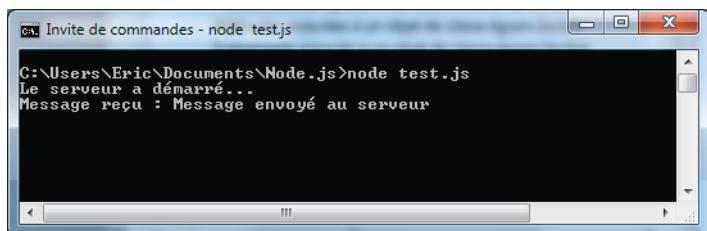
```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message) {
  console.log("Message reçu : " + message);
});
server.bind(3000, function() {
  console.log("Le serveur a démarré...");
});

var client = dgram.createSocket("udp4");
var msg = new Buffer("Message envoyé au serveur");
client.send(msg, 0, msg.length, 3000, "localhost");
```

Le serveur est créé, puis mis à l'écoute sur le port 3000. Un client est ensuite créé, à partir duquel on envoie vers ce serveur, un buffer créé par `new Buffer()`.

Figure 9–6

Envoi d'un message du serveur



Permettre au serveur de répondre au client

Le serveur reçoit le message envoyé par le client et l'affiche dans sa console. Cependant, il serait intéressant de pouvoir indiquer au client que le message a été reçu par le serveur.

Pour cela, l'événement `message` indiquant la réception d'un message, contient également le paramètre `rinfo` (signifiant *remote information*) qui est un objet `{ address, port }`, dont les propriétés correspondent respectivement à l'adresse IP et au port de la socket émettrice du message reçu. Donc pour répondre au client, il suffit que le serveur envoie un message à destination de ce port sur cette adresse IP. Par exemple, une fois que le serveur a reçu le message du client, renvoyons au client un message contenant *'Merci, bien reçu'*.

Retourner un message d'acquittement au client

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message, rinfo) {
    console.log("Message reçu : " + message);
    var buf = new Buffer("Merci, bien reçu");
    server.send(buf, 0, buf.length, rinfo.port, rinfo.address);
});
server.bind(3000, function() {
    console.log("Le serveur a démarré...");
});

var client = dgram.createSocket("udp4");
var msg = new Buffer("Message envoyé au serveur");
client.send(msg, 0, msg.length, 3000, "localhost");

client.on("message", function(message) {
    console.log(message.toString());
});
```

Le message retourné au client est créé sous forme d'objet de classe `Buffer`, puis envoyé par la méthode `server.send()` en utilisant le paramètre `rinfo` associé au message précédemment reçu. Le client recevra ce message en écoutant l'événement `message`, mais

cette écoute peut aussi être intégrée directement dans la fonction de callback de la méthode `dgram.createSocket()`, comme le montre le programme suivant.

Autre forme d'écriture du précédent programme

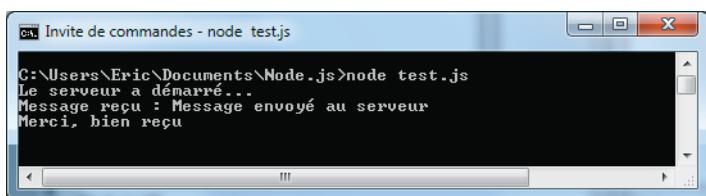
```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message, rinfo) {
    console.log("Message reçu : " + message);
    var buf = new Buffer("Merci, bien reçu");
    server.send(buf, 0, buf.length, rinfo.port, rinfo.address);
});
server.bind(3000, function() {
    console.log("Le serveur a démarré...");
});

var client = dgram.createSocket("udp4", function(message) {
    console.log(message.toString());
});
var msg = new Buffer("Message envoyé au serveur");
client.send(msg, 0, msg.length, 3000, "localhost");
```

Dans les deux formes d'écriture, le résultat est identique et montre le dialogue entre le client et le serveur en UDP, dans les deux sens.

Figure 9–7

Communication client serveur
en UDP



La méthode `send()` peut servir à envoyer des messages du client vers le serveur, mais également du serveur vers le client.

Dissocier le programme du client et celui du serveur

Le code précédent est opérationnel, mais dans la réalité le code du client et celui du serveur sont dans des fichiers différents. On peut le faire ici en créant deux fichiers, l'un `server.js` et l'autre `client.js`, qui seront tous deux activés par la commande `node`.

Fichier server.js

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message, rinfo) {
    console.log("Message reçu : " + message);
    var buf = new Buffer("Merci, bien reçu");
    server.send(buf, 0, buf.length, rinfo.port, rinfo.address);
});
server.bind(3000, function() {
    console.log("Le serveur a démarré sur le port 3000...");
});
```

Le serveur se met en attente de messages du client, et lorsqu'il reçoit un message, il l'affiche puis retourne un autre message au client lui indiquant "*Merci, bien reçu*".

Fichier client.js

```
var dgram = require("dgram");
var client = dgram.createSocket("udp4", function(message) {
    console.log("Message reçu : " + message);
});

console.log("Le client a démarré sur un port attribué par Node...");
console.log("On envoie un message au serveur...");

var msg = new Buffer("Message envoyé au serveur");
client.send(msg, 0, msg.length, 3000, "localhost");
```

Le client démarre en envoyant aussitôt un message au serveur, puis se met en attente d'éventuels messages, qui seront affichés dans la console du client.

Démarrons en premier le serveur.

Figure 9-8
Démarrage du serveur



Le serveur est démarré, puis se met en attente de messages. On démarre ensuite le client.

Figure 9–9
Démarrage d'un client

```
C:\> node client.js
Le client a démarré sur un port attribué par Node...
On envoie un message au serveur...
Message reçu : Merci, bien reçu
```

Un message est envoyé au serveur, qui répond par un acquittement. La console du serveur montre le message reçu.

Figure 9–10
Envoi de messages au serveur
par le client

```
C:\> node server.js
Le serveur a démarré sur le port 3000...
Message reçu : Message envoyé au serveur
```

Événements associés à un objet de classe `dgram.Socket`

Nous récapitulons ci-dessous les événements pouvant survenir sur une socket UDP, que celle-ci soit client ou serveur.

Tableau 9–3 Événements sur une socket UDP (client ou serveur)

Événement	Signification
<code>message</code>	Un message a été reçu sur la socket. La fonction de callback de traitement de l'événement prend deux paramètres : - le message reçu ; - l'objet <code>rinfo { address, port }</code> permettant de connaître l'émetteur du message reçu.
<code>listening</code>	La socket est prête à recevoir des messages.
<code>close</code>	La socket est fermée et ne peut plus recevoir de message. Cet événement survient lors de l'appel à <code>socket.close()</code> .
<code>error</code>	Une erreur est survenue sur la socket.

Méthodes associées à un objet de classe dgram.Socket

De même, nous listons ci-dessous les différentes méthodes que l'on peut utiliser sur une socket UDP, que celle-ci soit utilisée en tant que client ou serveur. La socket a été préalablement créée au moyen de `dgram.createSocket()`.

Tableau 9–4 Méthodes sur une socket UDP (client ou serveur)

Méthode	Signification
<code>socket.send(buf, offset, length, port, address, [callback])</code>	Envoie un buffer d'octets vers la socket repérée par son adresse IP (ou " <code>localhost</code> ") et son port. Le paramètre <code>buf</code> est le buffer (de classe Buffer) que l'on souhaite envoyer. Le paramètre <code>offset</code> indique à partir de quel indice dans le buffer on commence. Le paramètre <code>length</code> représente le nombre d'octets que l'on souhaite transmettre. Les paramètres <code>port</code> et <code>address</code> sont respectivement le port et l'adresse IP de la socket (voire " <code>localhost</code> ") vers laquelle on envoie le buffer. La fonction de callback est optionnelle et est appelée par Node lorsque le buffer est transmis. Le buffer peut alors être réutilisé si nécessaire. Si vous devez modifier ce buffer après son envoi, cela ne peut être fait que dans cette fonction de callback, afin d'être sûr que Node ne l'utilise plus.
<code>socket.bind(port, [address], [callback])</code>	Met la socket UDP en attente de messages envoyés par d'autres sockets, sur le port indiqué. La fonction de callback est optionnelle, elle est activée par Node lorsque la socket est prête à recevoir les messages d'utilisateurs. L'événement <code>listening</code> est déclenché (une seule fois) dès que la socket est prête à recevoir ces messages.
<code>socket.close()</code>	Ferme la socket. Elle ne recevra plus d'événements <code>message</code> .
<code>socket.address()</code>	Retourne un objet <code>rinfo</code> de type <code>{ address, port }</code> permettant de connaître l'adresse IP et le port de la socket.

Afin d'illustrer la méthode `close()`, montrons qu'une fois cette méthode appelée, la socket ne peut plus recevoir de nouveaux messages. Nous utilisons un client et un serveur, le client effectuant l'envoi de deux messages sur la socket du serveur. Mais comme le serveur ferme sa socket après la réception du premier message, il ne pourra pas recevoir le second.

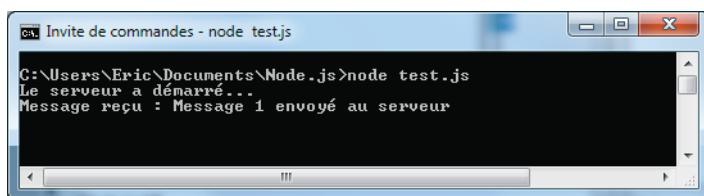
Appeler la méthode `socket.close()` après la réception d'un message par le serveur

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message, rinfo) {
    console.log("Message reçu : " + message);
    server.close();
});
server.bind(3000, function() {
    console.log("Le serveur a démarré...");
});

var client = dgram.createSocket("udp4", function(message) {
    console.log(message.toString());
});
var msg1 = new Buffer("Message 1 envoyé au serveur");
client.send(msg1, 0, msg1.length, 3000, "localhost");
var msg2 = new Buffer("Message 2 envoyé au serveur");
client.send(msg2, 0, msg2.length, 3000, "localhost");
```

Figure 9-11

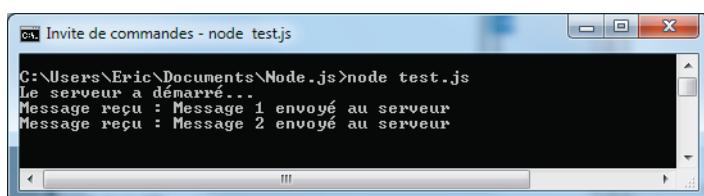
Après `close()`, le serveur ne peut plus recevoir de messages.



Si l'on enlève l'instruction de fermeture de la socket, les deux messages envoyés par le client sont reçus par le serveur.

Figure 9-12

Sans `close()`, le serveur peut recevoir les messages.



Exemple : communication entre deux sockets UDP

On souhaite montrer un exemple de communication entre deux sockets UDP, dont l'une sera considérée comme le client et l'autre le serveur. On sait que dans le cas de sockets UDP, la notion de client et serveur est relative et chacune des sockets peut être l'un ou l'autre au fil du temps.

Le serveur sera démarré sur le port 3000, et attendra de recevoir un message du client, qui lui sera démarré sur le port 3001. Dès qu'un message a été reçu par le serveur, celui-ci peut envoyer des messages au client, qui peut également lui répondre. Le client a donc l'initiative du commencement du dialogue. Les messages sont introduits par le clavier dans chacune des consoles.

Un exemple de dialogue entre le client et le serveur est illustré par les deux fenêtres suivantes.

Figure 9-13

Dialogue client-serveur,
côté serveur

```
C:\Users\Eric\Documents\Node.js>node server.js
Le serveur a démarré sur le port 3000...
bonjour
hello
OK
!!!
ca va bien ???
parfait !!!
```

Figure 9-14

Dialogue client-serveur,
côté client

```
C:\Users\Eric\Documents\Node.js>node client.js
Le client a démarré sur le port 3001...
bonjour
hello
OK
!!!
ca va bien ???
parfait !!!
```

Les programmes du client et du serveur sont les suivants.

Fichier server.js

```
var dgram = require("dgram");
var server = dgram.createSocket("udp4", function(message, rinfo) {
    console.log(message.toString());
    process.stdin.resume();
    process.stdin.removeAllListeners("data").on("data", function(chunk) {
        var buf = new Buffer(chunk.toString().replace(/\r|\n/g, ""));
        server.send(buf, 0, buf.length, rinfo.port, rinfo.address);
    });
});
server.bind(3000, function() {
    console.log("Le serveur a démarré sur le port 3000...");
});
```

Dès que le serveur reçoit un message, celui-ci est affiché dans la console du serveur, puis la console se met en attente de saisie sur le clavier au moyen de `process.stdin.resume()`. Les caractères introduits (événement `data`) sont transformés en objet de classe `Buffer` puis envoyés au client (en supprimant les éventuels retours à la ligne du texte saisi).

Fichier client.js

```
var dgram = require("dgram");
var client = dgram.createSocket("udp4", function(message, rinfo) {
    console.log(message.toString());
    process.stdin.resume();
    process.stdin.removeAllListeners("data").on("data", function(chunk) {
        var buf = new Buffer(chunk.toString().replace(/\r|\n/g, ""));
        client.send(buf, 0, buf.length, rinfo.port, rinfo.address);
    });
});
client.bind(3001, function() {
    console.log("Le client a démarré sur le port 3001...");
});

process.stdin.resume();
process.stdin.on("data", function(chunk) {
    var buf = new Buffer(chunk.toString().replace(/\r|\n/g, ""));
    client.send(buf, 0, buf.length, 3000, "localhost");
});
```

Le programme du client est presque le même que celui du serveur. Toutefois, le client a l'initiative du commencement du dialogue avec le serveur. C'est pourquoi, en dehors de la réception d'un message, le client effectue une lecture du clavier, puis il envoie ce qui a été saisi au serveur. Ce traitement peut ensuite s'effectuer à chaque réception de message du serveur.

Remarquez l'utilisation de la méthode `process.stdin.removeAllListeners("data")`. En effet, la méthode `on()` étant placée dans le gestionnaire de l'événement réception de message, les gestionnaires s'accumuleraient à chaque réception de message, ce qui déclencherait autant de traitements pour chaque message reçu.

10

Gestion des connexions HTTP

Dans les précédents chapitres, nous avons étudié le fonctionnement des connexions TCP et UDP. Mais depuis l'avènement d'Internet, les connexions HTTP sont majoritaires car elles peuvent s'utiliser via un navigateur web classique.

Node permet de créer facilement un serveur HTTP qui délivrera les pages HTML disponibles sur le serveur. Il permet également de créer des clients HTTP qui pourront se connecter à un serveur HTTP, comme nous avions des clients TCP ou UDP qui se connectaient à des serveurs de même type. Ce chapitre a pour but de vous présenter et expliquer ces différents modes de fonctionnement.

Créer un serveur HTTP

La création d'un serveur HTTP s'effectue au moyen de la méthode `http.createServer()`, définie dans le module `http` de Node. Cette méthode retourne un objet `server` de classe `http.Server`, sur lequel on utilise la méthode `server.listen(port)` permettant de mettre en attente le serveur sur un numéro de port particulier.

Tableau 10–1 Méthodes de création d'un serveur HTTP

Méthode	Signification
<code>http.createServer([requestListener])</code>	Retourne un objet de classe <code>http.Server</code> correspondant au serveur HTTP créé par Node (objet <code>server</code>). La fonction de callback indiquée est optionnelle et permet de traiter l'événement <code>request</code> indiquant qu'un client s'est connecté au serveur.

Tableau 10-1 Méthodes de création d'un serveur HTTP (suite)

Méthode	Signification
<code>server.listen(port, [callback])</code>	Met le serveur à l'écoute du port indiqué. La fonction de callback est optionnelle et elle est déclenchée lorsque le serveur est à l'écoute des connexions des clients (la fonction de callback correspond à l'événement <code>listening</code>).

Ces deux méthodes sont utilisées dans l'exemple suivant.

Utiliser la méthode `http.createServer()`

Voici un exemple d'utilisation de la méthode `http.createServer()` et de la méthode `server.listen()` associée. Nous créons ici un serveur HTTP qui écoutera le port 3000.

Créer un serveur HTTP affecté au port 3000

```
var http = require("http");
var server = http.createServer();
server.on("request", function(request, response) {
  var html = "";
  html += "<html><head><meta charset=utf-8></head>";
  html += "<body><p>Bienvenue sur le serveur HTTP de Node !</p></body>";
  html += "</html>";
  response.end(html);
});
server.listen(3000);
```

Nous créons le serveur HTTP au moyen de `http.createServer()`, puis nous écoutons les événements `request` déclenchés lorsque les utilisateurs se connectent à notre serveur HTTP. Le port d'écoute (ici, 3000) est spécifié dans la méthode `server.listen(port)`.

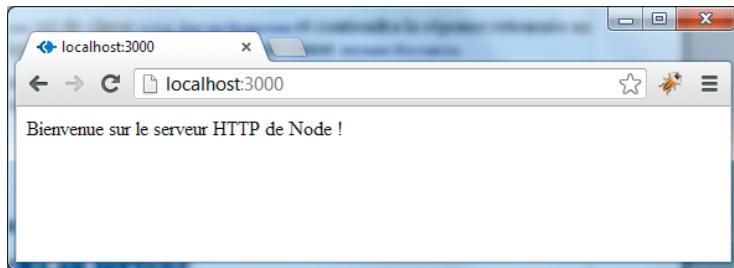
Le traitement de l'événement `request`, déclenché lorsqu'un utilisateur se connecte au serveur, correspond à une fonction de callback qui prend les deux paramètres `request` et `response`.

- Le paramètre `request` est de classe `http.IncomingMessage` et correspond aux arguments transmis par le client HTTP afin que le serveur puisse traiter sa requête. C'est également un stream en lecture de classe `stream.Readable`.
- Le paramètre `response` est de classe `http.ServerResponse` et contiendra la réponse retournée au client HTTP. C'est également un stream en écriture de classe `stream.Writable`.

La construction de la réponse se fait ici en HTML et elle est transmise au moyen de la méthode `response.end()`. Le test du serveur HTTP est effectué au moyen d'un navigateur quelconque, ici Chrome.

Figure 10–1

Un client connecté à un serveur HTTP

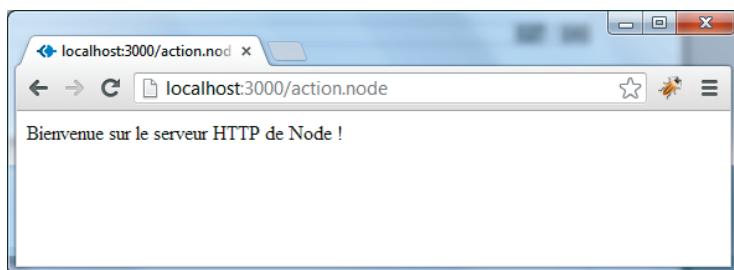


Nous avons indiqué l'URL `http://localhost:3000`. À noter que toute autre URL (utilisant le serveur `localhost` et le port 3000) produit le même résultat car, pour l'instant, les paramètres de la requête ne sont pas analysés par le serveur, qui construit toujours la même réponse.

Par exemple, en introduisant l'URL `http://localhost:3000/action.node` :

Figure 10–2

Utiliser différentes URL pour se connecter au serveur HTTP



Cela montre que l'affichage produit est toujours le même. Il faudra que le code de notre programme Node soit plus consistant afin de créer des affichages différents... Ce que nous apprendrons à faire dans la seconde partie de cet ouvrage.

Autre forme de la méthode `http.createServer()`

L'événement `request` étant l'un des événements principaux pouvant survenir dans un serveur HTTP, Node donne la possibilité d'indiquer le traitement correspondant à cet événement directement dans la méthode `http.createServer()`, en utilisant une fonction de callback en paramètre de la méthode.

Le code précédent peut donc également s'écrire :

Gérer l'événement request sous forme de callback

```
var http = require("http");
var server = http.createServer(function(request, response) {
    var html = "";
    html += "<html><head><meta charset=utf-8></head>";
    html += "<body><p>Bienvenue sur le serveur HTTP de Node !</p></body>";
    html += "</html>";
    response.end(html);
});
server.listen(3000);
```

Le résultat est identique dans le navigateur.

Objet associé au paramètre request

La fonction de traitement de l'événement `request` prend deux paramètres, nommés `request` et `response`. Nous étudions ici le paramètre `request`, qui est de classe `http.IncomingMessage`, qui dérive de la classe `stream.Readable`. L'objet `request` est donc un stream en lecture. Affichons son contenu à l'aide de la méthode `util.inspect()` du module `util` (ce module est étudié au chapitre 4, « Méthodes utilitaires »).

Afficher le contenu de l'objet request

```
var http = require("http");
var util = require("util");
var server = http.createServer(function(request, response) {
    var html = "";
    html += "<html><head><meta charset=utf-8></head>";
    html += "<body><p>Bienvenue sur le serveur HTTP de Node !</p></body>";
    html += "</html>";
    response.end(html);
    if (request.url == "/")
        console.log(util.inspect(request, { depth : 0 }));
});
server.listen(3000);
```

Le programme est identique au précédent, sauf pour la partie qui affiche le contenu de l'objet `request` dans la console du serveur. Nous testons d'abord si l'URL est bien `"/"`, car l'événement `request` est aussi appelé pour afficher le fichier `favicon.ico`, ce qui multiplie les affichages dans la console. Ensuite, nous utilisons `util.inspect()` du module `util`, en spécifiant une profondeur de 0 afin de ne pas avoir trop de données affichées.

Le fichier `favicon.ico` est un fichier image affiché par les navigateurs devant l'URL indiquée dans la barre d'adresses de ces derniers. Avant chaque affichage d'une URL dans la fenêtre du navigateur, ce fichier, s'il existe, est transmis par le serveur au navigateur afin d'être affiché dans la barre d'adresses.

Figure 10–3
Contenu de l'objet request

```
C:\Users\Eric\Documents\Node.js>node test.js
{
  _readableState: [Object],
  readable: true,
  domain: null,
  events: {},
  _maxListeners: 10,
  socket: [Object],
  connection: [Object],
  httpVersion: '1.1',
  complete: false,
  headers: [Object],
  trailers: {},
  pending: [],
  pendingIndex: 0,
  url: '/',
  method: 'GET',
  statusCode: null,
  client: [Object],
  consuming: true,
  dumped: true,
  httpVersionMajor: 1,
  httpVersionMinor: 1,
  upgrade: false,
  read: [Function] >
```

Les principales propriétés de l'objet `request` (affichées sur la figure 10–3) sont listées dans le tableau 10–2. Elles sont en lecture seule (donc non modifiables).

Tableau 10–2 Propriétés principales de l'objet request

Propriété	Signification
<code>request.url</code>	URL pour laquelle l'événement <code>request</code> est déclenché. Elle correspond à ce qui suit le nom du serveur et le port. Attention : comme l'événement <code>request</code> est appelé également pour le fichier <code>/favicon.ico</code> , il faudra tester si l'URL est différente de ce fichier afin d'effectuer une seule fois le traitement.
<code>request.method</code>	Méthode associée à la requête (" <code>GET</code> ", " <code>POST</code> ", " <code>PUT</code> ", " <code>DELETE</code> "). Si l'URL est introduite directement dans la barre d'adresses du navigateur, la requête est " <code>GET</code> ").
<code>request.headers</code>	En-têtes (<code>headers</code>) utilisés dans la requête reçue.
<code>request.client</code>	Objet de classe <code>net.Socket</code> correspondant au client TCP qui se connecte au serveur.

Par exemple, affichons dans la fenêtre du navigateur le contenu de `request.headers`, et dans la console du serveur les différentes URL pour lesquelles l'événement `request` est déclenché.

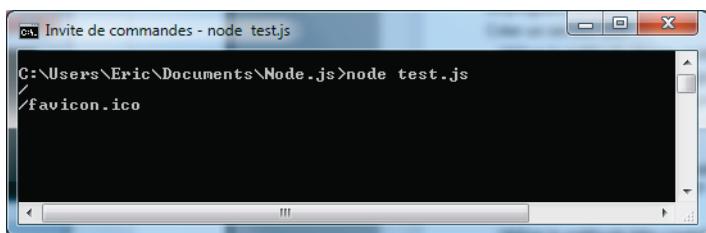
Afficher le contenu de request.headers dans le navigateur

```
var http = require("http");
var util = require("util");
var server = http.createServer(function(request, response) {
  console.log(request.url);
  var html = util.inspect(request.headers, { depth : 0 });
  if (request.url == "/")
    response.end(html);
});
server.listen(3000);
```

Nous obtenons dans la console du serveur l'affichage des deux fichiers pour lesquels l'événement `request` est déclenché.

Figure 10-4

URL qui ont déclenché l'événement `request`.



Dans la fenêtre du navigateur, on obtient l'affichage de `request.headers`.

Figure 10-5

Contenu de l'objet `request.headers`



Cela signifie que si l'on veut récupérer la valeur d'une propriété de l'objet `request.headers`, par exemple la propriété `cache-control`, il suffit de l'utiliser sous la forme `request.headers["cache-control"]`. Rappelons que les propriétés de l'objet `request` sont en lecture seule.

Objet associé au paramètre response

Le paramètre `response` est le second paramètre associé à l'événement `request` (lequel indique qu'un client se connecte au serveur). Le paramètre `response` est un stream en écriture qui permet d'envoyer une réponse au navigateur, afin qu'il l'affiche.

L'objet `response` est de classe `http.ServerResponse`, définie par Node. Cette classe dérive de `stream.Writable`, ce qui permet d'utiliser l'objet `response` comme un stream en écriture.

Nous savons qu'un stream en écriture (voir le chapitre 5, « Gestion des streams ») possède les deux méthodes `write()` et `end()` permettant respectivement d'écrire des octets sur le stream et de le fermer.

Tableau 10-3 Méthodes définies sur le stream en écriture associé à l'objet `response`

Méthode	Signification
<code>response.write(chunk, [encoding], [callback])</code>	Écrit le buffer d'octets indiqué par <code>chunk</code> sur le stream. Si le buffer est représenté par une chaîne de caractères, le paramètre facultatif <code>encoding</code> permet de préciser l'encodage utilisé ("utf8" par défaut). Le paramètre <code>callback</code> représente une éventuelle fonction de callback qui sera appelée à la fin de l'exécution de l'instruction.
<code>response.end([chunk], [encoding], [callback])</code>	Ferme le stream. Le buffer <code>chunk</code> , si indiqué, sera écrit sur le stream avant sa fermeture. Les paramètres <code>chunk</code> , <code>encoding</code> et <code>callback</code> sont facultatifs et ont la même signification que dans la méthode <code>write(chunk, encoding, callback)</code> . Cette instruction est obligatoire dans le cas d'un serveur HTTP.

On pourra ainsi utiliser autant d'instructions `write()` que souhaitées, qui permettront d'envoyer la réponse au navigateur. Lorsque l'instruction `end()` sera utilisée, les autres instructions `write()` ou `end()` qui suivront ne seront plus prises en compte.

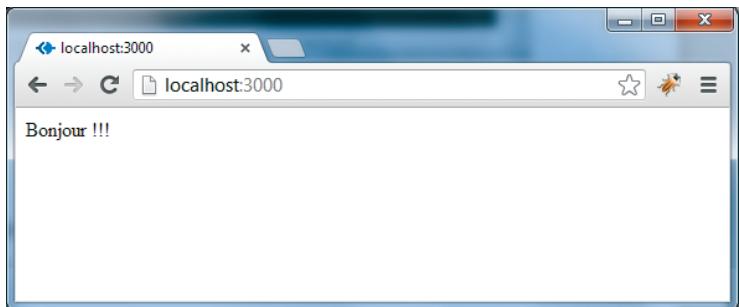
Remarquons que l'instruction `response.end()` est obligatoire, sinon le stream en écriture associé à `response` n'est pas fermé, ce qui provoque l'attente du navigateur, qui n'affiche rien car il attend cette instruction `end()`.

Utilisation des instructions `write()` et `end()` avec l'objet `response`

```
var http = require("http");
var util = require("util");
var fs = require("fs");
var server = http.createServer(function(request, response) {
  console.log(request.url);
  response.write("<html><head></head>");
  response.end();
});
```

```
    response.write("<body><p> Bonjour !!!</p></body>");
    response.end("<html>");
});  
server.listen(3000);
```

Figure 10–6
Réponse envoyée
au navigateur



Remarquons que si l'instruction `response.end(html)` ne sert pas à afficher du code HTML, elle peut simplement s'écrire `response.end()`, sans argument.

Envoyer un fichier statique en réponse au navigateur

Un cas très utile et fréquent consiste à envoyer un fichier HTML en réponse au navigateur, plutôt que de construire le code HTML dans le traitement de l'événement `request` du serveur. Cette possibilité permet de modifier le fichier HTML sans avoir à corriger le code du serveur, ce qui permet de produire un affichage différent dans la réponse envoyée au navigateur.

Pour cela, on utilise la méthode `readable.pipe(writable)`, définie dans le module `stream` de Node. Elle permet d'envoyer un stream en lecture vers un stream en écriture, sans gérer nous-mêmes les différents événements liés aux streams (voir le chapitre 5, « Gestion des streams »).

Utilisation de la méthode pipe() pour écrire le flux dans la réponse

```
var http = require("http");
var util = require("util");
var fs = require("fs");
var server = http.createServer(function(request, response) {
  console.log(request.url);
  var index = fs.createReadStream("index.html");
  index.pipe(response);
});
server.listen(3000);
```

Le fichier `index.html` est ouvert en tant que stream en lecture par l'instruction `fs.createReadStream("index.html")` du module `fs`. Ce stream en lecture est ensuite envoyé par l'instruction `pipe()` vers le stream en écriture représenté par l'objet `response`.

Remarquons que l'instruction `pipe()` ferme le stream destination par défaut, dès que le stream en entrée est vide. Donc l'instruction `response.end()` est effectuée par défaut, ce qui est obligatoire pour le navigateur.

Le fichier `index.html` peut être le suivant :

Fichier index.html envoyé vers le stream response

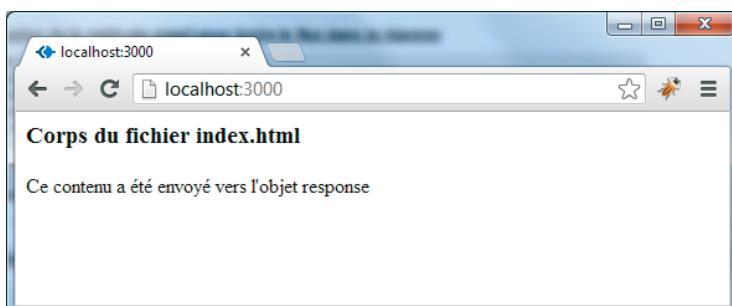
```
<html>
<head>
<meta charset="utf-8" />
</head>

<body>
<h3> Corps du fichier index.html </h3>
<p> Ce contenu a été envoyé vers l'objet response </p>
</body>

</html>
```

Lorsqu'une URL est indiquée dans la barre d'adresses, le fichier `index.html` décrit précédemment s'affiche dans la fenêtre du navigateur. Toute modification du contenu de ce fichier est reportée à l'écran, sans avoir à redémarrer le serveur.

Figure 10–7
Fichier statique envoyé
au navigateur



Quelle que soit l'URL entrée, le fichier `index.html` s'affiche et le contenu visible à l'écran est ainsi toujours le même. Une amélioration serait de produire un affichage différent selon les URL utilisées, en ayant à notre disposition plusieurs fichiers HTML.

Utiliser plusieurs fichiers statiques dans la réponse

La suite du programme précédent consiste à analyser l'URL entrée dans le navigateur et à renvoyer le fichier HTML correspondant à l'URL utilisée. Si une URL n'est pas reconnue (pas de fichier HTML associé), on renvoie le fichier `404.html` créé sur le serveur.

Utiliser plusieurs fichiers HTML en réponse

```
var http = require("http");
var util = require("util");
var fs = require("fs");
var url = require("url");
var server = http.createServer(function(request, response) {
    console.log(request.url);
    var filename = url.parse(request.url).pathname;
    if (filename == "/favicon.ico") return;
    if (filename == "/") filename = "/index.html";
    filename = "." + filename;
    fs.exists(filename, function(exists) {
        if (!exists) filename = "404.html";
        var file = fs.createReadStream(filename);
        file.pipe(response);
    });
});
server.listen(3000);
```

Nous récupérons d'abord l'URL transmise dans `request.url`, pour laquelle nous extrayons le nom du fichier (`pathname`). Nous excluons le fichier `favicon.ico` et nous transformons l'URL `/` en `/index.html`.

Le fichier résultat comporte le caractère `"/"` au début de son nom. Nous le préfixons de `". "` afin d'obtenir un chemin relatif par rapport au répertoire d'exécution du serveur. Si le fichier existe, on l'envoie au navigateur par la méthode `pipe()`, sinon on transmet le fichier `404.html` comme demandé.

Fichier `404.html` retourné en cas d'erreur

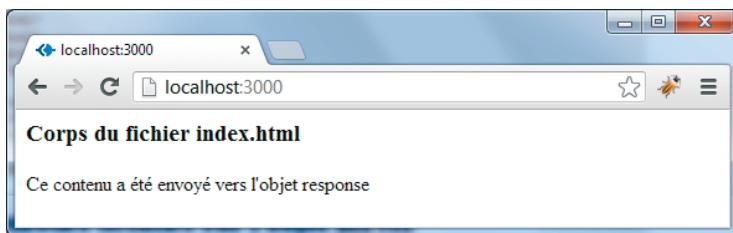
```
<html>
<head>
<meta charset="utf-8" />
</head>

<body>
<h3>404 Error : File not found</h3>
</body>

</html>
```

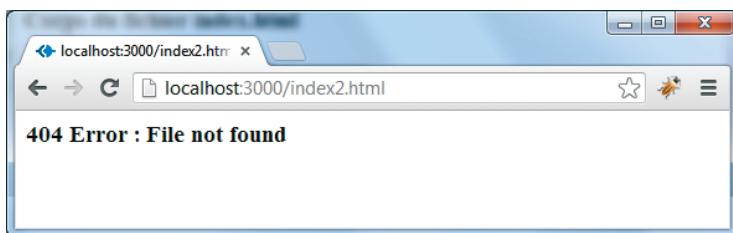
Si l'URL correspond à un fichier sur le serveur (ici, `index.html`), l'affichage est le suivant.

Figure 10–8
Fichier envoyé au navigateur



Tandis que si l'on entre une URL inconnue (ici, `index2.html`), on obtient :

Figure 10–9
Réponse envoyée si le fichier
n'est pas trouvé.



Envoyer un en-tête HTTP dans la réponse au navigateur

Les éléments envoyés précédemment au navigateur étaient inscrits dans le corps (*body*) de la réponse HTTP. Une réponse HTTP contient également une partie en-tête (*header*) qui comporte des informations complémentaires, par exemple :

- les cookies utilisés ;
- le type du contenu (type MIME), spécifiant si c'est du texte, une vidéo, un fichier PDF, etc. ;
- la durée de vie du contenu dans le cache du navigateur ;
- etc.

Node permet de spécifier ces informations au moyen de la méthode `response.writeHead(statusCode, headers)` définie sur l'objet `response`. Le paramètre `statusCode` sert à indiquer le résultat de la requête HTTP :

- `200` : OK ;
- `403` : accès interdit ;
- `404` : fichier non trouvé ;
- etc.

Tandis que le paramètre `headers` représente un tableau d'objets spécifiant les en-têtes envoyés.

Tableau 10-4 Méthodes définies sur le stream en écriture associé à l'objet response

Méthode	Signification
<code>response.writeHead(statusCode, headers)</code>	Indique le code retour de la requête HTTP dans le paramètre <code>statusCode</code> et les en-têtes dans le tableau <code>headers</code> . Le tableau <code>headers</code> est un tableau d'objets dont les clés sont les noms des en-têtes (par exemple, <code>{ "Content-Type" : "text/html" }</code>).
<code>response.setHeader(name, value)</code>	Positionne l'en-tête <code>name</code> à la valeur <code>value</code> . Très pratique si l'on veut positionner un en-tête sans l'envoyer immédiatement (à utiliser lorsque la méthode <code>writeHead()</code> provoque une erreur si elle est appelée plusieurs fois).

Le programme précédent est légèrement modifié pour tenir compte de ces informations à transmettre.

Indiquer le statusCode dans la réponse au navigateur

```
var http = require("http");
var util = require("util");
var fs = require("fs");
var url = require("url");
var server = http.createServer(function(request, response) {
  console.log(request.url);
  var filename = url.parse(request.url).pathname;
  if (filename == "/favicon.ico") return;
  if (filename == "/") filename = "/index.html";
  filename = "." + filename;
  fs.exists(filename, function(exists) {
    if (!exists) {
      filename = "404.html";
      response.writeHead(404, {"Content-Type" : "text/html"});
    } else response.writeHead(200, {"Content-Type" : "text/html"});
    var file = fs.createReadStream(filename);
    file.pipe(response);
  });
});
server.listen(3000);
```

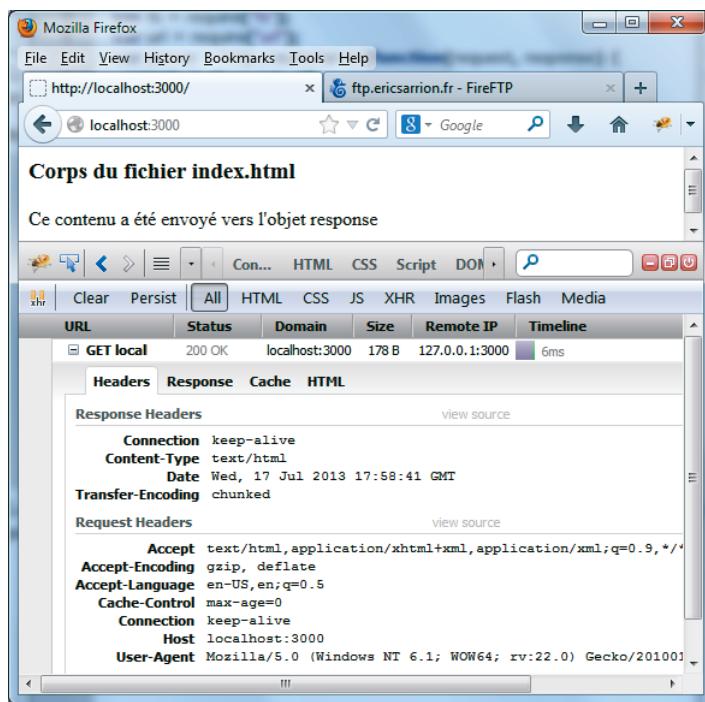
Les en-têtes sont visualisables dans un outil de débogage tel que Firebug. Ainsi, en affichant une page HTML qui existe (ici, `index.html`), on obtient la figure 10-10.

Le `statusCode` est bien positionné à la valeur 200 et les en-têtes par défaut sont envoyés vers le navigateur par le serveur Node (dans la partie Response Headers).

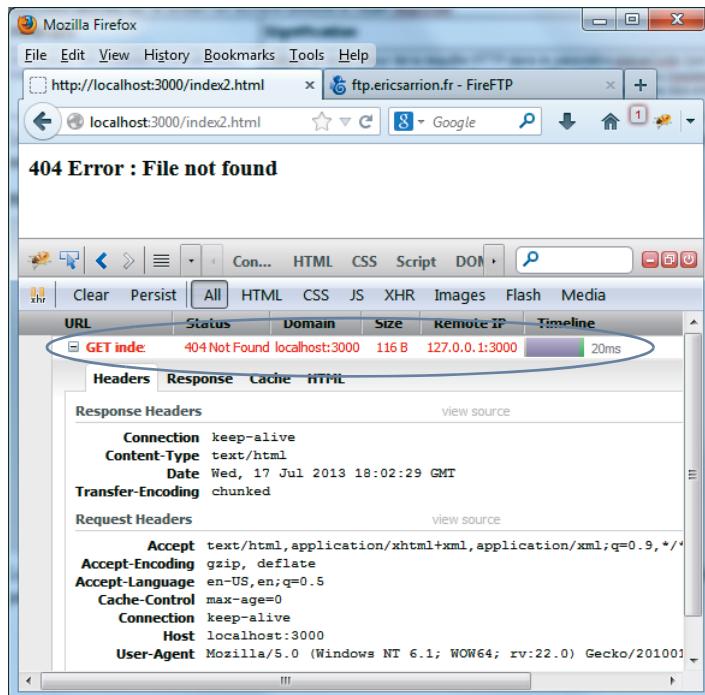
Si on obtient le code d'erreur 404, en utilisant une URL inconnue (`index2.html`), l'affichage est alors celui de la figure 10-11.

Figure 10–10

Visualisation des en-têtes dans Firebug

**Figure 10–11**

URL inconnue dans Firebug



On obtient maintenant le `statusCode` 404, entouré dans la fenêtre de débogage afin de montrer l'erreur.

Événement associé à l'objet response

L'objet `response` est un stream en écriture qui doit se terminer en appelant `response.end()`. Si pour une raison quelconque on stoppe l'envoi de la réponse avant que l'instruction `response.end()` ait été exécutée (par exemple, en fermant le navigateur ou en arrêtant le chargement de la page HTML), Node produit l'événement `close` sur l'objet `response`, afin d'informer l'utilisateur.

Tableau 10-5 Événement géré par l'objet response

Événement	Signification
<code>close</code>	La réponse n'a pas pu être entièrement reçue par le navigateur, car celui-ci a été fermé ou le chargement de la page a été interrompu.

Utiliser l'événement close sur l'objet response

```
var http = require("http");
var server = http.createServer(function(request, response) {
  console.log(request.url);
  setTimeout(function() {
    response.end("Bonjour");
  }, 5000);
  response.on("close", function() {
    console.log("close response");
  });
});
server.listen(3000);
```

L'utilisation de la méthode `setTimeout()` permet de différer l'envoi de la fin de la réponse (ici, de cinq secondes). Si entre-temps le navigateur est fermé ou bien le chargement de la page arrêté, l'événement `close` est déclenché sur l'objet `response` et le message "`close response`" est affiché dans la console du serveur. Si on attend cinq secondes, la fin de la réponse est envoyée et l'événement `close` n'est jamais déclenché.

Événements associés au serveur

Nous avons précédemment utilisé l'événement `request` sur l'objet `server` créé par la méthode `http.createServer()`. D'autres événements sont disponibles sur l'objet `server`, ils sont décrits dans le tableau 10-6.

Tableau 10–6 Événements gérés par le serveur HTTP

Événement	Signification
<code>request</code>	Un client HTTP a effectué une requête au serveur. La fonction de callback de traitement de l'événement prend les deux paramètres <code>request</code> et <code>response</code> définis précédemment.
<code>close</code>	Le serveur a effectué <code>server.close()</code> et chacun des clients a également clôturé sa connexion avec lui. Une fois que l'événement <code>close</code> a été émis avec ces deux conditions, le serveur ne peut plus recevoir de nouvelles connexions. Remarquons que tant qu'un client n'a pas fermé sa connexion avec le serveur, d'autres clients peuvent toujours se connecter même si le serveur a effectué <code>server.close()</code> .

Méthodes définies sur l'objet server

Les principales méthodes définies sur l'objet `server` retourné par la méthode `http.createServer()` sont les méthodes `server.listen()` et `server.close()`.

Tableau 10–7 Méthodes définies sur le serveur HTTP

Méthode	Signification
<code>server.listen(port, [callback])</code>	Met le serveur à l'écoute du port indiqué. La fonction de callback est optionnelle et elle est déclenchée lorsque le serveur est à l'écoute des connexions des clients (la fonction de callback correspond à l'événement <code>listening</code>).
<code>server.close ([callback])</code>	Effectue la fermeture du serveur HTTP, qui n'acceptera de nouvelles connexions que si toutes les connexions déjà ouvertes seront fermées par les clients HTTP. La fonction de callback, optionnelle, sera appelée lorsque l'événement <code>close</code> sera déclenché (voir tableau 10-6).

Créer un client HTTP

Comme pour les serveurs TCP et UDP, il est possible de créer un client HTTP qui se connectera à un serveur HTTP. Pour cela, on utilise l'une des deux méthodes `http.request()` ou `http.get()` définies par Node dans le module `http`.

Utiliser la méthode `http.request()`

La méthode `http.request(options, callback)` est la méthode générique de Node permettant d'effectuer des requêtes HTTP vers un serveur HTTP (Node ou autre).

Si la requête est de type GET (la majorité des cas), on utilisera alors une forme plus simple avec la méthode `http.get()`.

Le fonctionnement de la méthode `http.request(options, callback)` est le suivant. Elle retourne un objet `request` de classe `http.ClientRequest` qui est un stream en écriture sur lequel il faudra au moins appeler la méthode `request.end()` pour indiquer la fin d'écriture sur le stream et donner la main au serveur. Le serveur peut alors traiter la requête reçue du client et envoyer sa réponse qui sera reçue via la fonction de callback indiquée en paramètre de `http.request()`.

Les éléments envoyés vers le serveur via le stream en écriture peuvent, par exemple, correspondre à un fichier que l'on transfère sur le serveur.

La fonction de callback indiquée dans `http.request(options, callback)` possède un paramètre `response` qui est un stream en lecture, permettant de lire la réponse du serveur. Comme pour tout stream en lecture, on écoute les événements `data` reçus sur cet objet `response`, qui sont la réponse du serveur. Cette réponse peut donc être exploitée dans la fonction de callback indiquée dans `http.request()` et être reçue en plusieurs morceaux, correspondant aux différents événements `data` reçus.

Une autre forme de la méthode `http.request()` consiste à écouter les événements `response` se produisant sur l'objet `request` retourné par `http.request()` au lieu d'indiquer une fonction de callback en paramètre de la méthode.

Si la fonction de callback (ou l'événement `response`) n'est pas appelée, cela signifie qu'une erreur s'est produite lors de la requête au serveur. Dans ce cas, l'événement `error` est déclenché sur l'objet `server`.

Tableau 10–8 Méthode `http.request()`

Méthode	Signification
<code>http.request(options, callback)</code>	Effectue une requête HTTP au serveur indiqué dans le paramètre <code>options</code> . Retourne un objet de classe <code>http.ClientRequest</code> , qui est un stream en écriture qui servira à envoyer des éléments au serveur (par exemple, un fichier à transférer sur le serveur). La fonction de callback est appelée lorsque la connexion au serveur est effectuée avec succès (sinon l'événement <code>error</code> est déclenché sur l'objet retourné par <code>http.request()</code>). Elle possède un paramètre <code>response</code> qui est un stream en lecture permettant de lire la réponse du serveur dans les événements <code>data</code> du stream.

Les principales options utilisables dans la méthode `http.request(options, callback)` sont les suivantes.

Tableau 10–9 Options utilisables dans la méthode `http.request()`

Option	Signification
<code>options.hostname</code>	Nom de domaine (ou adresse IP) du serveur à accéder. Par défaut, "localhost".
<code>options.port</code>	Port à accéder sur le serveur. Par défaut, 80.
<code>options.path</code>	Chemin de l'URL à accéder sur le serveur. Par défaut, "/", soit la racine du site.
<code>options.method</code>	Méthode utilisée pour communiquer avec le serveur ("GET", "POST", "DELETE" ou "PUT"). Par défaut, "GET".

Montrons un premier exemple d'utilisation de la méthode `http.request()`. On veut récupérer le code HTML du site nodejs.org, pour la page d'accueil du site.

Utiliser la méthode `http.request(options, callback)`

```
var http = require("http");
var options = {
  hostname: "nodejs.org",
  port: 80,
  path: "/",
  method: "GET"
};

var request = http.request(options, function(response) {
  console.log("**** statusCode : " + response.statusCode);
  console.log("**** headers: " + JSON.stringify(response.headers));
  response.setEncoding("utf8");
  response.on("data", function (chunk) {
    console.log("**** Données reçues : " + chunk);
  });
});

request.on("error", function(e) {
  console.log("Erreur dans http.request() : " + e.message);
});

request.end();
```

Nous déclarons la variable `options` contenant les caractéristiques de la requête à effectuer. Seule l'option `hostname` est obligatoire, car les trois autres options correspondent aux valeurs par défaut.

La requête vers le serveur est ensuite effectuée au moyen de la méthode `http.request()`. La fonction de callback indiquée en paramètre comporte le paramètre `response` ayant les propriétés `statusCode` et `headers`. La méthode `JSON.stringify(response.headers)` sert à retourner sous forme de chaîne de caractères un objet au format JSON (ce qui est le cas des objets JavaScript).

De plus, l'objet `response` étant un stream en lecture, il peut recevoir l'événement `data` correspondant aux données lues sur le stream. La méthode `response.setEncoding("utf8")` sert à indiquer l'encodage des données à afficher, sinon elles s'affichent sous forme de buffer d'octets (classe `Buffer`).

Si la requête échoue, l'événement `error` positionné sur l'objet `request` permet d'afficher le message d'erreur. Enfin, il faut obligatoirement terminer l'envoi de la requête en effectuant l'instruction `request.end()`, sinon rien ne se passe.

Figure 10-12
Code HTML récupéré par
`http.request()`

```
C:\Users\Eric\Documents\Node.js>node test.js
**** statusCode : 200
**** headers: {"server":"nginx","date":"Fri, 19 Jul 2013 13:13:1
type":"text/html","content-length":8684,"last-modified":"Fri,
3:00 GMT","connection":"keep-alive","accept-ranges":"bytes"}
**** Données reçues : <!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link type="image/x-icon" rel="icon" href="favicon.ico">
    <link type="image/x-icon" rel="shortcut icon" href="favicon.
    <link rel="stylesheet" href="pipe.css">
    <link rel="stylesheet" href="sh_vim-dark.css">
    <link rel="alternate"
      type="application/rss+xml"
      title="node blog"
      href="http://feeds.feedburner.com/nodejs/123123123">
  </head>
  <body id="front">
    <div id="intro">
      Node.js is a platform built on <a
      href="http://code.google.com/p/v8/">Chrome's JavaScript
      for easily building fast, scalable network applications.
      uses an event-driven, non-blocking I/O model that makes
      lightweight and efficient, perfect for data-intensive re
      applications that run across distributed devices.</p>
      <p>Current Version: v0.10.13</p>
      <div class=buttons>
        <a href="http://nodejs.org/dist/v0.10.13/node-v0.10.13.t
        tton downloadbutton" id="downloadbutton">INSTALL</a>
        <a href="download/" class=button id=
**** Données reçues : all-dl-options"Downloads</a
        ><a href="/" class="button" id="docsbutton">API Docs<
        </div>
        <a href="http://github.com/joeyent/node"><img class="fork
        nodejs.org/images/forkme.png" alt="Fork me on GitHub"></a>
      </div>
      <div id="quotes" class="clearfix"><h2>Node.js in the Indust
      ass="jaleoo">
```

On peut voir ici la réception des événements `data`, dans lesquels on affiche les données reçues.

Utiliser l'événement response plutôt que la fonction de callback

La fonction de callback indiquée en paramètre de la méthode `http.request(options, callback)` peut être déportée dans le traitement de l'événement `response`, déclenché lorsque le client a reçu la réponse du serveur. On peut donc également écrire :

Utiliser l'événement response à la place de la fonction de callback

```
var http = require("http");
var options = {
  hostname: "nodejs.org",
  port: 80,
  path: "/",
  method: "GET"
};

var request = http.request(options);

request.on("response", function(response) {
  console.log("***** statusCode : " + response.statusCode);
  console.log("***** headers: " + JSON.stringify(response.headers));
  response.setEncoding("utf8");
  response.on("data", function (chunk) {
    console.log("***** Données reçues : " + chunk);
  });
});

request.on("error", function(e) {
  console.log("Erreur dans http.request() : " + e.message);
});

request.end();
```

Le fonctionnement est identique au code précédent.

Utiliser l'événement end sur le stream en lecture

On peut améliorer le programme précédent pour afficher les buffers uniquement lorsque le dernier a été reçu, et non pas au fil de l'eau comme ici. Pour cela, le stream en lecture reçoit l'événement `end` lorsque le dernier buffer a été reçu. Il suffit donc d'afficher l'ensemble des buffers uniquement à ce moment-là, et non plus dans chaque événement `data`.

Afficher les éléments reçus dans l'événement end

```
var http = require("http");
var options = {
  hostname: "nodejs.org",
  port: 80,
  path: "/",
  method: "GET"
};

var buffer = "";

var request = http.request(options, function(response) {
  console.log("**** statusCode : " + response.statusCode);
  console.log("**** headers: " + JSON.stringify(response.headers));
  response.setEncoding("utf8");
  response.on("data", function (chunk) {
    buffer += chunk;
  });
  response.on("end", function () {
    console.log("**** Données reçues : " + buffer);
  });
});

request.on("error", function(e) {
  console.log("Erreur dans http.request() : " + e.message);
});

request.end();
```

L'événement `data` ne sert plus qu'à concaténer les caractères reçus dans une chaîne globale, ici `buffer`. Cette chaîne est ensuite affichée dans l'événement `end`, signifiant que le stream en lecture est terminé.

Figure 10–13

Code HTML récupéré par
http.request()



```
C:\Users\Eric\Documents\Node.js>node test.js
*** statusCode : 200
*** headers: {'server': 'nginx', 'date': 'Fri, 19 Jul 2013 14:22:22 GMT', 'content-type': 'text/html', 'content-length': '8630', 'last-modified': 'Fri, 2000-07-19T14:22:22Z', 'connection': 'keep-alive', 'accept-ranges': 'bytes'}
*** Données reçues : <!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link type="image/x-icon" rel="icon" href="favicon.ico">
    <link type="image/x-icon" rel="shortcut icon" href="favicon.ico">
    <link rel="stylesheet" href="pipe.css">
    <link rel="stylesheet" href="sh_vim-dark.css">
    <link rel="alternate" type="application/rss+xml" title="node blog" href="http://feeds.feedburner.com/node.js/123123123" />
    <title>node.js</title>
  </head>
  <body id="front">
    <div id="intro">
      
      <p>Node.js is a platform built on <a href="http://code.google.com/p/v8/">Chrome's JavaScript engine</a> for easily building fast, scalable network applications. It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.</p>
      <p>Current Version: v0.10.13</p>
      <div class="buttons">
        <a href="http://nodejs.org/dist/v0.10.13/node-v0.10.13.tgz" downloadbutton" id="downloadbutton">INSTALL</a>
        <a href="download/" class="button" id="all-dl-options">Download Options</a>
        <a href="api/" class="button" id="docsbutton">API Docs</a>
        <a href="http://github.com/joyent/node"></a>
      </div>
      <div id="quotes" class="clearfix"><h2>Node.js in the Industry</h2><div>The software stack is built in Node. The key benefits for us are: ease, fast development and a huge flexibility to do both high level functionality.<br><a href="http://backbeam.io">Alberto Gimeno</a></div>!!!</div>
```

Envoyer les informations reçues dans un fichier

Une suite de cet exemple consisterait à envoyer les informations reçues du serveur HTTP vers un fichier local, plutôt que de les afficher dans la console. Le programme précédent est modifié pour créer un nouveau stream en écriture correspondant au fichier dans lequel on souhaite écrire les éléments reçus.

Écrire les éléments reçus dans un fichier

```
var http = require("http");
var fs = require("fs");
var options = {
  hostname: "nodejs.org",
  port: 80,
  path: "/",
  method: "GET"
};

var request = http.request(options, function(response) {
  console.log("**** statusCode : " + response.statusCode);
  console.log("**** headers: " + JSON.stringify(response.headers));
  response.setEncoding("utf8");
  var fichier = fs.createWriteStream("resultat.txt");
  response.pipe(fichier);
});

request.on("error", function(e) {
  console.log("Erreur dans http.request() : " + e.message);
});

request.end();
```

Nous n'utilisons plus les événements `data` ou `end` afin de collecter les informations. En effet, le stream en lecture est redirigé (`pipe`) vers le stream en écriture correspondant au fichier à écrire.

Utiliser la méthode `http.get()`

Afin de simplifier l'utilisation de la méthode `http.request()`, Node a créé la méthode `http.get(options, callback)`. Le principe est le même que dans la méthode `http.request()`, mais on suppose ici que la requête est toujours "GET", tandis que l'instruction `request.end()` permettant de lancer la requête vers le serveur est exécutée en interne sans avoir besoin de l'écrire.

Tableau 10–10 Méthode `http.get()`

Méthode	Signification
<code>http.get(options, callback)</code>	Effectue une requête GET HTTP au serveur indiqué dans le paramètre <code>options</code> . Retourne un objet de classe <code>http.ClientRequest</code> , qui est un stream en écriture qui servira à envoyer des éléments au serveur (par exemple, un fichier à transférer sur le serveur). La fonction de callback est appelée lorsque la connexion au serveur est effectuée avec succès (sinon, l'événement <code>error</code> est déclenché sur l'objet retourné par <code>http.get()</code>). Elle possède un paramètre <code>response</code> qui est un stream en lecture permettant de lire la réponse du serveur dans les événements <code>data</code> du stream.

Le programme précédent peut alors s'écrire plus simplement :

Utiliser la méthode `http.get()`

```
var http = require("http");
var fs = require("fs");
var options = {
  hostname: "nodejs.org",
  port: 80,
  path: "/"
};

var request = http.get(options, function(response) {
  console.log("**** statusCode : " + response.statusCode);
  console.log("**** headers: " + JSON.stringify(response.headers));
  response.setEncoding("utf8");
  var fichier = fs.createWriteStream("resultat.txt");
  response.pipe(fichier);
});

request.on("error", function(e) {
  console.log("Erreur dans http.get() : " + e.message);
});
```

On voit que l'instruction `request.end()` n'est plus nécessaire ici. Le fonctionnement est identique au précédent.

Transmettre des données vers le serveur HTTP

Pour l'instant, nous avons reçu des données du serveur que nous avons ensuite affichées sur la console ou mises dans un fichier. Mais il est également possible, pour un client HTTP, de transmettre des données au serveur. Cette possibilité n'est offerte qu'en utilisant la méthode `http.request()`, car le type de requête à utiliser ne peut pas être "`GET`", mais uniquement "`POST`", "`DELETE`" ou "`PUT`". Ceci exclut donc la possibilité d'utiliser la méthode `http.get()`.

Reprendons un exemple que nous avons précédemment utilisé. On effectue une requête vers un serveur, mais au lieu d'accéder à un serveur quelconque, on accède à un serveur HTTP Node que nous avons mis en place à l'aide de la méthode `http.createServer()`. Voici les codes correspondant au serveur HTTP et au client HTTP, réunis pour l'instant dans le même fichier.

Serveur HTTP et client HTTP

```
var http = require("http");

///////////////
// SERVEUR HTTP

var server = http.createServer();
server.on("request", function(request, response) {
    var html = "";
    html += "<html><head><meta charset=utf-8></head>";
    html += "<body><p>Bienvenue sur le serveur HTTP de Node !</p></body>";
    html += "</html>";
    response.end(html);
});
server.listen(3000);

/////////////
// CLIENT HTTP

var options = {
    hostname: "localhost",
    port: 3000,
    method : "GET",
    path: "/"
};

var request = http.request(options, function(response) {
    console.log("**** statusCode : " + response.statusCode);
    response.setEncoding("utf8");
    response.on("data", function(chunk) {
        console.log("**** Données reçues : " + chunk);
    });
});

request.on("error", function(e) {
    console.log("Erreur dans http.get() : " + e.message);
});

request.end();
```

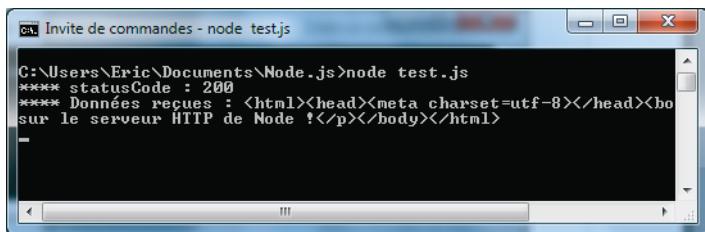
Le code du serveur HTTP est classique. Il renvoie simplement, via le paramètre `response`, le code HTML de la page. Ce serveur est lancé sur le port 3000.

Le code du client HTTP est également classique. Il effectue une requête `GET` vers le serveur `localhost`, sur le port 3000. Il est donc censé interroger notre serveur HTTP précédemment mis en place. Lorsque le client HTTP reçoit la réponse du serveur, il affiche le `statusCode` et les données reçues via le stream de lecture associé au paramètre `response`.

L'exécution du fichier contenant ce programme produit le résultat présenté à la figure 10-14.

Figure 10-14

Le client affiche les données reçues du serveur.



Le problème maintenant va être de transmettre des informations du client HTTP vers le serveur. Pour cela, le type de requête doit être "POST", "PUT" ou "DELETE", mais en aucun cas "GET". Les informations à transmettre sont envoyées via l'objet `request` associé au client HTTP, qui est un stream en écriture. On pourra donc effectuer des instructions `request.write(data)` et `request.end(data)` afin de transmettre ces données au serveur HTTP.

Le serveur HTTP récupérera ces données dans la fonction de callback utilisée lors de la création du serveur (ou dans le traitement de l'événement `request` sur l'objet `server`, ce qui revient au même). Le paramètre `request` utilisé dans la fonction de callback est un stream en lecture qui récupère les données envoyées depuis les clients HTTP (uniquement si le mode de transmission est "POST", "DELETE" ou "PUT"). On transforme le code précédent comme suit :

Transmettre des données du client HTTP vers le serveur HTTP

```
var http = require("http");

///////////////////////////////
// SERVEUR HTTP

var server = http.createServer();
server.on("request", function(request, response) {
    request.on("data", function(chunk) {
        console.log("Le serveur a reçu : " + chunk);
    });
    var html = "";
    html += "<html><head><meta charset=utf-8></head>";
    html += "<body><p>Bienvenue sur le serveur HTTP de Node !</p></body>";
    html += "</html>";
    response.end(html);
});
server.listen(3000);
```

```
///////////////
// CLIENT HTTP

var options = {
  hostname: "localhost",
  port: 3000,
  method : "POST",
  path: "/"
};

var request = http.request(options, function(response) {
  console.log("**** statusCode : " + response.statusCode);
  response.setEncoding("utf8");
  response.on("data", function(chunk) {
    console.log("**** Données reçues : " + chunk);
  });
});

request.on("error", function(e) {
  console.log("Erreur dans http.get() : " + e.message);
});

request.end("Données transmises au serveur");
```

Dans le client HTTP :

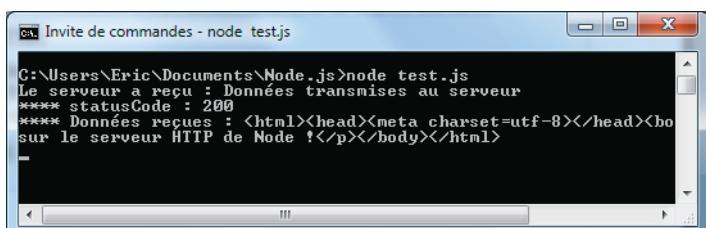
- Nous avons transformé la requête en type "**POST**" (tout sauf "**GET**").
- Puis nous avons indiqué les données à transmettre au moyen de **request.end(data)**. On aurait également pu utiliser **request.write(data)** pour transmettre d'autres données, à condition de terminer par **request.end(data)**.

Tandis que dans le serveur HTTP :

- Nous avons mis en place l'événement **data** sur le paramètre **request** de la fonction de callback du serveur HTTP. Cet événement **data** sera activé pour chaque **request.write(data)** ou **request.end(data)** effectué par le client HTTP symbolisé ici par la variable **request**.

Vérifions que cela fonctionne :

Figure 10-15
Transmission de données du client vers le serveur



Le serveur HTTP reçoit bien la chaîne de caractères transmise par le client HTTP. Si le client transmet plusieurs données via plusieurs instructions `request.write(data)`, elles sont reçues par le serveur HTTP.

Transmettre plusieurs paquets de données au serveur HTTP

```
var http = require("http");

///////////////
// SERVEUR HTTP

var server = http.createServer();
server.on("request", function(request, response) {
  request.on("data", function(chunk) {
    console.log("Le serveur a reçu : " + chunk);
  });
  var html = "";
  html += "<html><head><meta charset=utf-8></head>";
  html += "<body><p>Bienvenue sur le serveur HTTP de Node !</p></body>";
  html += "</html>";
  response.end(html);
});
server.listen(3000);

///////////////
// CLIENT HTTP

var options = {
  hostname: "localhost",
  port: 3000,
  method: "POST",
  path: "/"
};

var request = http.request(options, function(response) {
  console.log("**** statusCode : " + response.statusCode);
  response.setEncoding("utf8");
  response.on("data", function(chunk) {
    console.log("**** Données reçues : " + chunk);
  });
});

request.on("error", function(e) {
  console.log("Erreur dans http.get() : " + e.message);
});

request.write("Données 1 transmises au serveur");
request.write("Données 2 transmises au serveur");
request.end("Données 3 transmises au serveur");
```

Figure 10–16

Plusieurs écritures du client vers le serveur

Le serveur reçoit les données transmises via l'événement `data` sur le stream en lecture associé au paramètre `request` de la fonction de callback.

Dissocier le programme du client et celui du serveur

L'application précédente réunit dans un même fichier le programme du client et celui du serveur. Dans une application réelle, les deux programmes sont le plus souvent séparés dans deux fichiers distincts.

Fichier server.js

```
var http = require("http");
var server = http.createServer();
server.on("request", function(request, response) {
  request.on("data", function(chunk) {
    console.log("Le serveur a reçu : " + chunk);
  });
  var html = "";
  html += "<html><head><meta charset=utf-8></head>";
  html += "<body><p>Bienvenue sur le serveur HTTP de Node !</p></body>";
  html += "</html>";
  response.end(html);
});
server.listen(3000);
```

Fichier client.js

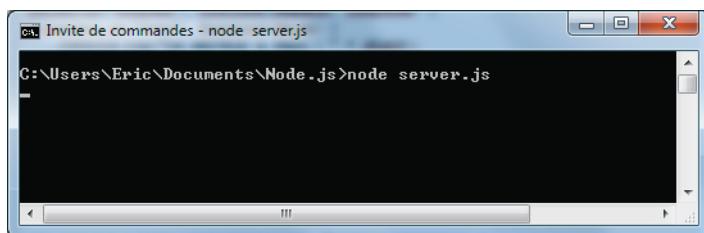
```
var http = require("http");
var options = {
  hostname: "localhost",
  port: 3000,
  method: "POST",
  path: "/"
};
```

```
var request = http.request(options, function(response) {  
    console.log("**** statusCode : " + response.statusCode);  
    response.setEncoding("utf8");  
    response.on("data", function(chunk) {  
        console.log("**** Données reçues : " + chunk);  
    });  
});  
  
request.on("error", function(e) {  
    console.log("Erreur dans http.get() : " + e.message);  
});  
  
request.write("Données 1 transmises au serveur");  
request.write("Données 2 transmises au serveur");  
request.end("Données 3 transmises au serveur");
```

Pour tester ce programme, il faut d'abord lancer le serveur, afin qu'il soit prêt à recevoir les connexions des clients.

Figure 10–17

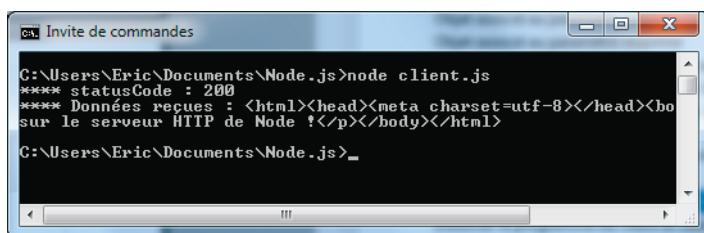
Le serveur est lancé.



Puis on lance le client.

Figure 10–18

Le client est lancé.



La connexion avec le serveur HTTP est immédiatement effectuée. Les données transmises par le client s'affichent alors sur la console du serveur.

Figure 10-19

Console du serveur une fois que le client a été lancé.

```
C:\Users\Eric\Documents\Node.js>node server.js
Le serveur a reçu : Données 1 transmises au serveur
Le serveur a reçu : Données 2 transmises au serveur
Le serveur a reçu : Données 3 transmises au serveur
```

Transmettre un fichier vers le serveur (upload de fichier)

Appliquons l'exemple précédent au cas particulier où l'on souhaite uploader un fichier (du client HTTP vers le serveur HTTP). Le serveur devra stocker le fichier reçu dans son répertoire principal.

Fichier server.js

```
var http = require("http");
var server = http.createServer();
var fs = require("fs");
server.on("request", function(request, response) {
  var fichier = fs.createWriteStream("mylogo.png");
  request.pipe(fichier);
  console.log("Le fichier a été sauvegardé sur le serveur !");
  response.end("Le fichier a été transmis au serveur !");
});
server.listen(3000);
```

Les données transmises du client vers le serveur se trouvent dans le *stream en lecture* associé au paramètre `request`. Ce stream en lecture est envoyé vers le *stream en écriture* associé au fichier à écrire sur disque, ici `mylogo.png`.

Fichier client.js

```
var http = require("http");
var fs = require("fs");
var options = {
  hostname: "localhost",
  port: 3000,
  method: "POST",
  path: "/"
};

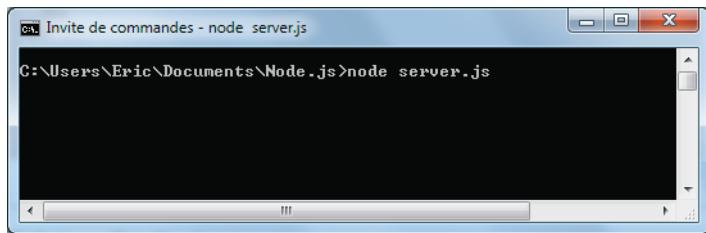
var request = http.request(options, function(response) {
  console.log("**** statusCode : " + response.statusCode);
  response.setEncoding("utf8");
```

```
response.on("data", function(chunk) {  
    console.log("**** Données reçues : " + chunk);  
});  
  
request.on("error", function(e) {  
    console.log("Erreur dans http.request() : " + e.message);  
});  
  
var fichier = fs.createReadStream("logo.png");  
fichier.pipe(request);
```

Les données à envoyer au serveur correspondent au contenu du fichier PNG nommé `logo.png`. Pour les transmettre, il suffit d'effectuer un `pipe` d'un stream en lecture (représenté par le stream associé au fichier `logo.png`) vers un stream en écriture (représenté ici par la variable `request` associée au client HTTP).

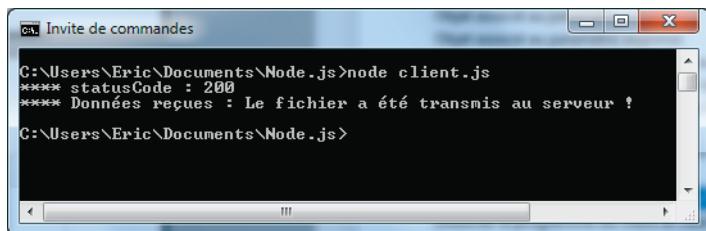
Pour tester ce programme, démarrons tout d'abord le serveur, afin qu'il soit prêt à recevoir le fichier à transmettre.

Figure 10–20
Le serveur est lancé.



Le serveur est maintenant en attente d'une connexion d'un client. Démarrons le client.

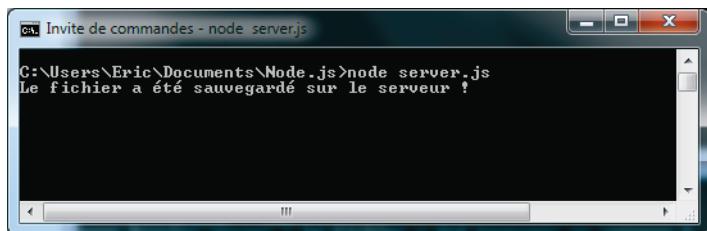
Figure 10–21
Le client envoie un fichier
au serveur.



Le fichier `Logo.png` est aussitôt transmis au serveur, qui retourne au client la chaîne "[Le fichier a été transmis au serveur](#)". Sur la console du serveur, on a maintenant le message suivant.

Figure 10-22

Console du serveur une fois que le client a été lancé.



```
C:\Users\Eric\Documents\Node.js>node server.js
Le fichier a été sauvegardé sur le serveur !
```

Dans le répertoire du serveur, on peut maintenant voir le fichier [myLogo.png](#) correspondant au fichier sauvegardé sur le serveur.

11

Utiliser les web sockets avec socket.io

Les *web sockets* sont une technologie récente qui permet de faciliter le dialogue entre un client et un serveur. Il est usuel, avec un serveur HTTP, que les clients effectuent des requêtes au serveur afin que ce dernier leur transmette la réponse. C'est donc le client (ici, le navigateur) qui est à l'origine de la demande vers le serveur, mais jamais l'inverse. Ainsi, le serveur n'est jamais à l'initiative de l'envoi des requêtes au navigateur (sauf si ce dernier en fait la demande).

Avec les web sockets, ce comportement n'a plus cours. Une fois la connexion entre un client et un serveur établie (grâce aux web sockets), la communication s'effectue à tout instant dans les deux sens. Cette technologie ouvre de nouvelles possibilités pour les applications client serveur, en particulier avec Node.

Node implémente cette technologie au moyen d'un module nommé `socket.io`. Nous étudions son utilisation dans ce chapitre.

Installer le module `socket.io`

Le module `socket.io` s'installe au moyen de la commande `npm install socket.io`, tapée depuis le répertoire principal de l'application Node.

Figure 11-1
Installation du module
socket.io

```
C:\>Users\Eric\Documents\Node.js>npm install socket.io
npm WARN package.json @ No README.md file found!
npm http GET https://registry.npmjs.org/socket.io
npm http 200 https://registry.npmjs.org/socket.io
npm http GET https://registry.npmjs.org/socket.io-client/0.9.16
npm http GET https://registry.npmjs.org/policyfile/0.0.4
npm http GET https://registry.npmjs.org/base64id/0.1.0
npm http GET https://registry.npmjs.org/redis/0.7.3
npm http 200 https://registry.npmjs.org/policyfile/0.0.4
npm http GET https://registry.npmjs.org/policyfile/-/policyfile-
npm http 200 https://registry.npmjs.org/socket.io-client/0.9.16
npm http GET https://registry.npmjs.org/socket.io-client/-/socket
16.tgz
npm http 200 https://registry.npmjs.org/base64id/0.1.0
npm http 200 https://registry.npmjs.org/redis/0.7.3
npm http GET https://registry.npmjs.org/base64id/-/base64id-0.1.
npm http GET https://registry.npmjs.org/redis/-/redis-0.7.3.tgz
npm http 200 https://registry.npmjs.org/policyfile/-/policyfile-
npm http 200 https://registry.npmjs.org/socket.io-client/-/socket
16.tgz
npm http 200 https://registry.npmjs.org/base64id/-/base64id-0.1.
npm http 200 https://registry.npmjs.org/redis/-/redis-0.7.3.tgz
npm http GET https://registry.npmjs.org/uglify-js/1.2.5
```

Une fois le module installé, les fonctionnalités des web sockets incluses dans le module `socket.io` sont accessibles dans les programmes écrits avec Node.

L'utilisation des web sockets nécessite d'écrire un programme côté serveur et un autre côté client, afin que la communication entre le client et le serveur soit établie.

- Le programme côté serveur sera écrit avec Node, en utilisant les fonctionnalités du module `socket.io` précédemment installé.
- Le programme côté client sera intégré dans une page HTML et écrit en JavaScript traditionnel, mais en incluant un fichier JavaScript permettant d'utiliser les web sockets fournies par le module `socket.io`.

La communication entre le client et le serveur s'effectue par événements. Lorsque le client veut communiquer avec le serveur, il lui envoie un événement que celui-ci peut recevoir et traiter. Le même mécanisme fonctionne dans l'autre sens.

Examinons ces deux types de communications.

Communication du client vers le serveur

Nous voulons ici établir une connexion par web socket entre la page `index.html` et le serveur Node, afin que la page HTML transmette des informations au serveur.

Programme côté client

Considérons d'abord le programme situé du côté client (fichier `index.html`).

Fichier index.html

```
<html>
<head>
  <script src="http://localhost:3000/socket.io/socket.io.js"></script>
  <script>
    var socket = io.connect('http://localhost:3000');
    socket.emit("event1", "Message transmis du client vers le serveur");
  </script>
</head>

<body>
</body>

</html>
```

Le code du fichier `index.html` se concentre principalement sur la partie JavaScript. On inclut d'abord le fichier `socket.io.js` fourni par le module `socket.io`, qui permet d'utiliser les fonctionnalités des web sockets depuis une page HTML. Le fichier `socket.io.js` est situé sur le serveur Node dans le répertoire `socket.io`, et il est inclus en utilisant l'URL indiquée. Le port 3000 est ici utilisé car le serveur HTTP sera démarré sur ce port.

Une fois le fichier `socket.io.js` inclus dans la page HTML, on peut utiliser les fonctionnalités des web sockets à travers la variable `io` déclarée dans le fichier JavaScript inclus. La méthode `io.connect(url)` permet de se connecter en web sockets sur le serveur ayant l'URL indiquée. Un objet `socket` permettant la communication avec le serveur est retourné par la méthode, permettant d'émettre un événement à l'aide de la méthode `socket.emit(event, params)`.

Tableau 11-1 Méthodes des web sockets utilisables côté client

Méthode	Signification
<code>io.connect(url)</code>	Crée une liaison avec le serveur et retourne un objet <code>socket</code> qui servira à la communication entre ce client et le serveur.
<code>socket.emit(event, params)</code>	Envoie un événement au serveur en lui transmettant les paramètres indiqués, qui sont soit sous forme d'objet JSON, soit sous forme de chaîne de caractères.

Programme côté serveur

Le programme du serveur consiste à établir lui aussi une connexion, avec les clients qui en font la demande. Une fois la connexion établie avec un client, la communication par web sockets peut s'effectuer. Voici le programme du serveur (fichier `server.js`).

Fichier server.js

```
var http = require("http");
var fs = require('fs');

var server = http.createServer(function(request, response) {
    var index = fs.createReadStream("index.html");
    index.pipe(response);
});

server.listen(3000);
var io = require("socket.io").listen(server);

io.sockets.on("connection", function (socket) {
    console.log("Un client s'est connecté");
    socket.on("event1", function(data) {
        console.log(data);
    });
});
```

Un serveur HTTP est créé au moyen de la méthode `http.createServer()`, comme expliqué dans le chapitre précédent. Le serveur est mis à l'écoute du port 3000, qui est le même port que celui avec lequel le client s'est connecté (cf. fichier `index.html` précédent).

Pour que la connexion avec le serveur soit effectuée par web sockets, on utilise l'instruction `require("socket.io").listen(server)` du module `socket.io`. Cela crée une liaison par web sockets entre le serveur et le client, sur le port du serveur précédemment utilisé. Cette instruction retourne un objet (ici, nommé `io`) permettant de recevoir les demandes de connexion des clients (chaque connexion de client est effectuée dans la page HTML au moyen de l'instruction `io.connect(url)` vue précédemment).

Une demande de connexion de client est reçue à travers l'événement `connection`. La fonction de callback associée au traitement de cet événement reçoit un objet `socket` en paramètre, qui identifie la connexion entre le serveur et le client. À l'intérieur de ce traitement, on gère les autres événements que le client est susceptible d'envoyer au serveur, notamment l'événement `event1` envoyé dans le fichier `index.html` au moyen de l'instruction `socket.emit(event, params)`. Ces nouveaux événements sont gérés sur l'objet `socket` associé à la connexion, qui peut ainsi gérer autant d'événements que souhaité.

Attardons-nous sur le code du serveur HTTP. Celui-ci renvoie ici systématiquement le fichier `index.html`. En effet, si on veut que le code JavaScript du fichier `index.html` soit exécuté, il faut retourner ce fichier au navigateur afin qu'il soit traité. L'exécution du code JavaScript inscrit dans le fichier `index.html` permet ainsi la connexion au serveur par web sockets et l'envoi de l'événement `event1`.

Tableau 11–2 Méthode des web sockets utilisables côté serveur

Méthode	Signification
<code>require("socket.io").listen(en(server))</code>	Transforme le serveur HTTP en serveur pouvant dialoguer par web sockets avec les clients qui vont se connecter au serveur. Retourne un objet <code>io</code> sur lequel on pourra écouter les connexions des clients.

En plus de la méthode `require("socket.io").listen(server)`, `socket.io` fournit l'événement `connection` qui permet de détecter la connexion d'un client sur le serveur (effectuée au moyen de `io.connect(url)` vue précédemment).

Tableau 11–3 Événement connection

Événement	Signification
<code>connection</code>	Événement reçu sur l'objet <code>io.sockets</code> , servant à recevoir les connexions des utilisateurs sur le serveur. Une fois cet événement réceptionné, tous les autres événements transmis par les clients sont reçus dans la fonction de callback traitant cet événement.

Exécution du programme

On démarre d'abord le serveur au moyen de la commande `node server.js`.

Figure 11–2

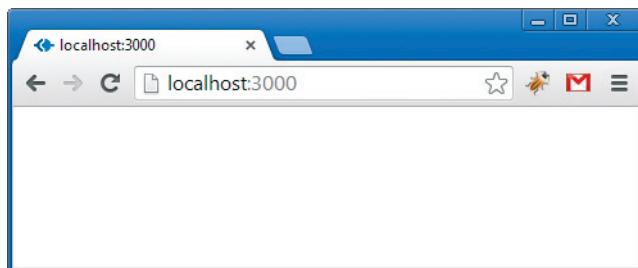
Le serveur est lancé.



Un client se connecte via un navigateur en tapant l'URL `http://localhost:3000` dans la barre d'adresses. N'importe quelle URL située sur le serveur (associée au bon port) active le programme du serveur qui renvoie le fichier `index.html`.

Figure 11–3

Connexion d'un client au serveur



La requête `localhost:3000` a été traitée par le serveur, qui renvoie le fichier `index.html` (ici, vide). Mais le script inclus dans le fichier s'est exécuté, comme on peut le voir dans la console du serveur.

Figure 11–4
Console du serveur
une fois que le client
s'est connecté au serveur.

```
C:\Users\Eric\Documents\Node.js>node server.js
info  - socket.io started
debug - served static content /socket.io.js
debug - client authorized
info  - handshake authorized gz7J2UQMz2PtRHFPvTL0
debug - setting request GET /socket.io/1/websocket/gz7J2UQMz2
debug - set heartbeat interval for client gz7J2UQMz2PtRHFPvTL0
debug - client authorized for
debug - websocket writing 1:::
Un client s'est connecté
Message transmis du client vers le serveur
```

Le serveur a détecté la connexion d'un client (du simple fait que le client se connecte sur le serveur en indiquant son URL et son port) et également l'événement `event1` émis par le client dès l'affichage de la page `index.html` (d'où l'affichage de "Message transmis du client vers le serveur").

Si de nouveaux clients se connectent, le serveur reçoit les événements `connection` le lui indiquant. Mais si on arrête le serveur et qu'on le redémarre ensuite, ces événements `connection` sont également transmis au serveur (pour les navigateurs qui étaient encore connectés au serveur).

Déconnexion d'un client

On a vu comment le serveur recevait la connexion d'un client au moyen de l'événement `connection`. Un client qui se déconnecte (par exemple, en fermant son navigateur) génère un nouvel événement nommé `disconnect`. Celui-ci est géré dans le traitement de l'événement `connection`, sur l'objet `socket` qui lui est transmis.

Gérer la déconnexion d'un client (fichier `server.js`)

```
var http = require("http");
var fs = require('fs');

var server = http.createServer(function(request, response) {
  var index = fs.createReadStream("index.html");
  index.pipe(response);
});

server.listen(3000);
var io = require("socket.io").listen(server);
```

```
io.sockets.on("connection", function (socket) {
  console.log("Un client s'est connecté");
  socket.on("event1", function(data) {
    console.log(data);
  });
  socket.on("disconnect", function() {
    console.log("Un client s'est déconnecté");
  });
});
```

Déconnectons le client précédent en fermant son navigateur. La console du serveur affiche alors :

Figure 11–5

Console du serveur une fois qu'un client s'est déconnecté.

```
debug - websocket writing 2::  
debug - set heartbeat timeout for client gz7J2UQMz2PtRHFPvTL0  
debug - got heartbeat packet  
debug - cleared heartbeat timeout for client gz7J2UQMz2PtRHFP  
debug - set heartbeat interval for client gz7J2UQMz2PtRHFPvTL  
debug - emitting heartbeat for client gz7J2UQMz2PtRHFPvTL0  
debug - websocket writing 2::  
debug - set heartbeat timeout for client gz7J2UQMz2PtRHFPvTL0  
debug - got heartbeat packet  
debug - cleared heartbeat timeout for client gz7J2UQMz2PtRHFP  
debug - set heartbeat interval for client gz7J2UQMz2PtRHFPvTL  
debug - emitting heartbeat for client gz7J2UQMz2PtRHFPvTL0  
debug - websocket writing 2::  
debug - set heartbeat timeout for client gz7J2UQMz2PtRHFPvTL0  
debug - got heartbeat packet  
debug - cleared heartbeat timeout for client gz7J2UQMz2PtRHFP  
debug - set heartbeat interval for client gz7J2UQMz2PtRHFPvTL  
info - transport end <socket end>  
debug - set close timeout for client gz7J2UQMz2PtRHFPvTL0  
debug - cleared close timeout for client gz7J2UQMz2PtRHFPvTL0  
debug - cleared heartbeat interval for client gz7J2UQMz2PtRHFPvTL0  
Un client s'est déconnecté  
debug - discarding transport
```

L'événement `disconnect` s'est déclenché sur l'objet `socket` associé au client, ce qui provoque l'affichage du message dans la console du serveur.

Communication du serveur vers le client

Jusqu'à présent, nous nous sommes intéressés à la communication du client vers le serveur. Voyons maintenant la communication en sens inverse, c'est-à-dire du serveur vers le client.

Programme côté client

Le programme côté client va consister à recevoir et à traiter un événement `event2` reçu du serveur, et à l'afficher dans la page `index.html` actuellement visible.

Fichier index.html

```
<html>
<head>
<meta charset=utf-8>
<script src="http://localhost:3000/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost:3000');
  socket.on("event2", function(data) {
    var p = document.createElement("p");
    p.innerHTML = "Événement event2 : " + data;
    document.body.appendChild(p);
  });
</script>
</head>

<body>
</body>

</html>
```

Le client se connecte au serveur au moyen de l'instruction `io.connect(url)` comme nous l'avions fait précédemment, ce qui retourne un objet `socket`, sur lequel on met en place un gestionnaire d'événements en utilisant la méthode `socket.on(event, callback)` similaire à celle utilisée côté serveur. Les données transmises du serveur vers le client sont indiquées en paramètres de la fonction de callback (paramètre `data`). Elles sont ici encapsulées dans un élément `<p>` qui est inséré à la fin de l'élément `<body>` dans la page HTML.

Le tableau 11-4 liste les méthodes précédemment employées.

Tableau 11-4 Méthodes des web sockets utilisables côté client

Méthode	Signification
<code>io.connect(url)</code>	Crée une liaison avec le serveur et retourne un objet <code>socket</code> qui servira à la communication entre ce client et le serveur.
<code>socket.on(event, callback)</code>	Positionne un gestionnaire d'événements permettant de traiter l'événement <code>event</code> reçu du serveur. La fonction de callback est de la forme <code>function(data)</code> dans laquelle <code>data</code> représente les données transmises (sous forme de chaîne ou d'objet, voir section suivante).

Programme côté serveur

Le programme côté serveur doit maintenant envoyer l'événement `event2` que le client attend.

Fichier server.js

```
var http = require("http");
var fs = require('fs');

var server = http.createServer(function(request, response) {
  var index = fs.createReadStream("index.html");
  index.pipe(response);
});

server.listen(3000);
var io = require("socket.io").listen(server);

io.sockets.on("connection", function (socket) {
  console.log("Un client s'est connecté");
  socket.emit("event2", "Message envoyé du serveur vers le client");
});
```

Lorsqu'un client se connecte, on peut alors lui envoyer un événement comportant un message à transmettre au client. Pour cela, on emploie la méthode `socket.emit(event, params)`, utilisée dans l'événement `connection` indiquant que le client est connecté.

Tableau 11-5 Méthode des web sockets utilisables côté serveur

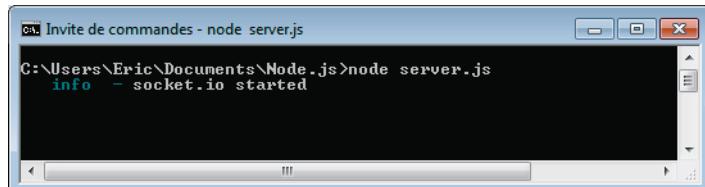
Méthode	Signification
<code>socket.emit(event, params)</code>	Envoie un événement au client en lui transmettant les paramètres indiqués, qui sont soit sous forme d'objet JSON, soit sous forme de chaîne de caractères.

Exécution du programme

On redémarre le serveur au moyen de la commande `node server.js`.

Figure 11-6

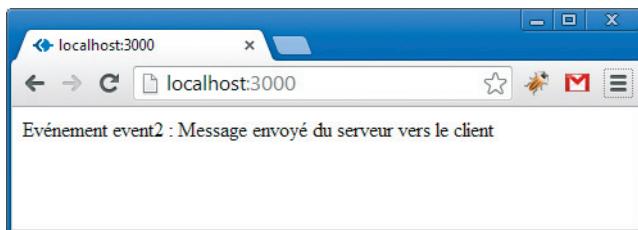
Le serveur est lancé.



Un client accède au serveur via l'URL `http://localhost:3000`. Dès sa connexion au serveur, ce dernier envoie un événement `event2` au client, ce qui provoque sa prise en compte par le client.

Figure 11-7

Connexion d'un client sur le serveur



La console du serveur affiche des informations de log permettant de voir les événements émis vers les clients.

Figure 11-8

Console du serveur une fois qu'un client s'est connecté.

```
C:\Users\Eric\Documents>node server.js
info  - socket.io started
debug - served static content /socket.io.js
debug - client authorized
info  - handshake authorized 3jLMh9CkhMUbjKR6MT9q
debug - setting request GET /socket.io/1/websocket/3jLMh9CkhMUbjKR6MT9q
debug - set heartbeat interval for client 3jLMh9CkhMUbjKR6MT9q
debug - client authorized for
debug - websocket writing 1:::
Un client s'est connecté
debug - websocket writing 5:::<"name":"event2","args":["Message
veur vers le client"]>
debug - emitting heartbeat for client 3jLMh9CkhMUbjKR6MT9q
debug - websocket writing 2:::
debug - set heartbeat timeout for client 3jLMh9CkhMUbjKR6MT9q
debug - got heartbeat packet
debug - cleared heartbeat timeout for client 3jLMh9CkhMUbjKR6MT9q
debug - set heartbeat interval for client 3jLMh9CkhMUbjKR6MT9q
```

Diffuser des informations à plusieurs clients

L'exemple précédent a montré comment le serveur pouvait communiquer avec un client particulier en lui transmettant un événement. Il est possible de diffuser un événement non pas à un seul client, mais à plusieurs. Les méthodes répertoriées dans le tableau 11-6 sont utilisables dans le code du serveur, dans la fonction de traitement de l'événement `connection`.

Tableau 11–6 Méthodes de diffusion des informations aux clients (à utiliser côté serveur)

Méthode	Signification
<code>socket.emit(event, params)</code>	Diffusion vers un seul client, celui représenté par <code>socket</code> .
<code>socket.broadcast.emit(event, params)</code>	Diffusion de l'événement à tous les clients connectés, sauf nous-mêmes (représenté ici par la variable <code>socket</code>).
<code>io.sockets.emit(event, params)</code>	Diffusion de l'événement à tous les clients connectés, y compris nous-mêmes.

Diffuser un message à tous les clients connectés, y compris nous-mêmes (fichier server.js)

```
var http = require("http");
var fs = require('fs');

var server = http.createServer(function(request, response) {
    var index = fs.createReadStream("index.html");
    index.pipe(response);
});

server.listen(3000);
var io = require("socket.io").listen(server);

io.sockets.on("connection", function (socket) {
    console.log("Un client s'est connecté");
    io.sockets.emit("event2", "Message envoyé à tous les clients connectés (y compris nous-même)");
});
```

Diffuser un message à tous les clients connectés, excepté nous-mêmes (fichier server.js)

```
var http = require("http");
var fs = require('fs');

var server = http.createServer(function(request, response) {
    var index = fs.createReadStream("index.html");
    index.pipe(response);
});

server.listen(3000);
var io = require("socket.io").listen(server);

io.sockets.on("connection", function (socket) {
    console.log("Un client s'est connecté");
    socket.broadcast.emit("event2", "Message envoyé à tous les clients connectés (excepté nous-même)");
});
```

Transmission des informations entre le client et le serveur

Les informations échangées entre le client et le serveur sont pour l'instant des chaînes de caractères, mais elles peuvent également s'écrire sous forme d'objet JSON.

Dans l'exemple suivant, on utilise les événements `event1` et `event2` :

- `event1` est émis du client vers le serveur et transmet un objet de la forme `{ msg }` ;
- `event2` est émis du serveur vers le client et transmet un objet de la forme `{ msg }`.

Le champ `msg` de l'objet transmis est affiché par le client ou le serveur lorsqu'ils le reçoivent.

Fichier `index.html`

```
<html>
<head>
<meta charset=utf-8>
<script src="http://localhost:3000/socket.io/socket.io.js"></script>
<script>
var socket = io.connect('http://localhost:3000');

// émission
socket.emit("event1", { msg : "Message transmis du client vers le serveur"
});

// réception
socket.on("event2", function(data) {
    var p = document.createElement("p");
    p.innerHTML = "Evénement event2 : " + data.msg;
    document.body.appendChild(p);
});
</script>
</head>

<body>
</body>

</html>
```

Fichier server.js

```

var http = require("http");
var fs = require('fs');

var server = http.createServer(function(request, response) {
  var index = fs.createReadStream("index.html");
  index.pipe(response);
});

server.listen(3000);
var io = require("socket.io").listen(server);

io.sockets.on("connection", function (socket) {
  // émission
  socket.emit("event2", { msg : "Message transmis du serveur vers le client" });

  //réception
  socket.on("event1", function(data) {
    console.log(data.msg);
  });
});

```

Dans chacun des deux programmes, les informations sont maintenant transmises sous forme d'objets.

Associer des données à une socket

Il est possible d'associer des données à une socket, de façon à les utiliser à tout moment. Ceci n'est possible que du côté serveur. On utilise pour cela les méthodes `socket.set(name, value)` et `socket.get(name, callback)`.

Tableau 11-7 Méthodes set() et get() associées aux web sockets côté serveur

Méthode	Signification
<code>socket.set(name, value)</code>	Association de l'objet <code>value</code> à la clé <code>name</code> de la web socket.
<code>socket.get(name, callback)</code>	Récupération de la clé <code>name</code> associée à la web socket. La valeur est transmise à la fonction de callback qui est de la forme <code>function(err, value)</code> .

Utilisons ces méthodes pour mémoriser le nom et le prénom d'une personne qui se connecte au serveur.

Fichier index.html

```
<html>
<head>
  <meta charset=utf-8>
  <script src="http://localhost:3000/socket.io/socket.io.js"></script>
  <script>
    var socket = io.connect('http://localhost:3000');
    socket.emit("event1", { nom : "Sarrion", prenom : "Eric" });
  </script>
</head>

<body>
</body>

</html>
```

Pour chaque client qui se connecte, on lui associe son nom et son prénom (ici, toujours le même, à savoir "[Sarrion Eric](#)"). Ces données sont ensuite transmises sous forme d'objet au serveur.

Fichier server.js

```
var http = require("http");
var fs = require('fs');

var server = http.createServer(function(request, response) {
  var index = fs.createReadStream("index.html");
  index.pipe(response);
});

server.listen(3000);
var io = require("socket.io").listen(server);

io.sockets.on("connection", function (socket) {
  socket.on("event1", function(data) {
    console.log(data.prenom + " " + data.nom + " s'est connecté");
    socket.set\("personne", data\);
  });
  socket.on("disconnect", function() {
    socket.get\("personne", function\(err, p\) {
      console.log\(p.prenom + " " + p.nom + " s'est déconnecté"\);
    }\);
  });
});
```

Lors de la connexion d'un client (événement `event1`), le nom et le prénom sont stockés dans la clé `personne` de la socket. Puis, lors de la déconnexion (événement `disconnect`), ces informations sont récupérées par `socket.get("personne", ...)`.

Lorsqu'un client se connecte, la console du serveur affiche :

Figure 11–9
Associer des données
à une socket (1)

```
C:\Users\Eric\Documents\Node.js>node server.js
info  - socket.io started
debug - served static content /socket.io.js
debug - client authorized
info  - handshake authorized tqdFIUz4KubTg0tI5uh
debug - setting request GET /socket.io/1/websocket/tqdFIUz4KubIg0tI5uh
debug - set heartbeat interval for client tqdFIUz4KubIg0tI5uh
debug - client authorized for
debug - websocket writing 1:
Eric Sarrion s'est connecté
```

Après fermeture de la fenêtre du navigateur, la console du serveur affiche :

Figure 11–10
Associer des données
à une socket (2)

```
C:\Users\Eric\Documents\Node.js>node server.js
info  - socket.io started
debug - served static content /socket.io.js
debug - client authorized
info  - handshake authorized TuE_A-TJji1HKYJ0JWGT
debug - setting request GET /socket.io/1/websocket/TuE_A-TJji1HKYJ0JWGT
debug - set heartbeat interval for client TuE_A-TJji1HKYJ0JWGT
debug - client authorized for
debug - websocket writing 1:
Eric Sarrion s'est connecté
info  - transport end <socket end>
debug - set close timeout for client TuE_A-TJji1HKYJ0JWGT
debug - cleared close timeout for client TuE_A-TJji1HKYJ0JWGT
debug - cleared heartbeat interval for client TuE_A-TJji1HKYJ0JWGT
Eric Sarrion s'est déconnecté
debug - discarding transport
```

Utiliser plusieurs programmes de traitement des sockets

Le programme serveur que nous avons utilisé jusqu'à présent traite toutes les demandes des utilisateurs qui se connectent sur le serveur, à partir du moment où ces derniers utilisent un événement connu par le serveur.

Il peut être préférable parfois de séparer les traitements sur le serveur, en associant certains événements à une partie de traitement, et d'autres événements à une autre partie. Le module `socket.io` permet cette séparation au moyen de ce qu'il appelle les *namespaces* (espaces de noms). Comme précédemment, on aura une partie de code à écrire du côté client, et une autre du côté serveur.

Programme côté client

On indique dans la méthode `io.connect(url)` une URL comportant l'espace de noms souhaité, par exemple `http://localhost:3000/messages` pour se connecter à la partie de traitement des messages sur le serveur.

Fichier index.html

```
<html>
<head>
<meta charset=utf-8>
<script src="http://localhost:3000/socket.io/socket.io.js"></script>
<script>
    var socket_messages = io.connect('http://localhost:3000/messages');
    var socket_infos = io.connect('http://localhost:3000/infos');

    // émission sous /messages
    socket_messages.emit("message", "Bonjour !");

    // réception sous /infos
    socket_infos.on("info", function(data) {
        alert(data);
    });
</script>
</head>

<body>
</body>

</html>
```

Nous avons ici défini deux espaces de noms (`messages` et `infos`), indiqués dans les URL utilisées pour se connecter au serveur. La méthode `io.connect(url)` retourne deux web sockets différentes selon l'URL de connexion utilisée dans chaque cas.

La variable `socket_messages` est utilisée pour envoyer un événement "`message`" au serveur s'il est à l'écoute de cet événement pour cette socket.

La variable `socket_infos` est quant à elle utilisée pour récupérer les événements `info` reçus du serveur.

Programme côté serveur

Le traitement côté serveur va consister à traiter toutes les demandes de connexion établies selon l'URL utilisée lors de `io.connect(url)` par le client.

Fichier server.js

```
var http = require("http");
var fs = require('fs');

var server = http.createServer(function(request, response) {
  var index = fs.createReadStream("index.html");
  index.pipe(response);
});

server.listen(3000);
var io = require("socket.io").listen(server);

// émission et réception sous /messages
io.of("/messages")
.on("connection", function (socket) {
  socket.on("message", function(data) {
    console.log("Réception du message : " + data);
    console.log("provenant des utilisateurs connectés sous /messages");
  });
});

// émission et réception sous /infos
io.of("/infos")
.on("connection", function (socket) {
  console.log("Connexion sous /infos");
  socket.emit("info", "Bienvenue dans la partie infos du serveur");
});
```

La méthode `io.of(namespace)` permet de scinder le programme du serveur en traitant les événements émis et reçus en fonction de l'espace de noms auquel ils sont rattachés.

Tableau 11-8 Méthode `io.of(namespace)`

Méthode	Signification
<code>io.of(namespace)</code>	Spécifie un espace de noms dans lequel on pourra traiter les événements reçus et également en transmettre au client.

DEUXIÈME PARTIE

Construire des applications web avec le framework Express

Introduction au module Connect

Le module Connect a été développé par la société Sencha, qui travaille également sur le framework pour mobiles, nommé Sencha Touch.

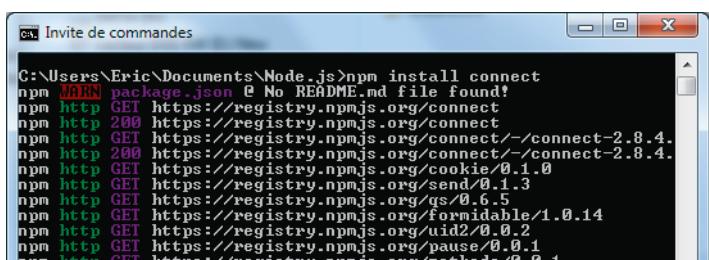
Le module Connect est un ensemble de middlewares, c'est-à-dire un ensemble de méthodes utilitaires facilitant le développement d'applications Node basées sur HTTP.

Dans ce chapitre, nous verrons les principales fonctionnalités offertes par le module Connect, ainsi que la possibilité d'en ajouter de nouvelles.

Installer le module Connect

L'installation du module Connect s'effectue à l'aide de l'utilitaire `npm`. Il suffit donc de taper la commande `npm install connect` dans la fenêtre d'un interpréteur de commandes pour que le module soit téléchargé et installé dans le répertoire `node_modules` du répertoire courant.

Figure 12–1
Installation du module Connect



```
C:\Users\Eric\Documents\Node.js>npm install connect
npm WARN package.json @ No README.md file found!
npm http GET https://registry.npmjs.org/connect
npm http 200 https://registry.npmjs.org/connect
npm http GET https://registry.npmjs.org/connect/-/connect-2.8.4
npm http 200 https://registry.npmjs.org/connect/-/connect-2.8.4
npm http GET https://registry.npmjs.org/cookie/0.1.0
npm http GET https://registry.npmjs.org/send/0.1.3
npm http GET https://registry.npmjs.org qs/0.6.5
npm http GET https://registry.npmjs.org/formidable/1.0.14
npm http GET https://registry.npmjs.org/uid2/0.0.2
npm http GET https://registry.npmjs.org/pause/0.0.1
npm http GET https://registry.npmjs.org/methods/0.0.1
```

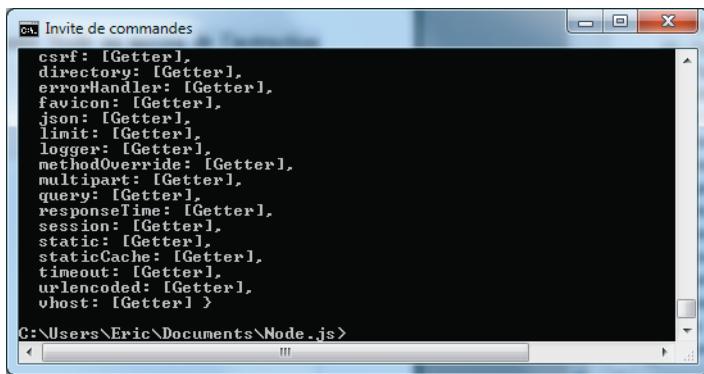
Une fois installé, le module est accessible dans un programme Node au moyen de l'instruction `require("connect")`. Par exemple, dans le fichier `test.js` :

Fichier test.js

```
var connect = require("connect");
console.log(connect);
```

Figure 12-2

Contenu du module Connect



Nous avons ici listé quelques-uns des middlewares qui composent le module Connect. Mais ce module est aussi composé d'une fonction principale appelée directement par l'instruction `connect()`. Cette fonction `connect()` correspond à la méthode `createServer()`, héritée du module `http` de Node et améliorée par le module Connect.

Pour créer un serveur HTTP en utilisant le module Connect, on pourra donc écrire :

Utiliser la méthode `createServer()` du module Connect

```
var connect = require("connect");
var app = connect.createServer(function(req, res) {
  res.end("Bonjour");
});
app.listen(3000);
console.log("Serveur HTTP démarré sur le port 3000");
```

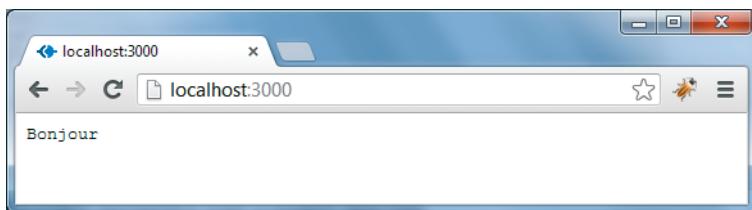
Il est de tradition, lors de l'utilisation du module Connect (et aussi du framework Express qui sera étudié plus loin), d'indiquer le serveur à l'aide de la variable de nom `app` (au lieu de `server` dans les chapitres précédents).

Il en va de même pour les paramètres `request` et `response`, notés ici `req` et `res`, et qui correspondent aux paramètres `request` et `response` vus précédemment.

Une fois le programme lancé, le serveur se met en attente sur le port 3000. Ouvrons un navigateur dans lequel on affiche la page `http://localhost:3000`.

Figure 12–3

Utilisation de Connect pour créer un serveur HTTP



La fonction de callback indiquée en paramètre de `app.createServer()` s'est exécutée pour afficher la réponse dans le navigateur.

Créer un serveur HTTP sans utiliser Connect

Pour le moment, le module Connect n'apporte rien de plus que le serveur HTTP fourni avec Node... Pour voir ce qu'il apporte de plus, il suffit de se demander comment on pourrait faire pour exécuter non pas une seule fonction de callback (indiquée en paramètre de la méthode `createServer()`) mais plusieurs.

Pour cela, utilisons simplement le module `http` de Node, sans utiliser le module Connect. On va créer deux fonctions de callback, appelées successivement lors de chaque requête HTTP sur le serveur :

- la première sera chargée de modifier l'en-tête HTTP en y ajoutant l'en-tête "X-nom" ayant la valeur "Eric" ;
- la seconde enverra la réponse au navigateur, ici la chaîne "Bonjour".

Remarquez que la seconde fonction de callback, qui retourne la réponse du serveur, doit être obligatoirement exécutée en dernier, sinon l'en-tête ne peut pas être modifié une fois la réponse envoyée.

Un serveur HTTP qui exécute deux fonctions de callback

```
var http = require("http");
var server = http.createServer();

server.on("request", function(req, res) {
  res.setHeader("X-nom", "Eric");
});

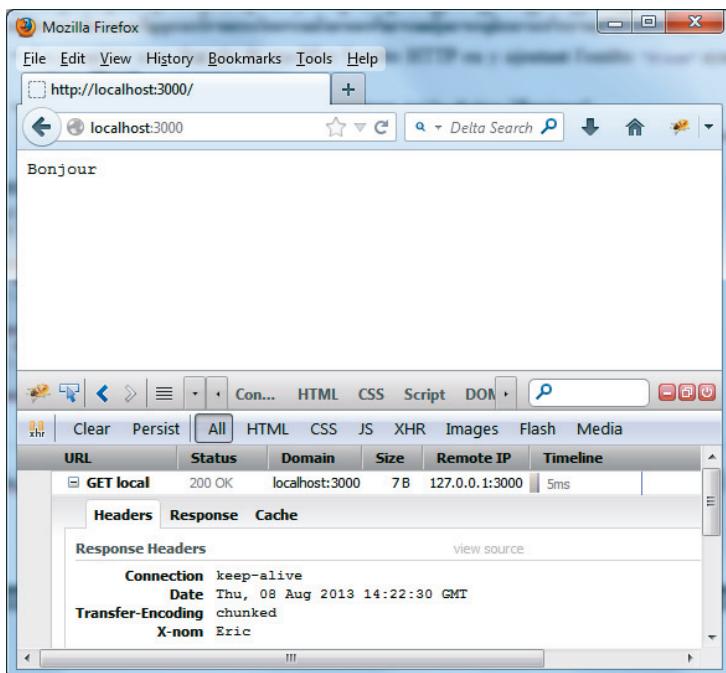
server.on("request", function(req, res) {
  res.end("Bonjour");
});

server.listen(3000);
```

Nous avons simplement utilisé l'événement `request` sur lequel nous avons mis en place deux fonctions de callback.

Figure 12-4

Modification des en-têtes et envoi de la réponse au client



Nous pouvons voir que l'en-tête "X-nom" a été transmis et que l'affichage dans le navigateur a bien eu lieu. Si l'ordre des événements `request` est inversé, on obtient l'affichage de la réponse mais pas la transmission de l'en-tête.

Créer un serveur HTTP en utilisant Connect

On peut écrire le programme de deux façons en utilisant le module Connect. Commençons par la plus intuitive.

Écrire les callbacks en paramètres de la méthode `connect.createServer()`

Cette première façon d'écrire l'enchaînement des fonctions de callback permet de les indiquer les unes à la suite des autres en tant que paramètres de la méthode `connect.createServer()`.

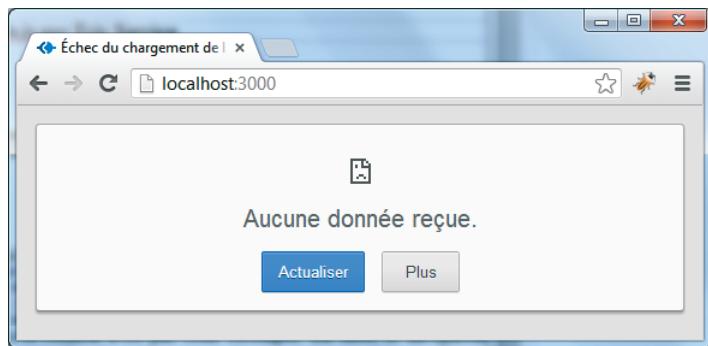
Enchaîner les callbacks dans la méthode `createServer()`

```
var connect = require("connect");
var app = connect.createServer(function(req, res) {
  console.log("Appel callback 1");
  res.setHeader("X-nom", "Eric");
}, function(req, res) {
  console.log("Appel callback 2");
  res.end("Bonjour");
});

app.listen(3000);
```

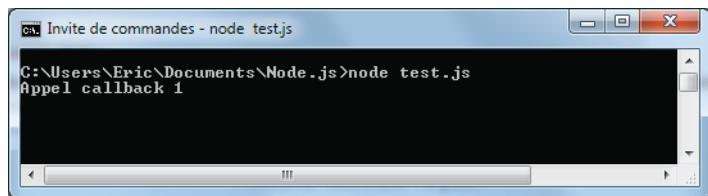
La méthode `connect.createServer()` peut prendre de 0 à n fonctions de callback en paramètres. Toutefois, le résultat n'est pas celui escompté.

Figure 12–5
Mauvaise utilisation
du module Connect



En effet, le navigateur se bloque et n'affiche pas la réponse attendue. La première fonction de callback est exécutée, mais pas la suivante. On peut voir ceci dans la console du serveur, à l'aide des messages affichés dans chacune des fonctions de callback.

Figure 12–6
Le serveur n'envoie
pas la réponse au client.



La première fonction de callback est bien exécutée, mais pas la seconde. L'un des principes de fonctionnement du module Connect veut que nous écrivions nous-mêmes l'enchaînement des fonctions de callback. Ainsi, on ajoute un paramètre dans chacune des fonctions de callback, traditionnellement appelé `next`, qui correspond en fait à la prochaine fonction de callback à appeler.

Voici comment devrait donc s'écrire le code précédent afin qu'il fonctionne selon les principes du module Connect.

Utiliser le paramètre next dans les fonctions de callback

```
var connect = require("connect");
var app = connect.createServer(function(req, res, next) {
  console.log("Appel callback 1");
  res.setHeader("X-nom", "Eric");
  next();
}, function(req, res) {
  console.log("Appel callback 2");
  res.end("Bonjour");
});

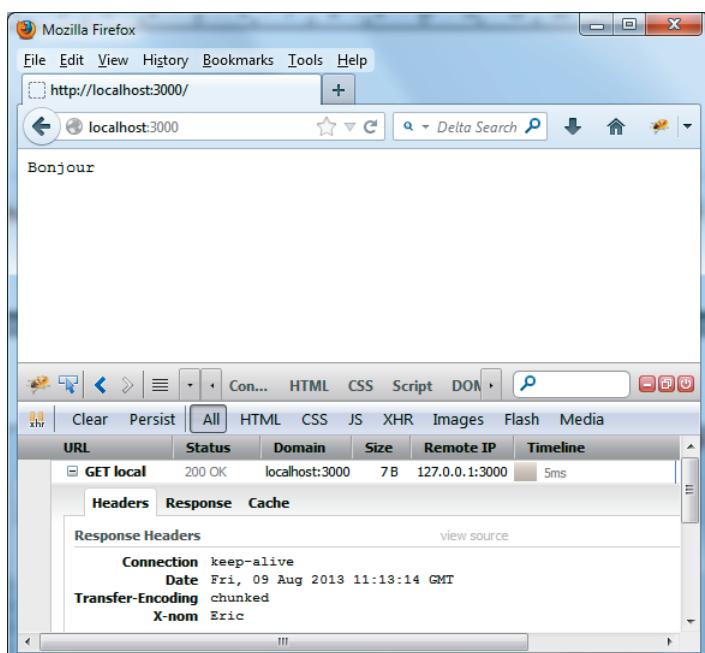
app.listen(3000);
```

Nous avons indiqué le paramètre `next` en dernier paramètre de la première fonction de callback, et nous avons appelé la méthode `next()` afin d'enchaîner avec l'appel de la fonction de callback suivante.

On obtient maintenant l'affichage de la page demandée, ainsi que la transmission de l'en-tête effectuée dans la première fonction de callback.

Figure 12-7

La réponse est envoyée au client grâce à `next()`.



Remarquons que la fonction de callback qui effectue l'envoi de la réponse par `res.end()` est obligatoirement la dernière fonction de callback à être appelée. Si vous enchaînez à la suite vers une nouvelle fonction de callback dans la liste, une erreur se produit.

Utiliser la méthode `app.use()`

La précédente façon d'écrire les fonctions de callback est certes assez intuitive, mais elle n'est pas très pratique à l'usage lorsque plusieurs fonctions de callback doivent se trouver dans la liste. Le module Connect a donc implémenté une autre façon afin de faciliter l'écriture, à savoir en utilisant la méthode `app.use()`.

Utiliser `app.use()` pour enchaîner les fonctions de callback

```
var connect = require("connect");
var app = connect.createServer();
app.use(function(req, res, next) {
  console.log("Appel callback 1");
  res.setHeader("X-nom", "Eric");
  next();
});
app.use(function(req, res) {
  console.log("Appel callback 2");
  res.end("Bonjour");
});
app.listen(3000);
```

La variable `app` est retournée par la méthode `connect.createServer()`, utilisée ici sans argument. Les fonctions de callback sont transmises par les différents appels effectués à la méthode `app.use(callback)`.

Remarquez que chaque fonction de callback appelle la fonction `next()` indiquée en paramètre, sauf la dernière fonction de callback qui est celle retournant le résultat à afficher au navigateur. Chacune de ces fonctions de callback, appelant ou non la fonction `next()`, est appelée un *middleware*. Celui-ci correspond donc à une fonction de traitement utilisant les paramètres `request` et `response` (ici, notés `req` et `res`), et pouvant éventuellement les modifier. Ce middleware pourra aussi être chaîné avec un autre, grâce au paramètre `next` qui permettra au module Connect de les enchaîner.

Si la méthode `connect.createServer()` est appelée sans argument (comme précédemment), on peut la remplacer par la fonction `connect()`, qui correspond au nom de la variable `connect` récupérée pour le module. Ceci est possible car le module Connect comporte une fonction principale qui peut être appelée de cette manière

(voir la section « Cas particulier : un module composé d'une fonction principale » du chapitre 2, « Gestion des modules »).

Créer l'objet app à partir de la fonction connect()

```
var app = connect(); // identique à var app = connect.createServer();
```

Le reste du programme est identique. Vous trouverez beaucoup plus souvent l'appel à la méthode `connect()`, que celui utilisant `connect.createServer()`.

Utiliser l'objet app en tant que fonction de callback

Une autre forme d'écriture que vous pourrez rencontrer est celle qui consiste à considérer l'objet `app` comme une fonction de callback, qui pourra alors s'utiliser comme argument de la méthode `http.createServer()` (définie dans le module `http` de Node).

Le précédent programme peut alors également s'écrire :

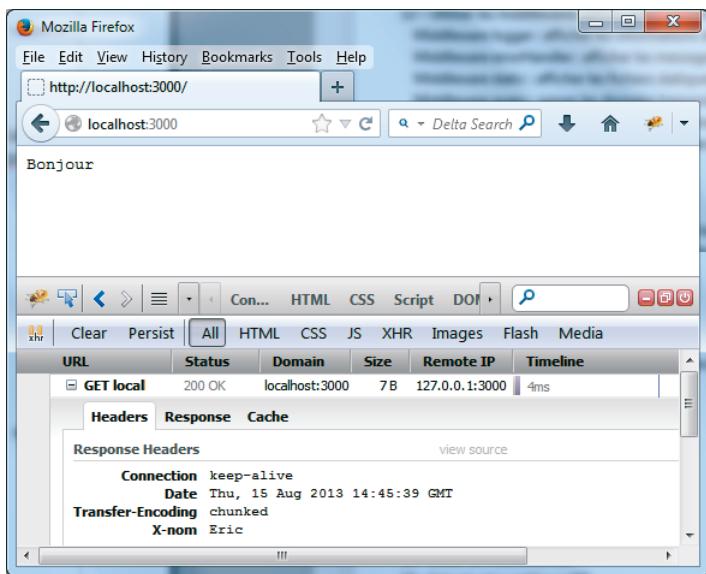
Utiliser http.createServer()

```
var connect = require("connect");
var http = require("http");
var app = connect.createServer();
app.use(function(req, res, next) {
    console.log("Appel callback 1");
    res.setHeader("X-nom", "Eric");
    next();
});
app.use(function(req, res) {
    console.log("Appel callback 2");
    res.end("Bonjour");
});

http.createServer(app).listen(3000);
```

Nous avons simplement transmis l'objet `app` en tant que paramètre de la méthode `http.createServer()`. On peut vérifier que le résultat est identique au précédent.

Figure 12–8
Utilisation de app.use()



Le résultat s'affiche dans la page et l'en-tête "X-nom" est transmis au navigateur.

Définir et utiliser un middleware

Un cas très fréquent est celui dans lequel vous utilisez des middlewares externes définis dans d'autres modules. Nous montrons ici comment définir un middleware **logger** permettant d'afficher dans la console du serveur les requêtes effectuées au serveur, et également comment les insérer dans un fichier sur disque pour conserver une trace de l'activité.

Dans les trois étapes qui suivent, vous verrez comment arriver à une solution optimale pour la conception de ce middleware.

Créer le middleware dans le programme de l'application

La première étape consiste à créer le middleware dans le code de notre application, comme nous l'avons fait dans les exemples précédents.

Définir et utiliser le middleware logger

```
var connect = require("connect");
var fs = require("fs");
```

```

var app = connect();

app.use(function(req, res, next) {
    var filename = "logs.txt";
    fs.writeFile(filename, req.url + "\r\n", { flag : "a+" });
    console.log(filename + " " + req.url);
    next();
});

app.use(function(req, res, next) {
    res.setHeader("X-nom", "Eric");
    next();
});

app.use(function(req, res) {
    res.end("Bonjour");
});

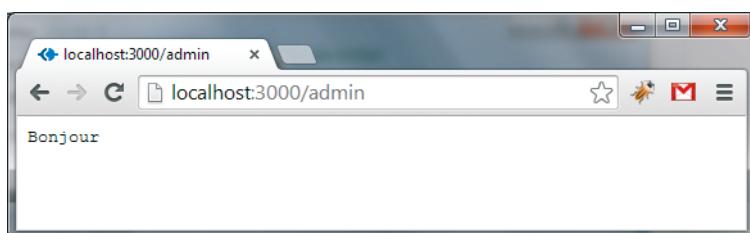
app.listen(3000);

```

Le middleware `logger` est défini au moyen de la méthode `app.use()`. Il écrit dans le fichier `logs.txt` l'URL de la requête et affiche dans la console du serveur les requêtes effectuées.

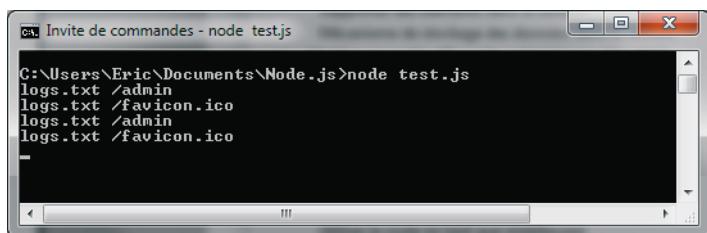
Pour tester ce middleware, il suffit de lancer le serveur, puis de se connecter sur celui-ci en introduisant des URL commençant par `http://localhost:3000`.

Figure 12-9
Utilisation de l'URL /admin pour activer le middleware logger



En rafraîchissant plusieurs fois la fenêtre précédente, la console du serveur affiche les URL utilisées lors des requêtes au serveur.

Figure 12-10
Les URL activées par le serveur sont affichées par le middleware logger dans la console.

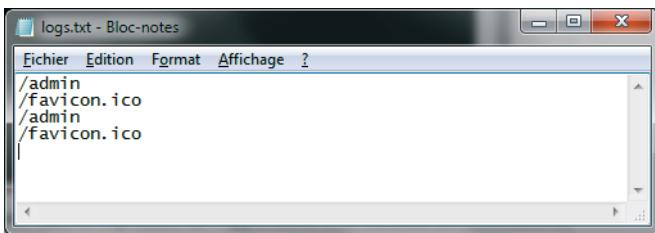


Les URL `/admin` et `/favicon.ico` sont demandées à chaque rafraîchissement de `http://localhost:3000/admin`.

Quant au fichier `logs.txt`, il est créé dans le répertoire de l'application et il contient les informations suivantes.

Figure 12-11

Contenu du fichier `logs.txt`
produit par le middleware
logger



Créer le middleware dans un fichier externe

La deuxième étape consiste à utiliser ce middleware à partir d'un fichier externe, afin qu'il puisse être réutilisé par d'autres applications. On crée le fichier `logger.js` qui contient le code source du middleware (défini en tant que module externe).

Fichier `logger.js`

```
var fs = require("fs");
function logger(req, res, next) {
    var filename = "logs.txt";
    fs.writeFile(filename, req.url + "\r\n", { flag : "a+" });
    console.log(filename + " " + req.url);
    next();
}

module.exports = logger;
```

Le middleware `logger()` est exporté en tant que fonction principale du module `logger.js`.

Utiliser le middleware logger

```
var connect = require("connect");
var logger = require("./logger.js");
var app = connect();

app.use(logger);
app.use(function(req, res, next) {
    res.setHeader("X-nom", "Eric");
    next();
});
app.use(function(req, res) {
    res.end("Bonjour");
```

```
});  
app.listen(3000);
```

L'utilisation du middleware `logger` s'effectue toujours au moyen de la méthode `app.use()`, en lui transmettant la référence vers la méthode `logger()` (elle-même associée au module `logger` en tant que fonction principale). D'où l'écriture sous la forme `app.use(logger)` comme effectuée ici.

Transmettre des paramètres au middleware

Le middleware `logger` précédent utilise toujours le même fichier `logs.txt` dans lequel il écrit les informations. On peut améliorer ce middleware en transmettant en paramètre le nom du fichier dans lequel on souhaite écrire les informations. On modifie donc le module `logger.js` en définissant le middleware `logger(filename)` qui sert à écrire dans un fichier (indiqué en paramètre) l'URL qui est utilisée dans la requête.

Le fichier `logger.js` définissant le middleware pourrait être le suivant.

Fichier logger.js

```
var fs = require("fs");  
function logger(filename) {  
    return function(req, res, next) {  
        fs.writeFile(filename, req.url + "\r\n", { flag : "a+" });  
        console.log(filename + " " + req.url);  
        next();  
    };  
}  
  
module.exports = logger;
```

Nous définissons la fonction `logger(filename)` qui retourne une fonction (fonction de callback) ayant comme paramètres `req`, `res` et `next`. La méthode `logger()` est exportée en tant que fonction principale du module `logger.js`.

Remarquons qu'on est obligé de passer par une fonction intermédiaire `logger()` qui retourne la fonction de callback, car la fonction `logger()` possède un paramètre `filename` que la fonction de callback ne peut pas intégrer dans sa liste de paramètres (elle est limitée à `req`, `res` et `next`).

Le module utilisant ce middleware pourrait alors s'écrire :

Utiliser le middleware logger()

```
var connect = require("connect");  
var logger = require("./logger.js");  
var app = connect();
```

```
app.use(logger("logs.txt"));
app.use(function(req, res, next) {
  res.setHeader("X-nom", "Eric");
  next();
});
app.use(function(req, res) {
  res.end("Bonjour");
});

app.listen(3000);
```

La mise en place du middleware `logger()` se fait par l'appel à `app.use(logger("logs.txt"))`. L'exécution de `logger("logs.txt")` retourne une fonction correspondant à la fonction de callback qui est donc indiquée en argument de `app.use(callback)`.

Après affichage de quelques URL dans le navigateur, elles apparaissent sur la console du serveur mais s'inscrivent également dans le fichier `logs.txt` sur le serveur. Bien sûr, le middleware positionnant l'en-tête "`X-nom`" est également exécuté.

Chaînage des méthodes dans Connect

Le framework Connect a été construit afin de permettre le chaînage des méthodes sur l'objet `app`. Le précédent programme peut donc également être écrit :

Chainer les méthodes sur l'objet app

```
var connect = require("connect");
var logger = require("./logger.js");
var app = connect();

app.use(logger("logs.txt"))
.use(function(req, res, next) {
  res.setHeader("X-nom", "Eric");
  next();
})
.use(function(req, res) {
  res.end("Bonjour");
})
.listen(3000);
```

Les méthodes `use()` et `listen()`, définies sur l'objet `app` par le module Connect, sont chaînées à la suite car elles retournent l'objet `app` lui-même.

Cas d'erreurs dans les middlewares

Un middleware peut provoquer une erreur qui empêche les middlewares suivants de se poursuivre. Dans ce cas, il suffit de créer un middleware spécial qui sera chargé de récupérer les éventuelles erreurs provoquées par les autres middlewares.

On crée donc le middleware `errorHandler` chargé de récupérer les erreurs et de les afficher sur la page du navigateur. Ce middleware sera inscrit dans le module `errohandler.js`.

Définition du middleware `errorHandler`

```
function errorHandler() {
  return function(err, req, res, next) {
    console.log("errorHandler");
    if (err) {
      res.writeHead(500, {"Content-Type": "text/html"});
      res.end("<h1>Erreur !</h1>\n<pre>" + err.stack + "</pre>");
    } else next();
  }
}

module.exports = errorHandler;
```

Un middleware gérant les erreurs possède un premier paramètre qui est l'objet `err` correspondant à l'erreur. Les paramètres traditionnels `req`, `res` et `next` suivent dans la liste des paramètres.

Le paramètre `err` n'est positionné que si une erreur est apparue. Dans ce cas, on décide ici d'afficher une page contenant le message d'erreur en utilisant `err.stack`. Sinon, on passe à la fonction de callback suivante à l'aide de `next()`.

Utilisation du middleware `errorHandler`

```
var connect = require("connect");
var logger = require("./logger.js");
var errorHandler = require("./errorhandler.js");
var app = connect();

app.use(logger("logs.txt"))
.use(function(req, res, next) {
  next(new Error("Il y a eu une erreur !")); // provoquer une erreur
})
.use(function(req, res, next) {
  res.setHeader("X-nom", "Eric");
```

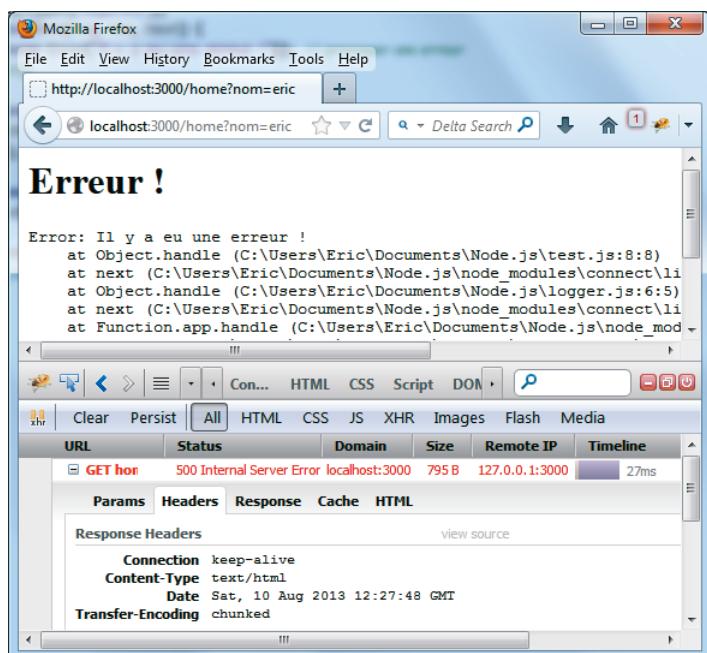
```
    next();
})
.use(function(req, res) {
  res.end("Bonjour");
})
.use(errorHandler()) // récupérer l'erreur
.listen(3000);
```

On a créé un middleware provoquant volontairement une erreur. Cette dernière est transmise au middleware suivant au moyen de `next(err)`.

Une fois l'erreur provoquée, elle est directement récupérée par le middleware qui la traite, à savoir `errorHandler`. Lorsque ce dernier reçoit l'erreur, il affiche le message d'erreur dans la page. Si le middleware `errorHandler` est appelé sans erreur, il se contente d'effectuer `next()` ce qui permet de passer à la fonction de callback suivante.

Figure 12-12

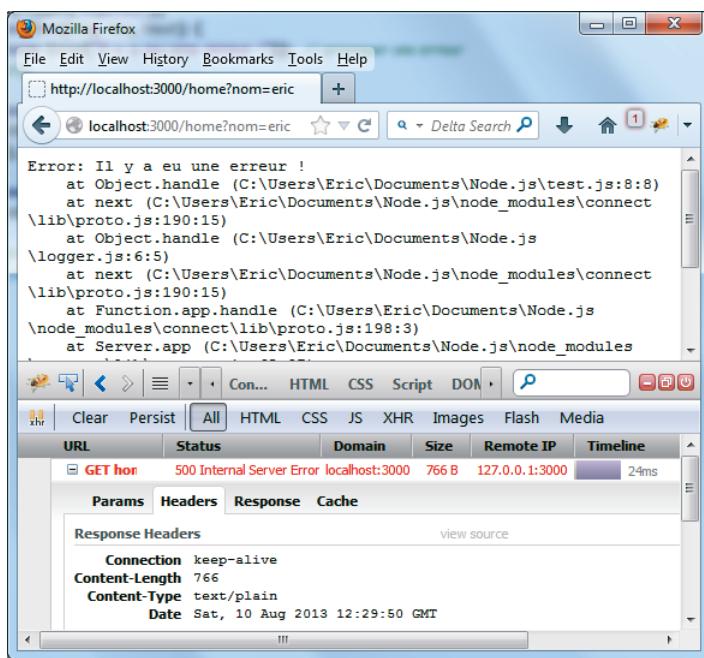
Erreur dans un middleware, gérée par le middleware `errorHandler`



En supposant que l'on ne gère pas l'erreur au moyen du middleware `errorHandler`, l'erreur est tout de même récupérée par le module Connect qui l'affiche dans la page, mais de façon moins stylée.

Figure 12–13

Erreur dans un middleware,
non gérée par le middleware
errorHandler



13

Utiliser les middlewares définis dans Connect

Comme nous venons de le voir, le module Connect permet de définir nos propres middlewares. Mais il propose également un ensemble de middlewares prêts à l'emploi.

Le but de ces middlewares est de donner facilement accès à des fonctionnalités traditionnelles de serveurs HTTP, par exemple :

- l'affichage d'informations dans les logs (sur l'écran du serveur ou dans des fichiers) ;
- la gestion des cookies et des sessions ;
- l'affichage amélioré des messages d'erreurs ;
- l'affichage aisément des fichiers statiques (images, PDF, etc.) ;
- la gestion facile des informations transmises dans la requête HTTP (requête ou en-tête) ;
- etc.

Dans ce chapitre, nous allons étudier les principaux middlewares du module Connect.

Middleware logger : afficher les informations dans les logs

Le middleware `logger` permet d'afficher des informations dans des fichiers de logs ou sur la console du serveur. Des informations types peuvent être affichées, mais on peut également indiquer une par une les informations que l'on souhaite voir à l'écran. Le middleware `Logger` est accessible via la méthode `connect.logger()`.

Tableau 13-1 Méthodes `connect.logger(format)`

Méthode	Signification
<code>connect.logger(format)</code>	Affiche les informations selon le format indiqué (<code>string</code>). La chaîne <code>format</code> peut contenir des valeurs prédéfinies (liste ci-dessous) mais aussi n'importe quelle chaîne de caractères. Les informations prédéfinies sont : <ul style="list-style-type: none"> - <code>:req[header]</code> (par exemple <code>:req[Accept]</code>) - <code>:res[header]</code> (par exemple <code>:res[Content-Length]</code>) - <code>:http-version</code> - <code>:response-time</code> - <code>:remote-addr</code> - <code>:date</code> - <code>:method</code> - <code>:url</code> - <code>:referrer</code> - <code>:user-agent</code> - <code>:status</code> Ainsi, pour afficher le type de la requête, l'URL et le status, on écrira la chaîne <code>format ":method :url - Status :status"</code> , ce qui affichera par exemple "GET/admin-Status 200".
<code>connect.logger("dev")</code>	Affiche les informations utiles en mode développement. Cela correspond à la chaîne " <code>:format :method :url :status :response-time ms</code> ". De plus, la valeur <code>:status</code> est indiquée en couleur pour visualiser une éventuelle erreur.
<code>connect.logger("tiny")</code> <code>connect.logger("short")</code> <code>connect.logger("default")</code>	Chacun de ces trois modes (" <code>tiny</code> ", " <code>short</code> " et " <code>default</code> ") affiche plus d'informations que le précédent.

La forme de la méthode `connect.logger(format)` précédente prend une chaîne de caractères en paramètre, qui indique la forme des informations à afficher. Une autre forme de la méthode existe, qui prend un objet `options` en paramètre. On peut alors indiquer un fichier dans lequel écrire (plutôt que sur la console du serveur).

Tableau 13–2 Méthode connect.logger(options)

Méthode	Signification
<code>connect.logger(options)</code>	L'objet <code>options</code> possède les propriétés suivantes : - <code>options.stream</code> : stream en écriture dans lequel les affichages seront effectués (à la place de la console du serveur). Le stream est créé au moyen de <code>fs.createWriteStream()</code> du module <code>fs</code> . Par défaut, <code>options.stream</code> vaut <code>process.stdout</code> , ce qui produit les affichages sur la console. - <code>options.buffer</code> : délai entre deux écritures dans le fichier, en ms. Il est obligatoire de positionner cette option en plus de <code>options.stream</code> . Indiquer <code>true</code> pour utiliser la valeur par défaut (1 000 ms). - <code>options.format</code> : format d'écriture des données. Correspond au paramètre <code>format</code> précédent.

Voici quelques exemples d'utilisation du middleware `logger`.

Utiliser `connect.logger("dev")`

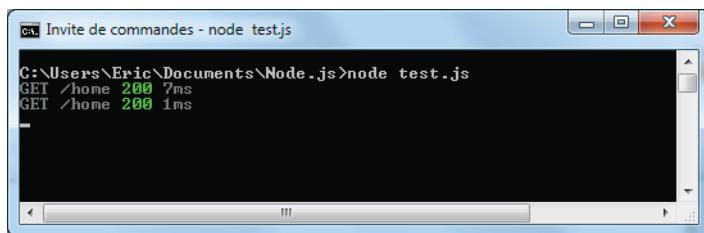
```
var connect = require("connect");
var app = connect();

app.use(connect.logger("dev"))
.use(function(req, res) {
  res.end("Bonjour");
})
.listen(3000);
```

En supposant qu'on se connecte sur le serveur au moyen d'un navigateur en tapant l'URL `http://localhost:3000/home`, la console du serveur affiche :

Figure 13–1

Utilisation du middleware logger (mode dev)



Comme indiqué précédemment, nous avons bien :

- le type de requête (ici, `GET` car l'URL a été introduite depuis la barre d'adresses du navigateur) ;
- l'URL demandée (ici, `/home`) ;

- le status (ici, 200, en vert pour montrer que l'URL a été traitée) ;
- le délai de réponse en millisecondes (ms).

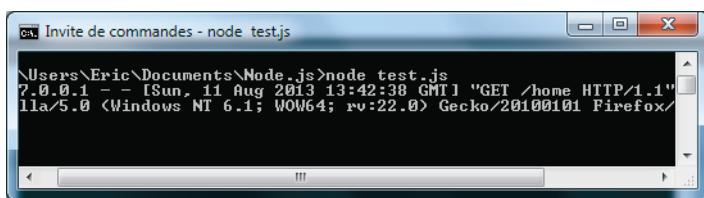
Plus d'informations peuvent être obtenues en utilisant le format "default".

Utiliser le format "default" pour le middleware logger

```
var connect = require("connect");
var app = connect();

app.use(connect.logger("default"))
.use(function(req, res) {
  res.end("Bonjour");
})
.listen(3000);
```

Figure 13-2
Utilisation du middleware
logger (mode default)



Enfin, supposons que l'on souhaite afficher les informations dans un fichier de logs, ici `logs.txt`. On écrira alors :

Utiliser un fichier de logs

```
var connect = require("connect");
var fs = require("fs");
var app = connect();

app.use(connect.logger( {
  stream : fs.createWriteStream("logs.txt"),
  format : "short",
  buffer : true
}))
.use(function(req, res) {
  res.end("Bonjour");
})
.listen(3000);
```

On peut aussi afficher les informations dans un fichier et à l'écran. Pour cela, il suffit d'utiliser deux fois le middleware `logger`, l'ordre d'écriture est important : d'abord l'écriture dans le fichier, puis l'écriture à l'écran.

Afficher dans les logs et à l'écran

```
var connect = require("connect");
var fs = require("fs");
var app = connect();

app
  .use(connect.logger( {           // affichage dans un fichier
    stream : fs.createWriteStream("logs.txt"),
    format : "short",
    buffer : true
  }))
  .use(connect.logger("dev")) // affichage à l'écran
  .use(function(req, res) {
    res.end("Bonjour");
})
.listen(3000);
```

Middleware `errorHandler` : afficher les messages d'erreur

Le middleware `errorHandler` va permettre d'afficher sous une forme plus appropriée les messages d'erreur se produisant lors de l'exécution. Le middleware `errorHandler` est accessible via la méthode `connect.errorHandler()`. Il est similaire au middleware que nous avions écrit dans le précédent chapitre, mais traite plus de cas (en particulier les réponses au format JSON ou en texte non HTML).

Utiliser le middleware `errorHandler`

```
var connect = require("connect");
var app = connect();

app
  .use(connect.logger("dev"))
  .use(function(req, res, next) {
    next(new Error("Ceci est une erreur !"));
})
  .use(connect.errorHandler())
  .use(function(req, res) {
    res.end("Bonjour");
})
.listen(3000);
```

Comme précédemment, nous avons provoqué volontairement une erreur afin de pouvoir la récupérer par le middleware `errorHandler`.

Figure 13–3
Utilisation du middleware
errorHandler

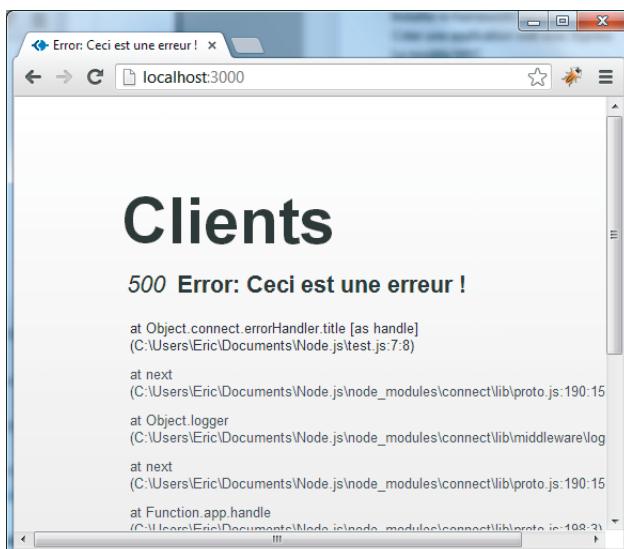


Le titre de l'application est par défaut "Connect", mais on peut le changer par la ligne suivante.

Changer le nom de l'application affiché dans le message d'erreur

```
connect.errorHandler.title = "Clients";
```

Figure 13–4
Paramétrage du middleware
errorHandler



Le nom de l'application affiché est maintenant "Clients". Cette ligne peut être insérée dès que le module Connect a été chargé (c'est-à-dire après l'instruction `require("connect")`).

Middleware static : afficher les fichiers statiques

Dans le chapitre 10 consacré à la gestion des connexions HTTP, nous avions vu dans la section « Utiliser plusieurs fichiers statiques dans la réponse » comment afficher des fichiers statiques dans le navigateur. Le module Connect possède le middleware `static` qui effectue ce traitement. Toutes les URL rattachées à des fichiers statiques seront ainsi directement affichées dans le navigateur sans qu'il soit nécessaire d'écrire le code associé.

Le middleware `static` est associé à la méthode `connect.static(root)` dans laquelle `root` est le répertoire du serveur où sont stockés les fichiers statiques. Le middleware `static` peut être utilisé plusieurs fois pour indiquer différents emplacements dans lesquels pourront se trouver les fichiers statiques de l'application.

Les sous-répertoires ne sont pas concernés lors de la recherche des fichiers par le module Connect, mais ils peuvent être indiqués directement dans l'URL qui accède au serveur.

Dans l'exemple qui suit, nous créons le répertoire `public` qui contient les sous-répertoires `images` et `pdf`. Les fichiers se trouvant dans ces deux répertoires pourront être visualisés directement, sans aucun traitement de notre part.

Utiliser le middleware static pour afficher les images et les PDF

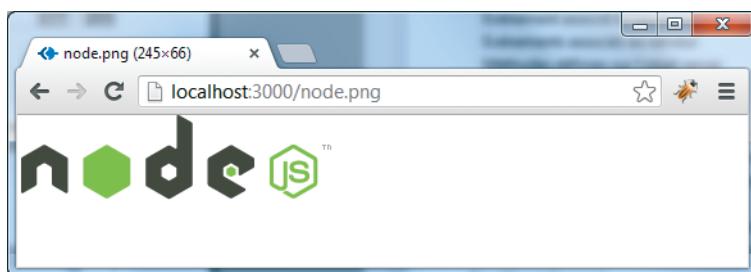
```
var connect = require("connect");
var app = connect();

app
  .use(connect.logger("dev"))
  .use(connect.static("public/images"))
  .use(connect.static("public/pdf"))
  .use(function(req, res) {
    res.end("Bonjour");
})
.listen(3000);
```

Déposons le fichier `node.png` dans le répertoire `public/images` et affichons l'URL `http://localhost/node.png` dans le navigateur.

Figure 13–5

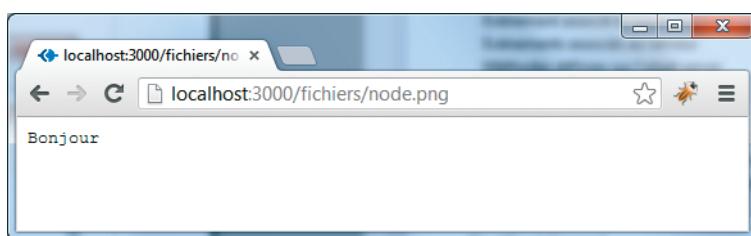
Utilisation du middleware static pour afficher une image



Il est à noter que si l'URL n'est pas `node.png` (par exemple, [fichiers/node.png](#)), le sous-répertoire `fichiers` n'existe pas et donc l'image ne s'affiche pas. Par exemple, si on entre l'URL `http://localhost:3000/fichiers/node.png`, ce n'est pas l'image qui s'affiche mais "Bonjour".

Figure 13–6

Affichage d'une URL non associée à un fichier statique



En revanche, si vous créez un sous-répertoire `fichiers` dans `public/images` en y mettant l'image `node.png`, l'URL `http://localhost:3000/fichiers/node.png` sera trouvée et l'image s'affichera.

Middleware query : parser les données transmises dans la requête

Le middleware `query` permet d'ajouter la propriété `query` à l'objet `req` transmis en paramètre. La propriété `req.query` sera un objet contenant les propriétés et valeurs transmises dans l'URL, après l'éventuel `"?"`. Cette propriété sera transmise de middleware en middleware, jusqu'à son utilisation éventuelle pour afficher le résultat dans la page. Si ce middleware n'était pas utilisé, il faudrait parser nous-mêmes l'URL transmise dans `req.url`.

Le middleware `query` est accessible via la méthode `connect.query()`. Voyons son utilisation sur un exemple.

Utiliser le middleware query

```
var connect = require("connect");
var app = connect();

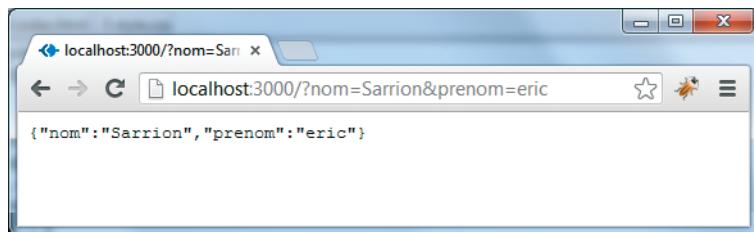
app
  .use(connect.logger("dev"))
  .use(connect.query())
  .use(function(req, res) {
    res.end(JSON.stringify(req.query));
})
.listen(3000);
```

Nous utilisons le middleware `query` au moyen de `connect.query()`, puis nous avons accès à `req.query` qui contient les informations transmises dans l'URL.

Pour tester ce programme, il faut indiquer une URL contenant une `query`, par exemple `http://localhost:3000?nom=Sarrion&prenom=eric`.

Figure 13-7

Contenu de l'objet `req.query` avec le middleware `query`



Si vous supprimez l'utilisation du middleware `query`, l'objet `req.query` n'est pas initialisé.

Middleware `bodyParser` : parser les données transmises dans l'en-tête

Le middleware `query` précédent permet de parser les données transmises directement dans l'URL. Mais il peut arriver que certaines données soient transmises dans l'en-tête, par exemple lors de l'envoi d'un formulaire en `POST`. Le middleware `bodyParser` effectue un travail similaire au middleware `query`, mais pour les données transmises dans l'en-tête. Il initialise l'objet `req.body` avec les données récupérées dans celui-ci.

Le middleware `bodyParser` est accessible via la méthode `connect.bodyParser()`. Pour l'utiliser, nous créons dans cet exemple un formulaire HTML qui contiendra les

champs `nom` et `prenom`, qui seront ensuite transmis (grâce à la méthode `POST`) à la page `index.html` en cas de validation du formulaire.

Remarquons que si la méthode `GET` est utilisée pour transmettre ces données, le middleware `query` devrait alors être utilisé au lieu de `bodyParser`.

Utiliser le middleware `bodyParser`

```
var connect = require("connect");
var app = connect();

app
  .use(connect.logger("dev"))
  .use(connect.static("forms"))
  .use(connect.bodyParser())
  .use(function(req, res) {
    res.end(JSON.stringify(req.body));
})
.listen(3000);
```

Nous utilisons ici le middleware `static` en plus de `bodyParser`. En effet, le formulaire à afficher est un fichier statique se trouvant dans le répertoire `forms` du serveur. Voici le formulaire qui sera affiché en premier.

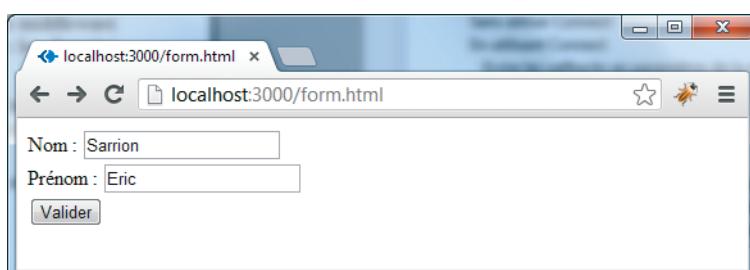
Fichier `form.html` (dans le répertoire `forms` du serveur)

```
<form action=index.html method=post>
  Nom : <input name=nom value=Sarrion /> <br>
  Prénom : <input name=prenom value=Eric /> <br>
  <input type=submit value=Valider />
</form>
```

L'attribut `method` est positionné à "`post`" de façon à activer le middleware `bodyParser`. L'activation du fichier `index.html` lors de la validation du formulaire déclenchera l'envoi de la réponse du serveur par `res.end()`.

Lors de l'affichage du formulaire `form.html`, on obtient :

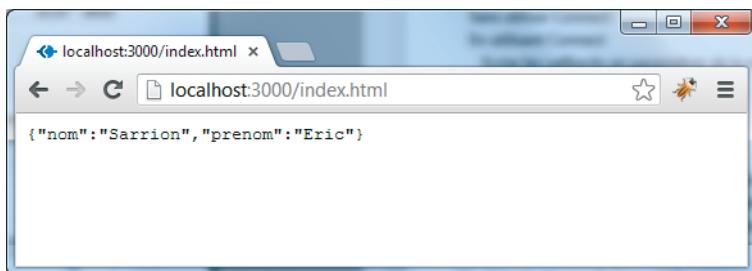
Figure 13–8
Saisie des informations
dans le formulaire



Puis après validation du formulaire, on obtient :

Figure 13–9

Contenu de req.body avec le middleware bodyParser



L'URL `index.html` est devenue active et le résultat se trouvant dans `req.body` est affiché sous forme JSON.

Middleware favicon : gérer l'icône affichée dans la barre d'adresses

Si vous en avez assez de voir l'URL `favicon.ico` apparaître dans vos logs, le middleware `favicon` est fait pour vous. En effet, il permet d'indiquer un fichier image qui servira d'icône à afficher dans la barre d'adresses du navigateur. Ainsi ce fichier ne sera plus constamment demandé au serveur par le navigateur pour chacune des URL qu'il traite.

Le middleware `favicon` est utilisable grâce à la méthode `connect/favicon(fichier)`. Si vous n'indiquez pas de fichier en paramètre, une image par défaut sera affichée par le module Connect.

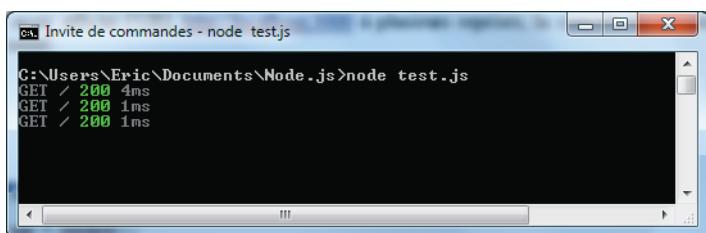
Utiliser le middleware favicon

```
var connect = require("connect");
var app = connect();

app
  .use(connect.logger("dev"))
  .use(connect/favicon())
  .use(function(req, res) {
    res.end("Bonjour");
})
.listen(3000);
```

Après avoir affiché l'URL `http://localhost:3000` à plusieurs reprises, la console du serveur affiche maintenant :

Figure 13-10
Utilisation du middleware
favicon



```
C:\Users\Eric\Documents\Node.js>node test.js
C:\Users\Eric\Documents\Node.js>
GET / 200 4ms
GET / 200 1ms
GET / 200 1ms
```

Il n'y a plus trace du fichier `favicon.ico...`

Middleware session : gérer les sessions

La session concerne des données associées à l'utilisateur qui navigue sur le site, et que l'on désire conserver le temps de sa visite sur le site. Une fois que son navigateur est fermé, les données stockées dans la session sont perdues. Un exemple de données stockées au cours d'une session peut être le panier d'achats d'un utilisateur.

Ce concept est pratique car il permet de stocker de façon temporaire des informations sans avoir à les sauvegarder nous-mêmes dans une base de données qui serait temporaire. Le mécanisme offert par le module Connect le fait à notre place.

Pour fonctionner, les sessions utilisent deux middlewares implémentés par le module Connect.

- Le middleware `cookieParser` qui permet de gérer des cookies (qui seront utilisés de façon interne par la session, même si celle-ci n'est pas sauvegardée dans des cookies). Ce middleware est géré par la méthode `connect.cookieParser(secret)`, dans laquelle `secret` est une chaîne de caractères quelconque qui servira à chiffrer les données stockées en session de façon à ce que personne d'autre ne puisse y accéder.
- Le middleware `session`, qui gère l'objet `req.session` conservant les données de session. Ce middleware est accessible grâce à la méthode `connect.session()`. En plus des données de la session, `req.session` propose des méthodes comme `req.session.destroy()`, qui permet de détruire toutes les données stockées dans la session de l'utilisateur.

Ajouter des éléments dans la session

Montrons un exemple utilisant les sessions. Nous mettons dans la session les informations transmises dans l'URL (partie `query`), puis nous les affichons dans la page HTML au moyen de l'objet `req.session`.

Utiliser les sessions

```
var connect = require("connect");
var app = connect();

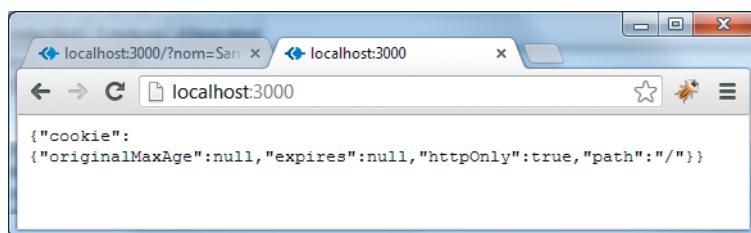
app
  .use(connect.logger("dev"))
  .use(connect.cookieParser("chaine secrete"))
  .use(connect.session())
  .use(connect.query())
  .use(function(req, res) {
    for (var prop in req.query)
      req.session[prop] = req.query[prop];
    res.end(JSON.stringify(req.session));
  })
  .listen(3000);
```

Une fois les middlewares `query`, `cookieParser` et `session` mis en place, on peut récupérer les données transmises dans l'URL au moyen de `req.query`. Chacune de ces données est placée dans `req.session`. L'objet `req.session` est alors affiché dans le résultat retourné au navigateur.

Si vous exécutez ce programme une première fois en tapant l'URL `http://localhost:3000`, sans transmettre de données dans l'URL, les valeurs suivantes sont affichées dans la page.

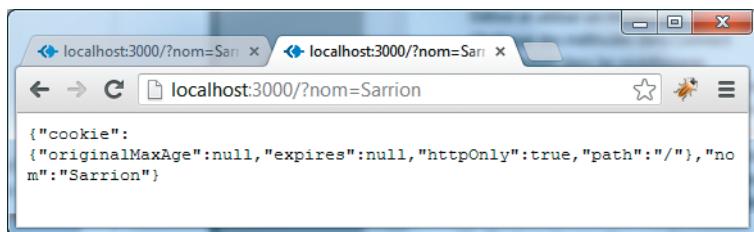
Figure 13-11

Contenu de `req.session` avec le middleware `session`



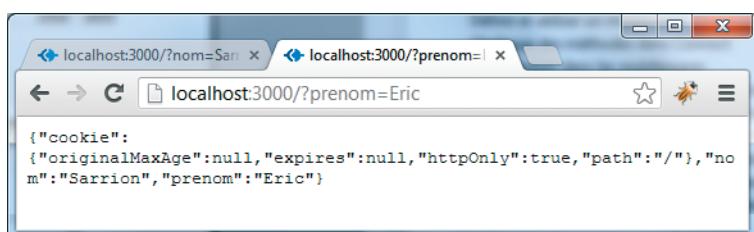
Le middleware `session` utilise une clé `cookie` dans l'objet `req.session` qui lui sert pour ses traitements internes. Si nous utilisons maintenant l'URL `http://localhost:3000?nom=Sarrion`, la session comporte la clé `nom` ayant la valeur "`Sarrion`".

Figure 13–12
Ajout du nom dans la session



Puis, si nous utilisons l'URL `http://localhost:3000?prenom=Eric`, la session comporte en plus la clé `prenom` de valeur "Eric".

Figure 13–13
Ajout du prénom
dans la session



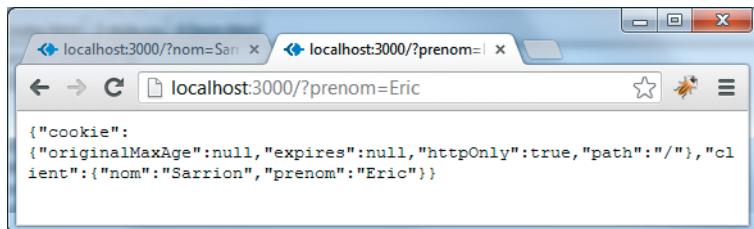
Bien entendu, il est possible de stocker dans la session des objets plus complexes que les chaînes de caractères, par exemple un objet `client` contenant le nom et le prénom.

Stocker un client dans la session

```
var connect = require("connect");
var app = connect();

app
  .use(connect.logger("dev"))
  .use(connect.cookieParser("chaine secrete"))
  .use(connect.session())
  .use(connect.query())
  .use(function(req, res) {
    req.session.client = { nom : "Sarrion", prenom : "Eric" };
    res.end(JSON.stringify(req.session));
})
.listen(3000);
```

Figure 13–14
Ajout d'un client { nom,
prenom } dans la session



Supprimer des éléments dans la session

Il est possible de supprimer tout ou partie des éléments dans la session.

Pour supprimer un élément, on utilise l'opérateur `delete`, en écrivant par exemple `delete req.session.nomElement`.

Supprimer le nom de la session

```
delete req.session.nom;
```

Pour supprimer tous les éléments de la session, on utilise la méthode `req.session.destroy()`.

Supprimer tous les éléments de la session

```
req.session.destroy();
```

Mécanisme de stockage des données dans la session

Pour l'instant, nous n'avons pas expliqué comment étaient stockées les données dans la session. Nous avons simplement constaté que tout fonctionnait normalement, aussi bien pour sauvegarder des informations dans la session que pour les récupérer.

Par défaut, le module Connect sauvegarde les données de session en mémoire (en utilisant `MemoryStore`). Si le serveur s'arrête, les informations sont perdues, ce qui peut être gênant dans certains cas.

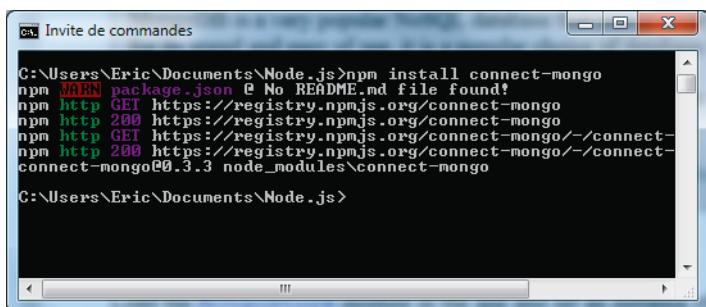
Pour palier ce problème, le module Connect permet de choisir le mode de stockage des données de session. On pourra ainsi choisir un mode plus sécurisé (stockage sur disque) qui permettra en plus de pouvoir partager la session d'un utilisateur entre plusieurs serveurs (*clusters*).

Pour cela, Node propose un module basé sur `MongoDB`, qui est une base de données rapide et facile à utiliser. On peut installer le module `connect-mongo` qui permet de faire le lien entre le code JavaScript et le gestionnaire de sessions associé à `MongoDB`.

Il faut auparavant installer `MongoDB` sur votre serveur (voir le chapitre 19, « Introduction à `MongoDB` »).

Installons `connect-mongo` à l'aide de `npm`. On tape la commande `npm install connect-mongo` dans une fenêtre de l'interpréteur de commandes.

Figure 13-15
Installation du module
connect-mongo



Le programme précédent doit être légèrement modifié pour indiquer à Node que l'on souhaite utiliser MongoStore au lieu de MemoryStore.

Utiliser MongoStore pour gérer les sessions

```
var connect = require("connect");
var app = connect();
var MongoStore = require('connect-mongo')(connect);

app
.use(connect.logger("dev"))
.use(connect.cookieParser("chaine secrete"))
.use(connect.session({
  store : new MongoStore({
    db: "mydb",
    host: "localhost",
    port: 27017
  })
})
.use(connect.query())
.use(function(req, res) {
  req.session.client = { nom : "Sarrion", prenom : "Eric" };
  res.end(JSON.stringify(req.session));
})
.listen(3000);
```

La différence réside dans l'utilisation du middleware `session`. On indique maintenant un objet `options` en paramètre de `connect.session(options)` comportant la clé `store`, initialisée avec l'adresse IP et le port de MongoDB, ainsi que le nom de la base de données qui sera utilisée (ici, `mydb`). La classe `MongoStore` est récupérée au moyen de l'instruction `require('connect-mongo')(connect)`.

L'intérêt de l'utilisation de MongoStore est que le serveur peut s'arrêter et redémarrer sans que les sessions des utilisateurs soient perdues, contrairement à MemoryStore, utilisé par défaut.

Middleware methodOverride : gérer les requêtes REST

Le middleware `methodOverride` permet de gérer les requêtes REST (*Representational State Transfer*), c'est-à-dire les requêtes adressées au serveur basées sur le type de la requête (indiqué dans l'attribut `method` des formulaires). On sait que les formulaires peuvent posséder un attribut `method` dont la valeur peut être :

- `GET` : utilisée pour que les valeurs transmises au serveur soient affichées dans l'URL de la requête. Après la validation du formulaire, on voit les valeurs saisies (dans le formulaire) s'afficher dans la barre d'adresses du navigateur. Ce mode de transmission n'est pas optimal car les valeurs peuvent être modifiées directement depuis l'URL affichée dans la barre d'adresses du navigateur sans passer par le formulaire. Le mode `GET` sera plutôt utilisé lors de la phase de tests du programme.
- `POST` : contrairement au mode `GET`, les données saisies dans le formulaire sont cachées lors de leur transmission au serveur car elles sont inscrites dans l'en-tête de la requête HTTP envoyée au serveur. L'utilisateur n'a donc pas les moyens de les modifier sans passer par le formulaire de saisie. Ce mode est donc plus adéquat lors de la mise en production de notre serveur.

En plus de ces deux modes traditionnels, REST permet de simuler d'autres valeurs de l'attribut `method` des formulaires, à savoir `PUT` et `DELETE`. D'autres valeurs encore existent, telles que `HEAD`, mais elles sont moins utilisées.

Regardons maintenant comment REST peut nous aider à architecturer nos programmes web.

Introduction à REST

REST considère que l'on peut effectuer quatre principales actions sur une donnée.

- Créer la donnée : on utilise dans ce cas une requête de type `POST`.
- Lire la donnée : on utilise dans ce cas une requête de type `GET`.
- Modifier la donnée : on utilise dans ce cas une requête de type `PUT`.
- Supprimer la donnée : on utilise dans ce cas une requête de type `DELETE`.

On voit ainsi que l'on peut effectuer les quatre actions de base sur une donnée en utilisant le type de la requête. Le problème, c'est que les valeurs autres que `GET` et `POST` ne peuvent pas être directement inscrites dans l'attribut `method` des formulaires, car ces valeurs ne sont pas reconnues par HTML. Il a donc fallu trouver un mécanisme pour transmettre les valeurs `PUT` et `DELETE` dans l'attribut `method`, en plus de `GET` et `POST`. C'est le rôle du middleware `methodOverride`.

Les quatre actions de base sur une donnée sont parfois appelées CRUD pour *Create, Read, Update et Delete*.

Utiliser le middleware methodOverride

Le middleware `methodOverride` est utilisé via la méthode `connect.methodOverride()`. L'utilisation des attributs `PUT` et `DELETE` se fait dans le formulaire HTML, ici par exemple en utilisant `PUT` :

Formulaire HTML utilisant la méthode PUT (fichier `form.html` dans le répertoire `forms`)

```
<form action=index.html method=put
```

Le formulaire possède l'attribut `method` valant toujours "`post`" (valeur obligatoire), et nous avons ajouté un champ caché (attribut `type` valant `hidden`) dont l'attribut `name` est "`_method`" tandis que l'attribut `value` est "`put`". Pour utiliser la méthode `DELETE`, il suffirait d'indiquer la valeur "`delete`" (au lieu de "`put`") dans l'attribut `value` du champ caché.

Remarquons que le champ est caché car on ne souhaite pas l'afficher dans la page HTML, mais seulement s'en servir pour la gestion interne du middleware `methodOverride`.

Si on écrit le précédent formulaire sans utiliser le middleware `methodOverride`, les données du formulaire seront transmises avec la méthode `POST`, comme indiqué dans le formulaire. Mais grâce à l'utilisation du middleware, l'attribut indiqué dans le champ caché écrase (*override*) celui inscrit dans le formulaire.

Il faut toutefois que l'attribut `method` indiqué dans le formulaire soit `POST`, sinon rien ne fonctionne.

Écrivons le programme Node qui utilise le middleware `methodOverride`.

Utiliser le middleware methodOverride

```
var connect = require("connect");
var app = connect();

app
    .use(connect.logger("dev"))
    .use(connect.static("forms"))
    .use(connect bodyParser())
    .use(connect.methodOverride())
    .use(function(req, res) {
        res.setHeader("Content-Type", "text/html");
```

```

    res.end(req.method + " " + req.url + "<br>" + JSON.stringify(req.body));
}
.listen(3000);

```

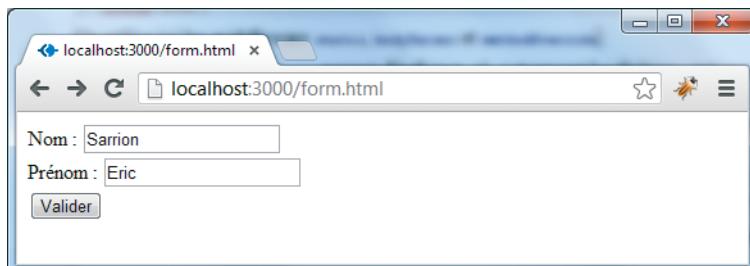
On utilise ici les middlewares `static`, `bodyParser` et `methodOverride`.

- Le middleware `static` permet d'indiquer où se trouvent les fichiers statiques, en particulier le fichier `form.html` associé au formulaire de saisie. On pourra ainsi accéder au formulaire en tapant l'URL `http://localhost:3000/form.html`.
- Le middleware `bodyParser` sert à récupérer dans `req.body` les données transmises dans le formulaire en mode `POST`, `PUT` ou `DELETE` (soit tous les types de requêtes sauf `GET`).
- Le middleware `methodOverride` sert à utiliser une méthode autre que `POST`. La méthode employée est indiquée dans le champ caché du formulaire dont l'attribut `name` est `_method`.

Lorsque le fichier `index.html` est activé lors de la validation du formulaire, l'envoi de la réponse est effectué. On indique ici que le contenu est HTML (au moyen de `res.setHeader("Content-Type", "text/html")`), puis on envoie la réponse, qui affichera la méthode utilisée (dans `req.method`).

Affichons le formulaire en tapant l'URL `http://localhost:3000/form.html`.

Figure 13–16
Saisie d'informations
dans le formulaire



Nous validons ensuite le formulaire en cliquant sur le bouton Valider.

Figure 13–17
Récupération des informations
saisies dans le formulaire avec
le middleware `methodOverride`



On retrouve bien les données saisies dans le formulaire et la méthode `PUT` utilisée. De plus, les données ne sont pas affichées dans la barre d'adresses, ce qui serait le cas si on avait utilisé la méthode `GET`.

Paramétriser le middleware `methodOverride`

Il est possible de paramétriser le middleware `methodOverride` en indiquant nous-mêmes la valeur de l'attribut `name` qui sera utilisée dans le champ caché. Par défaut, il vaut `"_method"`, mais on peut le modifier en utilisant `connect.methodOverride(methodName)`, où `methodName` est la nouvelle valeur que l'on souhaite utiliser dans les formulaires.

Utiliser `my-method` au lieu de `_method` dans le middleware `methodOverride`

```
var connect = require("connect");
var app = connect();

app
.use(connect.logger("dev"))
.use(connect.static("forms"))
.use(connect.bodyParser())
.use(connect.methodOverride("my-method"))
.use(function(req, res) {
  res.setHeader("Content-Type", "text/html");
  res.end(req.method + " " + req.url + "<br>" + JSON.stringify(req.body));
})
.listen(3000);
```

Le fichier `form.html` doit évidemment tenir compte de cette modification.

Fichier `form.html` (dans le répertoire `forms`)

```
<form action=index.html method=post>
  Nom : <input name=nom value=Sarrion /> <br>
  Prénom : <input name=prenom value=Eric /> <br>
  <input type=submit value=Valider />
  <input type=hidden name=my-method value=put />
</form>
```

14

Introduction au framework Express

Le framework Express (également nommé Express.js) est un ensemble de modules Node permettant de créer facilement des applications web à partir de Node. Il est basé sur le modèle MVC (*Model View Controller*) qui permet de donner une architecture cohérente à une application web. Cette architecture MVC fait qu'Express est très populaire dans la communauté Node, car de plus en plus d'applications Node sont construites sur ces bases.

De plus, le framework Express est basé sur le module Connect, étudié précédemment. Nous allons donc retrouver beaucoup de notions abordées dans les chapitres précédents sur Connect, en particulier les middlewares.

Dans ce chapitre, nous apprendrons à installer le framework Express, puis à créer simplement une application standard. Enfin, nous expliquerons le modèle MVC sur lequel Express est architecturé.

Installer le framework Express

Express est un ensemble de modules et d'exécutables disponibles depuis `npm`. Dans un interpréteur de commandes, il suffit donc de taper la commande `npm install -g express` pour installer Express en global et le rendre disponible pour toutes nos applications Node (en particulier l'exécutable `express` qui permet de créer une application vierge).

À la fin de l'installation, on obtient :

Figure 14-1
Installation du framework
Express

```

C:\ Invité de commandes
npm WARN package.json uid2@0.0.2 No README.md file found!
npm http 304 https://registry.npmjs.org/formidable/1.0.14
npm http 304 https://registry.npmjs.org/bytes/0.2.0
npm http 200 https://registry.npmjs.org/qs/0.6.5
npm http GET https://registry.npmjs.org/qs/-/qs-0.6.5.tgz
npm http 200 https://registry.npmjs.org/qs/-/qs-0.6.5.tgz
C:\Users\Eric\AppData\Roaming\npm>express -> C:\Users\Eric\AppData\_modules\express@3.5.0 C:\Users\Eric\AppData\Roaming\npm\node_modules\exp
ress methods@0.0.1
fresh@0.2.0
cookie-signature@1.0.1
range-parser@0.0.4
buffer-crc32@0.2.1
cookie@0.1.0
mkdirp@0.3.5
debug@0.7.2
commander@1.2.0 <keypress@0.1.0>
send@0.1.4 <mine@1.2.0>
connect@2.8.5 <uid2@0.0.2, qs@0.6.5, pause@0.0.1, bytes@0.2.0.14>
C:\Users\Eric\Documents\Node.js>

```

Vérifions que Express est accessible en tapant la commande `express -h`.

Figure 14-2
Aide sur la commande express

```

C:\ Invité de commandes
C:\Users\Eric\Documents\Node.js>express -h
Usage: express [options] [dir]
Options:
  -h, --help           output usage information
  -V, --version        output the version number
  -s, --sessions       add session support
  -e, --ejs            add ejs engine support (defaults to jade)
  -J, --jshtml          add jshtml engine support (defaults to J
  -H, --hogan           add hogan.js engine support
  -c, --css <engine>    add stylesheet <engine> support (less ist
  to plain css)
  -f, --force           force on non-empty directory
C:\Users\Eric\Documents\Node.js>

```

Une aide associée à la commande `express` s'affiche. On va maintenant se servir de la commande `express` pour créer une application web utilisant Express.

Créer une application web avec Express

Une fois Express installé, la commande `express` devient accessible. Il suffit de taper `express nomappli` pour créer le répertoire et les fichiers de base de l'application ayant pour nom `nomappli`.

Créons une application `testexpress` au moyen de la commande `express testexpress`.

Figure 14–3
Application créée avec Express

```
c:\ Invité de commandes
C:\Users\Eric\Documents\Node.js>express testexpress
  create : testexpress
  create : testexpress/package.json
  create : testexpress/app.js
  create : testexpress/public
  create : testexpress/public/javascripts
  create : testexpress/public/stylesheets
  create : testexpress/public/stylesheets/style.css
  create : testexpress/views
  create : testexpress/views/layout.jade
  create : testexpress/views/index.jade
  create : testexpress/public/images
  create : testexpress/routes
  create : testexpress/routes/index.js
  create : testexpress/routes/user.js

  install dependencies:
    $ cd testexpress && npm install

  run the app:
    $ node app

C:\Users\Eric\Documents\Node.js>_
```

La commande `express testexpress` a créé le répertoire `testexpress` et ses sous-répertoires, ainsi que divers fichiers que nous découvrirons par la suite.

Le fichier le plus important pour l'instant est le fichier `package.json` créé dans le répertoire de l'application `testexpress`. Le contenu de ce fichier est le suivant.

Fichier `package.json` (dans `testexpress`)

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.3.5",
    "jade": "*"
  }
}
```

On retrouve le contenu d'un fichier `package.json` classique, dont les clés sont expliquées dans le tableau 14–1.

Tableau 14–1 Clés du fichier `package.json`

Clé	Signification
<code>name</code>	Nom de l'application
<code>version</code>	Numéro de version de l'application

Tableau 14–1 Clés du fichier package.json (suite)

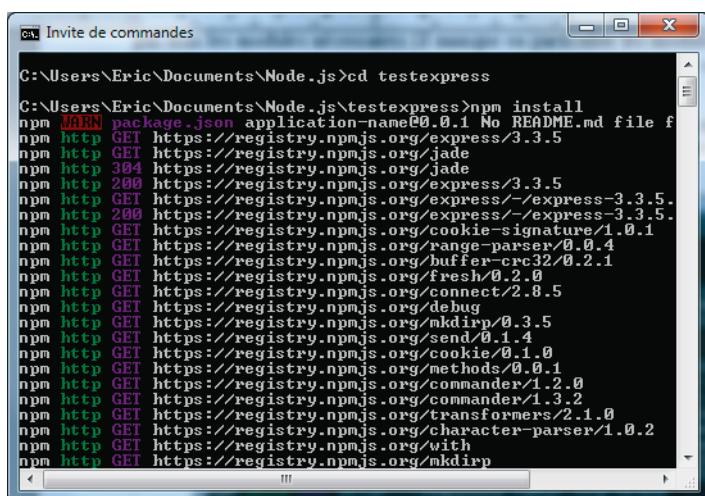
Clé	Signification
<code>private</code>	Indique si l'application est à usage privé ou si elle est destinée à être publiée sur <code>npm</code> .
<code>scripts + start</code>	La clé <code>start</code> incluse dans la clé <code>scripts</code> indique une commande qui se déclenchera si l'on tape la commande <code>npm start</code> . Ici, cela exécute <code>node app.js</code> , indiquant que le fichier de démarrage de l'application Node est <code>app.js</code> .
<code>dependencies</code>	Modules nécessaires pour exécuter l'application <code>testexpress</code> créée avec Express. Ici, l'application nécessite les modules <code>express</code> et <code>jade</code> , dont les versions sont indiquées.

Il faut bien comprendre que l'utilisation de la commande `express` pour créer une application ne génère pas tous les modules nécessaires (il manque en particulier les modules `express` et `jade`). La façon la plus simple d'inclure ces modules dans notre application est de faire comme indiqué dans la fenêtre de commandes : `cd testexpress` puis `npm install`.

La commande `npm install` permet d'utiliser le fichier `package.json` précédent et de charger automatiquement les modules manquants. Tapons cette commande dans le répertoire `testexpress`.

Figure 14–4

Début de l'installation des modules de l'application créée par Express



```
C:\Users\Eric\Documents\Node.js>cd testexpress
C:\Users\Eric\Documents\Node.js\testexpress>npm install
npm [WARN] package.json application-name@0.0.1 No README.md file found
npm http GET https://registry.npmjs.org/express/3.3.5
npm http GET https://registry.npmjs.org/jade
npm http 304 https://registry.npmjs.org/jade
npm http 200 https://registry.npmjs.org/express/3.3.5
npm http GET https://registry.npmjs.org/express/-/express-3.3.5
npm http 200 https://registry.npmjs.org/express/-/express-3.3.5
npm http GET https://registry.npmjs.org/cookie-signature/1.0.1
npm http GET https://registry.npmjs.org/range-parser/0.0.4
npm http GET https://registry.npmjs.org/buffer-crc32/0.2.1
npm http GET https://registry.npmjs.org/fresh/0.2.0
npm http GET https://registry.npmjs.org/connect/2.8.5
npm http GET https://registry.npmjs.org/debug
npm http GET https://registry.npmjs.org/mkdirp/0.3.5
npm http GET https://registry.npmjs.org/send/0.1.4
npm http GET https://registry.npmjs.org/cookie/0.1.0
npm http GET https://registry.npmjs.org/methods/0.0.1
npm http GET https://registry.npmjs.org/commander/1.2.0
npm http GET https://registry.npmjs.org/commander/1.3.2
npm http GET https://registry.npmjs.org/transformers/2.1.0
npm http GET https://registry.npmjs.org/character-parser/1.0.2
npm http GET https://registry.npmjs.org/with
npm http GET https://registry.npmjs.org/mkdirp
```

À la fin de l'exécution de cette commande, il s'affiche :

Figure 14–5
Fin de l'installation
des modules de l'application
créée par Express

```
npm http 200 https://registry.npmjs.org/async
express@3.5.1 node_modules\express
  methods@0.0.1
  fresh@0.2.0
  range-parser@0.0.4
  cookie-signature@1.0.1
  buffer-crc32@0.2.1
  cookie@0.0.1
  debug@0.7.2
  mkdirp@0.3.5
  commander@1.2.0 <(keypress@0.1.0)
  send@0.1.4 <(mime@1.2.10)
  connect@2.8.5 <(uid2@0.0.2, pause@0.0.1, qs@0.6.5, bytes@0.2.0.14)

jade@0.34.1 node_modules\jade
  character-parser@1.0.2
  mkdirp@0.3.5
  commander@1.3.2 <(keypress@0.1.0)
  monocle@0.1.50 <(readirp@0.2.5)
  transformers@2.1.0 <(promise@2.0.0, css@1.0.8, uglify-js@2.2.0.1.0 <(uglify-js@2.3.6))
  constantinople@1.0.1 <(uglify-js@2.3.6)

C:\Users\Eric\Documents\Node.js\testexpress>
```

L'exécution de l'application s'effectue en tapant la commande `node app` (comme indiqué à la fin de l'exécution de la commande `express testexpress`), mais on peut aussi taper `npm start`, qui exécute lui-même `node app.js` comme indiqué dans le fichier `package.json`.

Tapons simplement la commande `node app`, en étant dans le répertoire `testexpress`.

Figure 14–6
Lancement de l'application
Express

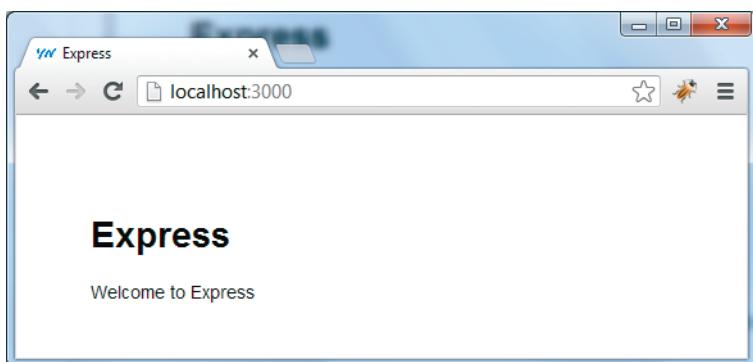
```
C:\Users\Eric\Documents\Node.js\testexpress>node app
Express server listening on port 3000

C:\Users\Eric\Documents\Node.js\testexpress>
```

Un serveur HTTP a été créé, puis lancé sur le port 3000. Testons ce serveur en saisissant l'URL `http://localhost:3000` dans un navigateur.

Figure 14–7

Page d'accueil de l'application créée par Express



Une page HTML par défaut s'affiche dans le navigateur, montrant que le serveur HTTP est démarré et répond aux requêtes des utilisateurs. Analysons les informations affichées dans la console du serveur.

Figure 14–8

Affichage de la console du serveur, suite aux requêtes d'affichage précédentes

A screenshot of a terminal window titled "Invite de commandes - node app". The window shows the command "node app" being run. Below it, the server logs show the following requests:

```
C:\Users\Eric\Documents\Node.js\testexpress>node app
Express server listening on port 3000
GET / 200 605ms - 170b
GET /stylesheets/style.css 200 8ms - 117b
GET /stylesheets/style.css 304 2ms
```

Les requêtes effectuées au serveur sont également affichées dans la console.

Architecture d'une application Express

Regardons comment a été construite l'application que nous venons d'exécuter. Commençons par le fichier `app.js` qui est celui qui a été exécuté par la commande `node app`.

Le fichier `app.js`

Le fichier `app.js` est traditionnellement le fichier de démarrage d'une application construite avec Express. Il est créé par la commande `express nomappli`, mais il peut ensuite être modifié pour l'adapter à notre application.

Ce document est la propriété exclusive de Alexandre Mbiam (alexmbiam@gmail.com) - 15 avril 2016 à 11:53

Voici le fichier `app.js` se trouvant dans le répertoire principal de l'application, tel qu'il a été construit par Express.

Fichier app.js

```
/**  
 * Module dependencies.  
 */  
  
var express = require('express')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.favicon());  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.methodOverride());  
app.use(app.router);  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
  app.use(express.errorHandler());  
}  
  
app.get('/', routes.index);  
app.get('/users', user.list);  
  
http.createServer(app).listen(app.get('port'), function(){  
  console.log('Express server listening on port ' + app.get('port'));  
});
```

On constate beaucoup de similitudes entre ce programme et ce que nous avions étudié dans le module Connect. L'objet `app` est ici le même que celui utilisé dans Connect, la différence étant qu'il est maintenant créé à partir de la fonction `express()` du module Express (au lieu de la fonction `connect()` du module Connect).

En fait, l'objet `app` défini par Express est une extension de l'objet `app` défini par Connect, avec de nouveaux middlewares créés par Express. Mais les middlewares de Connect sont utilisables dans Express, c'est pour cela qu'on utilise la méthode `use()` définie dans Connect, tandis que le middleware est préfixé de l'objet `express` au lieu de l'objet `connect`.

Les nouveautés ici introduites par Express (par rapport à Connect) sont :

- la définition des méthodes `get()` et `set()` sur l'objet `app` ;
- l'utilisation du nouveau middleware `app.router`.

Nous reviendrons sur ces nouveautés (et beaucoup d'autres) dans les chapitres suivants.

Remarquons ici que la dernière instruction du fichier crée le serveur HTTP, qui sera à l'écoute du port 3000 par défaut. Cette instruction peut être remplacée par celle que l'on avait écrite en étudiant le module Connect.

Ainsi les lignes suivantes :

```
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Sont équivalentes à celles-ci :

```
app.listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Si vous souhaitez plus de précisions sur l'équivalence de ces deux formes d'écriture, reportez-vous au chapitre 12, « Introduction au module Connect », section « Utiliser l'objet `app` en tant que fonction de callback ».

Autres répertoires et fichiers créés par Express

La commande `express testexpress` a également créé un ensemble de répertoires et de fichiers dans le répertoire principal de l'application, ici `testexpress`. Les répertoires créés sont les suivants.

Tableau 14–2 Répertoires créés par Express

Répertoire	Signification
node_modules	Il contient tous les modules installés lors de la commande <code>npm install</code> . Il s'agit des modules <code>express</code> et <code>jade</code> qui étaient indiqués dans le fichier <code>package.json</code> .
public	Il contient les sous-répertoires <code>images</code> , <code>javascripts</code> et <code>stylesheets</code> . Les fichiers inscrits directement dans le répertoire <code>public</code> pourront être accédés en donnant directement leur nom, car le répertoire <code>public</code> est mis en accès statique grâce au middleware <code>static</code> utilisé par Express dans <code>app.js</code> .
routes	Ce répertoire permet de définir le format des routes de l'application, c'est-à-dire les URL qui seront disponibles pour traiter les requêtes des utilisateurs. Nous y reviendrons dans le chapitre suivant.
views	Contient les vues de l'application. Deux fichiers sont actuellement présents : <code>index.jade</code> et <code>layout.jade</code> . L'extension <code>.jade</code> indique que l'on devra utiliser le module <code>jade</code> afin d'écrire les vues de l'application. D'autres extensions sont disponibles, en particulier <code>.ejs</code> (<i>Embedded JavaScript</i>).

Les répertoires `routes` et `views` permettent d'introduire le concept de modèle MVC, que nous étudions dans la section qui suit.

Le modèle MVC

Le modèle MVC (*Model View Controller*) représente une façon de découper un programme informatique en trois parties.

- Le modèle : il correspond à la gestion des données qui seront manipulées. Il concerne l'accès à la base de données, avec les quatre opérations que l'on peut effectuer sur les données (CRUD, soit *Create, Read, Update et Delete*).
- Les vues : elles correspondent à l'affichage des données aux utilisateurs, permettant ainsi de les créer, les modifier, les mettre à jour ou les supprimer. Les vues seront ici des pages HTML qui permettront l'interaction avec les utilisateurs.
- Les contrôleurs : ils permettent l'enchaînement des vues entre elles, en fonction des actions des utilisateurs dans chacune d'elles. On appelle cet enchaînement le « routage ».

Cette décomposition permet de simplifier l'écriture des programmes, et surtout elle permet une conception et une maintenance plus aisées de ces programmes. L'étude du module Express consistera principalement à comprendre comment écrire des contrôleurs et des vues. La gestion des modèles sera effectuée par des modules indépendants d'Express, tels que ceux permettant de gérer la base de données MongoDB que l'on étudiera dans la partie suivante de l'ouvrage.

Tout d'abord, voyons comment écrire des contrôleurs dans nos applications Express. C'est le rôle du routage effectué par le middleware `router` inclus dans Express, que nous étudions dans le chapitre qui suit.

15

Routage des requêtes avec Express

Ce chapitre concerne la gestion des routes dans Express. Le routage est une partie importante d'une application car il gère la façon d'accéder aux ressources, en particulier en définissant les URL qui permettront l'accès aux différentes pages HTML du site.

Contrairement à un langage comme PHP qui associe une URL à l'emplacement d'un fichier sur le serveur (en indiquant le chemin d'accès et le nom du fichier PHP), Express dissocie les deux. On peut donc indiquer une URL qui ne correspond pas directement à un fichier sur disque. Le mécanisme de routage inclus dans Express va mettre les deux en correspondance : une URL sera associée à un affichage via le mécanisme de routage.

Qu'est-ce qu'une route ?

La première des actions à effectuer lors de la construction d'une application avec Express sera de définir les URL qui seront utilisées pour afficher les pages HTML dans le navigateur. Ces URL vont servir à la définition des routes.

Analyse des routes déjà présentes dans app.js

Par défaut, deux routes sont déjà écrites dans le fichier `app.js` que nous avons précédemment utilisé au chapitre 14. On les réécrit ci-dessous :

Routes par défaut définies dans app.js

```
app.get('/', routes.index);
app.get('/users', user.list);
```

Chacune des instructions ci-dessus définit une route.

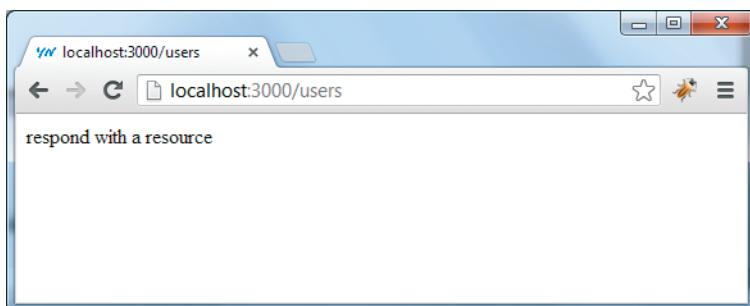
- La première route sera activée lorsque l'URL sera "/", par exemple lorsqu'on saisira l'URL `http://localhost:3000` dans la barre d'adresses du navigateur. Le second argument (ici, `routes.index`) définit une fonction de callback qui sera activée lorsque cette URL sera reconnue.
- La seconde route sera activée lorsque l'URL sera "/users", par exemple lorsqu'on saisira l'URL `http://localhost:3000/users` dans la barre d'adresses du navigateur. Comme précédemment, le second argument (ici, `user.list`) représente une fonction de callback qui sera activée lorsque cette URL sera reconnue.

Comme aucune autre définition de route n'est effectuée, toutes les autres formes d'URL produiront une erreur.

Remarquez qu'on utilise ici la méthode `app.get()` pour la définition des deux routes. Cela signifie que ces routes seront activées lorsque l'URL correspondante sera associée à une requête de type `GET`, ce qui est le cas si l'URL est entrée directement dans la barre d'adresses du navigateur.

Par exemple, en entrant l'URL `http://localhost:3000/users` dans la barre d'adresses du navigateur, il s'affiche :

Figure 15-1
Affichage de l'URL /users dans l'application Express

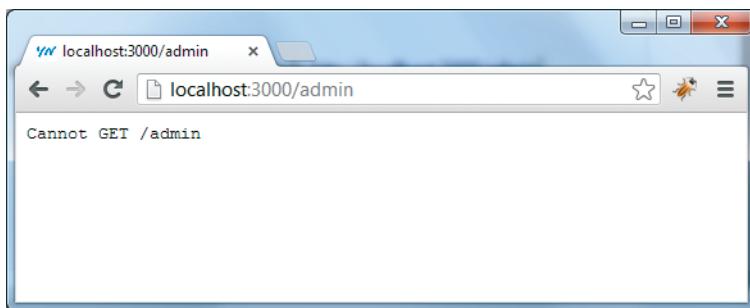


Cette URL ne serait pas activée si on y accédait depuis un formulaire utilisant, par exemple, la méthode `POST`, vu qu'elle est définie uniquement pour la méthode `GET`.

Entrons maintenant une URL non définie dans le routage, par exemple `http://localhost:3000/admin`.

Figure 15–2

Affichage d'une URL inconnue



Un message d'erreur (géré par Express) s'affiche, indiquant que cette URL ne peut pas être affichée. Le message nous dit qu'on ne peut pas accéder à l'URL "`/admin`" pour une requête de type `GET`. En effet, l'URL ayant été entrée à partir de la barre d'adresses du navigateur, elle correspond à une requête de type `GET`. D'autres types de requêtes existent : `GET`, `POST`, `PUT` et `DELETE` pour ne citer que les plus usuelles (ou encore `HEAD`, `OPTIONS`, `TRACE`, etc., consultez la documentation technique sur le protocole HTTP pour plus de détails).

Une route correspond donc à un type de requête (`GET`, `POST`, `PUT` ou `DELETE`) associé à une URL. Lorsque ces deux éléments sont reconnus par Express dans la route utilisée, la fonction de callback associée est déclenchée.

Voyons maintenant comment définir une nouvelle route dans Express.

Ajout d'une nouvelle route dans app.js

Définissons l'URL "`/admin`" dans le routage. On ajoute la ligne suivante dans le fichier `app.js`.

Ajout de la route `/admin` dans le routage

```
app.get("/admin", function(req, res) {  
    res.end("<h1>Bienvenue dans la partie administration du site</h1>");  
});
```

Nous avons ici défini la nouvelle route "`/admin`" qui sera activée par une requête de type `GET` (d'où l'utilisation de `app.get()`). La fonction de callback utilisée en second argument prend les deux paramètres `req` et `res`, respectivement associés aux objets `request` et `response` définis dans Node pour les serveurs HTTP. Ce sont les mêmes objets que ceux que nous avions utilisés lors de l'étude du module Connect (voir le chapitre 12, « Introduction au module Connect »).

La fonction de callback ici écrite peut être considérée comme un middleware, mais qui ne sera activé que lorsque cette route (définie par l'URL et le type de la requête) sera reconnue. Plutôt que d'utiliser `app.get()`, on peut utiliser `app.use()` comme ci-dessous.

Utiliser `app.use()` pour définir la route

```
app.use(function(req, res) {
  if (req.method == "GET" && req.url == "/admin")
    res.end("<h1>Bienvenue dans la partie administration du site</h1>");
});
```

La méthode `app.use()` permet de définir un middleware qui sera appelé pour chaque requête effectuée au serveur (quel que soit son type). On doit donc filtrer le type de la requête (ici, `GET`) ainsi que l'URL utilisée (ici, `"/admin"`).

Vérifions que cela fonctionne de la même façon.

Figure 15–3
Prise en compte d'une nouvelle route dans l'application



La méthode `app.get()` est donc une forme condensée de la méthode `app.use()`. Selon le type de la requête, on a aussi `app.post()`, `app.put()` et `app.delete()`.

Utiliser la route en tant que middleware

Nous avons dit dans la section précédente que la fonction de callback utilisée dans la méthode `app.get()` pouvait être considérée comme un middleware. On sait que dans ce cas, la fonction de callback peut s'écrire avec trois paramètres : `req`, `res` et `next` (voir le chapitre 12).

Écrivons pour la route `GET /admin` utilisant la requête `GET`, un middleware positionnant l'en-tête `"X-nom"` à la valeur `"Eric"`. La route doit continuer à afficher la page précédente.

Ajouter un middleware sur la route /admin pour la requête GET

```
app.get("/admin", function(req, res, next) {
  res.setHeader("X-nom", "Eric");
  next();
});

app.get("/admin", function(req, res) {
  res.end("<h1>Bienvenue dans la partie administration du site</h1>");
});
```

La première méthode `app.get()` utilise le middleware dont le code est la fonction de callback associée. L'appel de la fonction `next()` dans le middleware permet de passer au middleware suivant, sinon le serveur n'envoie pas la réponse au navigateur qui l'attend.

Remarquons que le module Express permet d'écrire le code précédent sous la forme d'un seul appel à la méthode `app.get()`, en écrivant les deux fonctions de callback à la suite dans la liste des arguments transmis à `app.get()`.

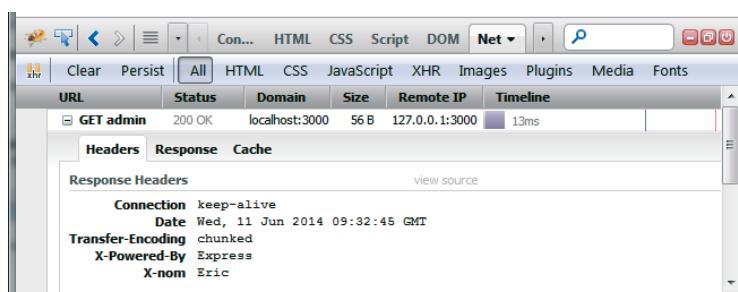
Écrire un seul appel à `app.get()` qui enchaîne plusieurs fonctions de callback

```
app.get("/admin", function(req, res, next) {
  res.setHeader("X-nom", "Eric");
  next();
}, function(req, res) {
  res.end("<h1>Bienvenue dans la partie administration du site</h1>");
});
```

Plusieurs fonctions de callback peuvent ainsi être enchaînées lors de la définition de la route, à condition d'utiliser la méthode `next()` pour permettre leur chaînage. Seule la fonction de callback qui renvoie le résultat au navigateur n'a pas besoin d'utiliser `next()`.

Utilisons Firefox et Firebug pour vérifier que cela fonctionne.

Figure 15–4
Utilisation de plusieurs fonctions de callback pour la route



L'en-tête "X-nom" est bien positionné dans les en-têtes transmis au navigateur, et le résultat est affiché dans la page HTML, montrant que les deux middlewares se sont exécutés.

Architecture REST

Express est très lié à l'architecture REST car il utilise le type de requête HTTP ([GET](#), [POST](#), [PUT](#) ou [DELETE](#)) pour effectuer ses traitements. Nous avons ainsi vu que le type de la requête attendue était défini par la méthode utilisée sur l'objet `app`.

Méthodes liées aux types de requêtes HTTP

Le tableau 15-1 liste les méthodes définies dans Express qui seront activées en fonction du type de la requête HTTP.

Tableau 15-1 Méthodes liées aux requêtes HTTP

Méthode	Signification
<code>app.get(url, callback)</code>	Appelle la fonction de callback lorsque l'URL est activée sur une requête GET .
<code>app.post(url, callback)</code>	Appelle la fonction de callback lorsque l'URL est activée sur une requête POST .
<code>app.put(url, callback)</code>	Appelle la fonction de callback lorsque l'URL est activée sur une requête PUT .
<code>app.delete(url, callback)</code>	Appelle la fonction de callback lorsque l'URL est activée sur une requête DELETE .

D'autres types de requêtes existent, mais elles sont moins utilisées. Elles sont également associées à des méthodes définies par Express sur l'objet `app`. Ces méthodes portent le même nom que le type de la requête (par exemple, `app.head()` pour une requête de type [HEAD](#)).

Les types de requêtes suivants sont également pris en compte par Express (liste non exhaustive) :

- HEAD
- TRACE
- OPTIONS
- CONNECT
- PATCH
- M-SEARCH

- NOTIFY
- SUBSCRIBE
- UNSUBSCRIBE

Utiliser le middleware methodOverride dans Express

Le middleware `methodOverride` est défini dans le module Connect. Or, on sait que les middlewares définis dans Connect sont également accessibles dans Express, en les préfixant par l'objet `express` au lieu de l'objet `connect`. Utilisons ce middleware afin de créer un formulaire activant la route "`/admin`" sur une requête de type `PUT`.

Le formulaire s'écrit comme suit, en respectant les conventions du middleware `methodOverride` :

Fichier `form.html` (dans le répertoire `forms`)

```
<form action=/admin method=post>
    Nom : <input name=nom value=Sarrion /> <br>
    Prénom : <input name=prenom value=Eric /> <br>
    <input type=submit value=Valider />
    <input type=hidden name=_method value=put />
</form>
```

L'URL à activer est indiquée dans l'attribut `action` du formulaire. L'attribut `method` de celui-ci doit valoir `POST` pour que cela fonctionne.

Le type de requête `PUT` est indiqué dans l'attribut `value` du champ caché, lequel doit posséder l'attribut `name` ayant la valeur "`_method`". Ceci est obligatoire afin que le middleware `methodOverride` fonctionne de la manière attendue.

Le fichier `app.js` définissant la nouvelle route s'écrit alors :

Fichier `app.js`

```
/***
 * Module dependencies
 */

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');
```

```
var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public'))));
app.use(express.static(path.join(__dirname, 'forms'))));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

app.put("/admin", function(req, res) {
  res.end("<h1>Bienvenue dans la partie administration du site</h1>");
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

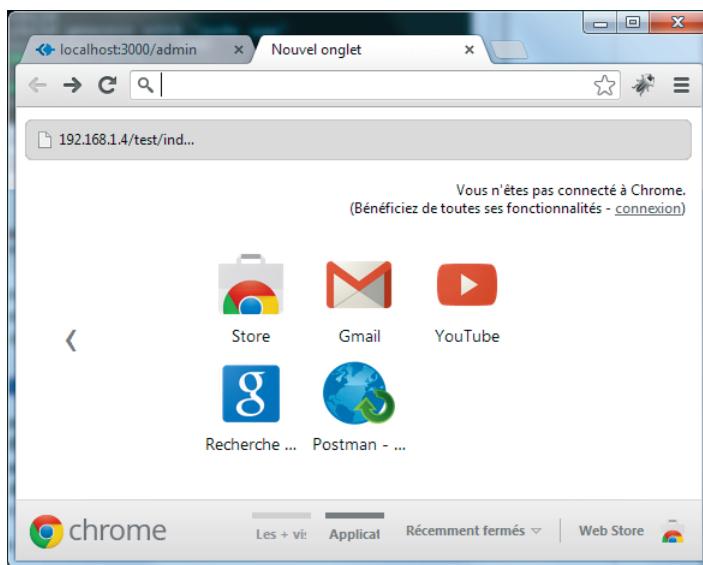
L'inclusion du middleware `methodOverride` est ici obligatoire, mais il est déjà ajouté par défaut par Express lors de la création du fichier `app.js`. La méthode `app.put()` est ici utilisée afin de gérer la requête `PUT` émise lors de la validation du formulaire.

Utiliser l'outil Postman sous Chrome

Le test d'une route peut parfois être difficile à effectuer, en particulier si les types de requêtes sont autres que `GET`. En effet, les requêtes de type `GET` peuvent facilement être utilisées dans une fenêtre du navigateur en tapant l'URL correspondante dans la barre d'adresses, mais cela est impossible pour les autres types de requêtes. Il faut alors passer par un formulaire comme celui écrit précédemment, ou alors utiliser un outil comme Postman qui simule tous les types de requêtes possibles.

Postman est un plug-in qui s'utilise dans le navigateur Chrome. Après installation, une icône s'affiche dans le navigateur.

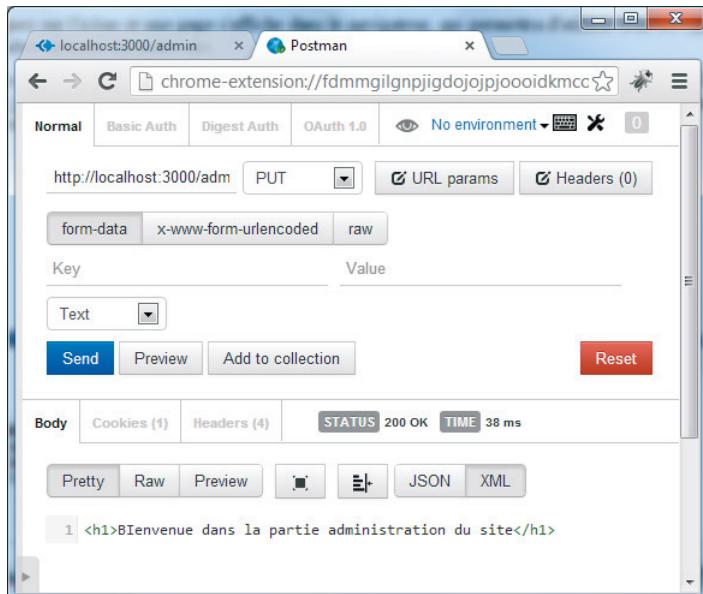
Figure 15–5
Plug-in Postman
de Chrome installé



Cliquez sur cette icône. Une nouvelle page s'ouvre alors dans le navigateur, qui permettra d'utiliser tous les types de requêtes souhaités sur notre site.

Vous pouvez alors saisir l'URL à tester, le type de requête à utiliser... Tapons l'URL <http://localhost:3000/admin> et positionnons le type de requête à [PUT](#). En considérant que le serveur s'est lancé en exécutant le programme précédent, on obtient :

Figure 15–6
Utilisation du plug-in
Postman dans Chrome



Cet outil est bien pratique pour effectuer ce type de test. On vous laisse découvrir vous-même l'éventail de ses possibilités...

Définir un middleware valable pour tous les types de routes

Nous avons vu que l'utilisation des méthodes `app.get()`, `app.post()`, `app.put()` et `app.delete()` déclencheait un traitement qui s'exécute si la route correspond. La route signifie ici l'URL et le type de requête.

Les questions qui se posent sont alors les suivantes.

- Est-il possible de définir un traitement qui s'effectuerait pour n'importe quel type de requête, à savoir aussi bien une requête de type `GET`, `POST`, `PUT` ou `DELETE` ?
- Est-il possible de définir un traitement qui s'effectuerait pour toutes les routes, indépendamment de l'URL indiquée et du type de la requête ?

La réponse à la seconde question est facile : il suffit d'utiliser `app.use()` qui permet de définir un middleware qui s'exécutera quelle que soit la route utilisée. Express s'en sert lorsqu'il écrit dans le code généré au début du fichier `app.js` les nombreux appels aux middlewares de Connect.

Pour répondre à la première question, deux solutions sont possibles.

- La première solution consiste à utiliser `app.use()` et à filtrer uniquement sur l'URL utilisée (dans la fonction de callback associée). C'est ce procédé que nous avions utilisé dans la section « Ajout d'une nouvelle route dans app.js ».
- La seconde solution est la plus directe à utiliser. Express a défini la nouvelle méthode `app.all(url, callback)` qui permet de spécifier un traitement pour une URL quel que soit le type de requête qui le sollicite.

Quelle que soit la solution choisie, la fonction de callback associée est considérée comme un middleware. Elle peut donc renvoyer elle-même une réponse au navigateur ou passer le relais à un autre middleware par l'utilisation du paramètre `next`.

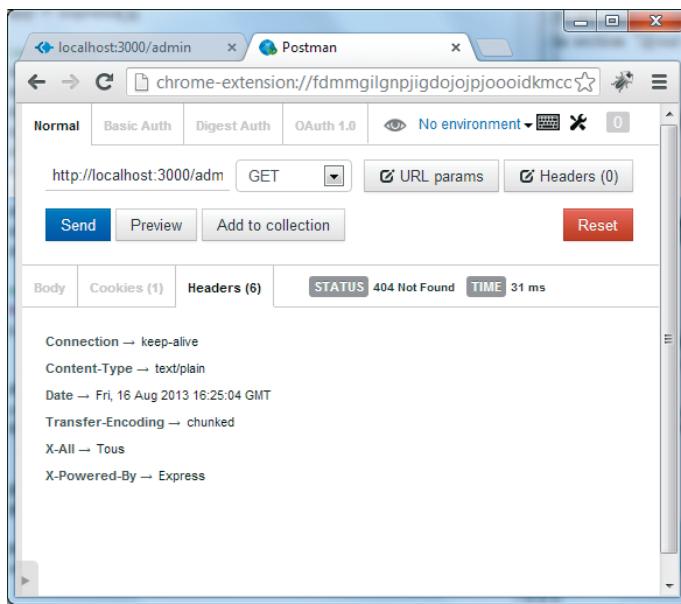
Par exemple, utilisons `app.all()` pour positionner un en-tête "X-all" valant "Tous".

Utiliser `app.all()`

```
app.all("/", function(req, res, next) {  
    res.setHeader("X-All", "Tous");  
    next();  
});
```

Postman permet de voir les en-têtes envoyés quel que soit le type de requête utilisé.

Figure 15–7
Simulation de l'envoi
d'une URL dans Chrome



On a ici utilisé une requête de type `GET`, et l'en-tête "X-All" est bien positionné. Elle est également placée quel que soit le type de requête.

Objet `app.routes` défini par Express

Une fois les routes définies au moyen des méthodes `app.get()` et autres, il peut être intéressant de connaître l'ensemble des routes utilisées.

Express définit un objet `app.routes`, dont les clés sont les noms des méthodes associées aux types de requêtes. On aura donc les clés "`get`", "`post`", "`put`", "`delete`", "`head`", etc.

Chacune des clés donne accès à un tableau d'objets, chacun d'entre eux définissant une route utilisée pour ce type de requête. Le tableau peut être vide si aucune route n'est associée à ce type de requête.

Afficher les routes utilisées

Voici un programme qui affiche le contenu de l'objet `app.routes.get`, définissant les routes accessibles avec le type de requête `GET`. Ces routes sont celles inscrites dans le programme lui-même. Elles seront affichées au moyen de la route `GET /admin`.

Afficher les routes définies dans app.routes.get

```
/**  
 * Module dependencies  
 */  
  
var express = require('express');  
var routes = require('./routes');  
var user = require('./routes/user');  
var http = require('http');  
var path = require('path');  
var util = require('util');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.favicon());  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.methodOverride());  
app.use(app.router);  
app.use(express.static(path.join(__dirname, 'public')));  
app.use(express.static(path.join(__dirname, 'forms')));  
  
// development only  
if ('development' == app.get('env')) {  
    app.use(express.errorHandler());  
}  
  
app.all("/admin", function(req, res, next) {  
    res.setHeader("X-All", "Tous");  
    next();  
});  
  
app.get('/', routes.index);  
app.get('/users', user.list);  
  
app.get("/admin", function(req, res) {  
    var html = "";  
    app.routes.get.forEach(function(route) {  
        html += "<p>" + util.inspect(route) + "</p>";  
    });  
    res.end(html);  
});
```

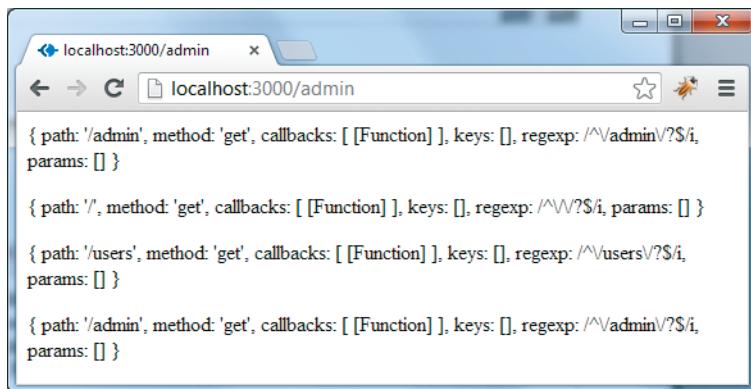
```
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Nous avons laissé les routes définies par défaut ainsi que celle définie par `app.all()`. La route `GET /admin` est celle qui effectue l'affichage des routes se trouvant en type `GET`. On utilise la méthode `util.inspect()` du module `util` car elle affiche plus d'informations que la méthode `JSON.stringify()` traditionnelle. Pour cela, on inclut bien sûr le module `util` dans notre programme.

Affichons le résultat en tapant l'URL `http://localhost/admin` dans le navigateur.

Figure 15–8

Contenu de l'objet `app.routes`



On voit que chaque route est définie comme un objet `{ path, method, callbacks, keys, regexp, params }`. Le path `/admin` est présent deux fois, car deux middlewares ont été utilisés sur cette route (l'un avec `app.all("/admin")` et l'autre avec `app.get("/admin")`).

Supprimer une route utilisée

Bien que cette problématique soit assez rare, il peut arriver que l'on souhaite supprimer une route déjà définie. On utilise alors l'opérateur `delete objRoute` qui détruit l'objet `objRoute` définissant la route.

Supprimons par exemple la route définissant `GET /users` lors de l'utilisation de la route `GET /admin`.

Supprimer la route GET /users et afficher les routes encore disponibles

```
app.get("/admin", function(req, res) {
  var html = "";
  app.routes.get.forEach(function(route, i) {
    if (route.path == "/users") delete app.routes.get[i];
  });
  app.routes.get.forEach(function(route) {
    html += "<p>" + util.inspect(route) + "</p>";
  });
  res.end(html);
});
```

Pour que la route soit supprimée de `app.routes`, il faut supprimer l'objet directement de `app.routes` et non pas sa copie, accessible dans le paramètre `route` de la fonction de callback.

Définir l'URL dans les routes

Jusqu'à présent, nous avons écrit l'URL de la route sous forme d'une simple chaîne de caractères. L'URL correspond au premier paramètre des méthodes `app.get()`, `app.post()`, etc. Les URL précédemment utilisées étaient `"/"`, `"/users"` et `"/admin"`. Nous allons voir maintenant que cette chaîne de caractères peut être plus complexe à écrire, selon les besoins que l'on a. Ce que nous expliquons ici s'applique quel que soit le type de requête dans la route (`GET`, `POST`, `PUT` ou `DELETE`).

URL définie sous forme de chaîne de caractères

La façon la plus simple de définir une URL est d'indiquer la chaîne de caractères qui la compose. Par exemple, `"/"`, `"/users"`, `"/admin"`, `"/users/12"`, `"/admin/users"`, etc.

Ces cas correspondent à ceux que nous avons précédemment utilisés dans notre programme. Nous n'en parlerons pas plus ici.

URL définie sous forme d'expression régulière

Plutôt que d'indiquer une chaîne de caractères, on peut utiliser une expression régulière pour représenter l'URL. L'expression régulière est alors entourée des caractères `/` et `/`, qui symbolisent une expression régulière en JavaScript.

On peut alors employer les caractères spéciaux utilisables dans une expression régulière, comme `.`, `+`, `?`, `*`, `(` et `)`, etc. Le tableau 15-2 récapitule certaines de ces valeurs.

Tableau 15–2 Caractères spéciaux dans une expression régulière

Caractères	Signification
.	Indique un caractère quelconque.
+	Indique une répétition de 1 à <i>n</i> fois du caractère ou du groupe de caractères qui précède.
?	Indique une répétition de 0 à 1 fois du caractère ou du groupe de caractères qui précède.
*	Indique une répétition de 0 à <i>n</i> fois du caractère ou du groupe de caractères qui précède.
()	Permet de regrouper un ensemble de caractères qui formeront un groupe, sur lequel pourra s'appliquer un des caractères de répétition précédents.
	Donne le choix entre ce qui est avant le signe et ce qui est après celui-ci.
[]	Permet de définir une liste de caractères autorisés, indiqués entre [et]. Si le caractère ^ suit le premier [, cela signifie qu'il convient d'exclure les caractères qui suivent (au lieu de les autoriser). Par exemple, [^abc] signifie qu'il faut interdire les caractères a, b et c. Si le caractère - est indiqué dans la liste, il permet de définir une plage de valeurs. Par exemple, [a-z] permet de définir tous les caractères de a à z, en minuscules.
^	Indique le début d'une chaîne de caractères.
\$	Indique la fin d'une chaîne de caractères.
\b	Indique un début ou une fin de mot.
\B	Inverse de \b (tout sauf un début ou une fin de mot). Cela représente les caractères au milieu d'un mot, en excluant le caractère du début et celui de la fin.
\d	Indique un chiffre de 0 à 9. Synonyme de [0-9].
\D	Inverse de \d (tout sauf un chiffre).
\s	Indique un « caractère blanc » (espace, retour chariot, tabulation, saut de ligne, saut de page).
\S	Inverse de \s (tout sauf un caractère blanc).
\w	Indique un caractère alphanumérique (y compris le caractère _). Synonyme de [a-zA-Z0-9_].
\r	Indique un retour chariot.
\n	Indique un saut de ligne.

Les URL qui correspondent à l'expression régulière indiquée dans l'URL seront considérées comme faisant partie de la route, et la fonction de callback associée à cette dernière sera donc activée.

Par exemple, écrivons la route suivante dans `app.js`.

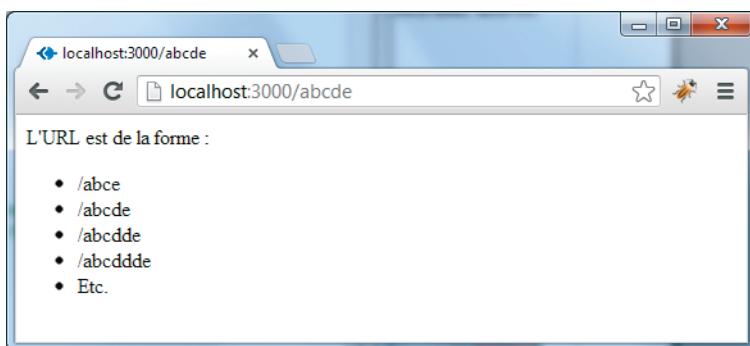
Route définie dans app.js

```
app.get(/abcd*e/, function(req, res) {
  var html = "<p>L'URL est de la forme :</p>";
  html += "<ul>";
  html += "<li>/abce</li>";
  html += "<li>/abcde</li>";
  html += "<li>/abcdde</li>";
  html += "<li>/abcddde</li>";
  html += "<li>Etc.</li>";
  html += "</ul>";
  res.end(html);
});
```

Elle permet de définir une URL contenant "abc" suivie par un nombre quelconque de "d", puis suivie de "e". Par exemple, <http://localhost:3000/abcde>.

Figure 15–9

Formes de l'URL qui déclenche la route.



Remplaçons la route précédente par celle-ci.

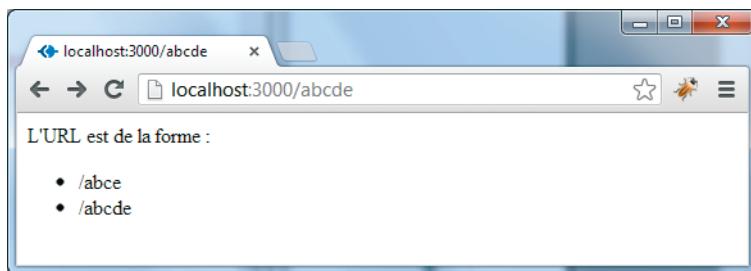
Route définie dans app.js

```
app.get(/abcd?e/, function(req, res) {
  var html = "<p>L'URL est de la forme :</p>";
  html += "<ul>";
  html += "<li>/abce</li>";
  html += "<li>/abcde</li>";
  html += "</ul>";
  res.end(html);
});
```

Cette route permet de définir une URL contenant "abc", suivie de 0 ou 1 caractère "d", puis suivie de "e". Par exemple, <http://localhost:3000/abcde>.

Figure 15–10

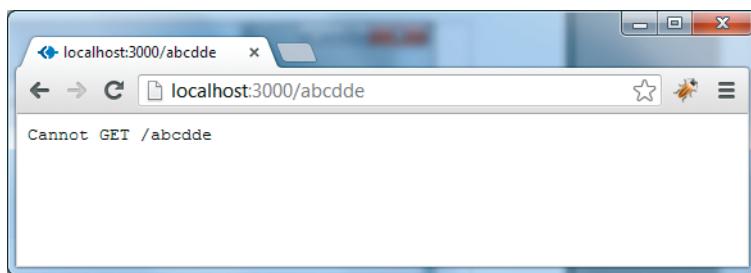
Formes de l'URL qui déclenche la route.



Si vous introduisez deux caractères "d" à la suite, l'URL ne sera pas reconnue et vous obtiendrez une erreur.

Figure 15–11

Affichage d'un message d'erreur si route inconnue



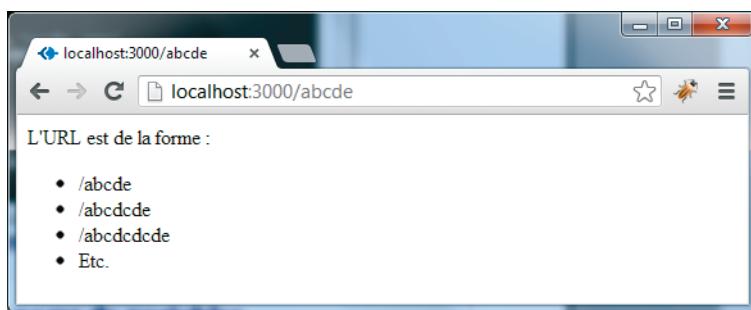
Enfin, testons le regroupement de caractères. Remplaçons la route précédente par celle-ci.

Route définie dans app.js

```
app.get(/ab(cd)+e/, function(req, res) {
  var html = "<p>L'URL est de la forme :</p>";
  html += "<ul>";
  html += "<li>/abcde</li>";
  html += "<li>/abcdcde</li>";
  html += "<li>/abcdcdcde</li>";
  html += "<li>Etc.</li>";
  html += "</ul>";
  res.end(html);
});
```

Cette route permet de définir une URL contenant "ab", suivie de 1 à n groupes de caractères "cd", puis suivie de "e". Par exemple, <http://localhost:3000/abcde>.

Figure 15–12
Formes de l'URL
qui déclenchent la route.



URL définie en utilisant des noms de variables

Une particularité des routes définies avec Express permet d'utiliser des noms de variables dans l'écriture de la chaîne de caractères représentant l'URL. Cette faculté n'est disponible que pour les URL sous forme de chaînes de caractères, mais pas pour celles exprimées au moyen d'expressions régulières.

La variable indiquée dans la définition de la route doit être préfixée du caractère `:` qui permet d'introduire le nom de la variable.

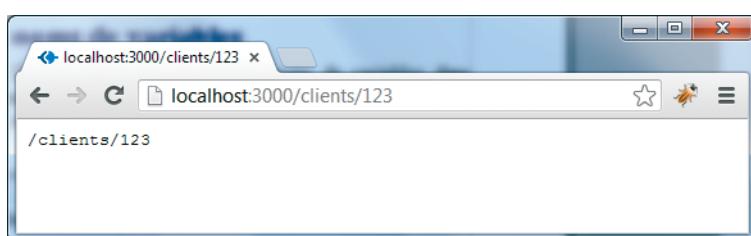
Un exemple de route sous cette forme serait par exemple celui où on indique la variable `:id` dans la chaîne définissant l'URL. Ainsi, pour indiquer des URL de la forme `/clients/:id`, on écrit :

Créer une route utilisant la variable `:id`

```
app.get("/clients/:id", function(req, res) {
  res.end(req.url);
});
```

On affiche ici l'URL utilisée dans la requête. Par exemple, pour l'URL `http://localhost:3000/123`, on obtient :

Figure 15–13
Utilisation d'un nom
de variable dans l'URL



L'intérêt des variables est de pouvoir récupérer leur valeur dans le code de la fonction de callback. La valeur de la variable `:id` est disponible dans l'objet `req.params.id`. Toute autre variable indiquée dans l'URL sera accessible de la même façon, en tant que propriété de l'objet `req.params`.

Par exemple, pour afficher la valeur de la variable `:id` dans la page :

Afficher la valeur de la variable `:id`

```
app.get("/clients/:id", function(req, res) {  
  res.end("Valeur de :id (dans req.params.id) = " + req.params.id);  
});
```

Figure 15-14

Récupération dans `req.params` de la variable transmise dans l'URL



Utiliser plusieurs noms de variables dans l'URL

Plusieurs noms de variables peuvent se trouver dans la définition de l'URL. Par exemple, on peut avoir une URL utilisant le nom et le prénom, qui seront ensuite affichés dans la page.

Utiliser les variables `:nom` et `:prenom` dans l'URL

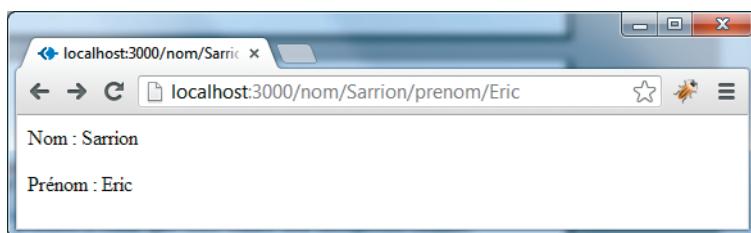
```
app.get("/nom/:nom/prenom/:prenom", function(req, res) {  
  res.setHeader("Content-Type", "text/html; charset=utf-8");  
  var html = "<p>Nom : " + req.params.nom + "</p>";  
  html += "<p>Prénom : " + req.params.prenom + "</p>";  
  res.end(html);  
});
```

On définit l'URL en utilisant les variables `:nom` et `:prenom`. Ces variables sont ensuite accessibles par `req.params.nom` et `req.params.prenom`.

Afin que les balises HTML soient correctement interprétées, on positionne l'en-tête "Content-Type" à "text/html". On indique le charset de la page affichée ("utf-8"), car des caractères accentués sont présents à l'affichage ("Prénom").

Figure 15–15

Utilisation de plusieurs variables dans les URL



Lorsque plusieurs noms de variables sont présents dans l'URL, le caractère séparateur traditionnel est "/", comme dans l'URL de la figure 15–15. Toutefois, on peut également utiliser d'autres caractères tels que "-", ".", ou ",". Cela permet, par exemple, d'écrire des URL de la forme :

- http://localhost/Sarrion-Eric
- http://localhost/fichier.doc
- http://localhost/JavaScript,Node,Ruby
- etc.

Noms de variables optionnels

On souhaite, par exemple, que la variable `:id` soit optionnelle dans l'URL. On pourra ainsi l'indiquer ou pas dans l'URL entrée dans la barre d'adresses du navigateur. Les routes de la forme `GET /clients/:id` et `GET /clients` seront donc valides.

Express permet de préciser ceci en indiquant le caractère "?" après le nom de la variable. Par exemple, indiquons que la variable `:id` est optionnelle :

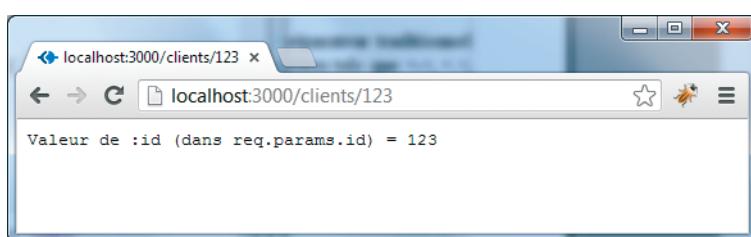
Indiquer que la variable `:id` est optionnelle

```
app.get("/clients/:id?", function(req, res) {
  res.end("Valeur de :id (dans req.params.id) = " + req.params.id);
});
```

Si on indique la valeur de `:id` (ici, 123) dans l'URL :

Figure 15–16

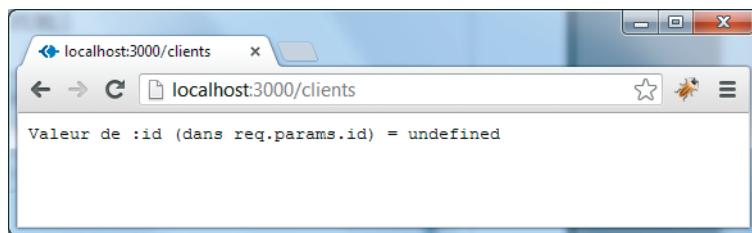
Utilisation d'un nom de variable facultatif (`:id` est présent)



Si on ne l'indique pas :

Figure 15-17

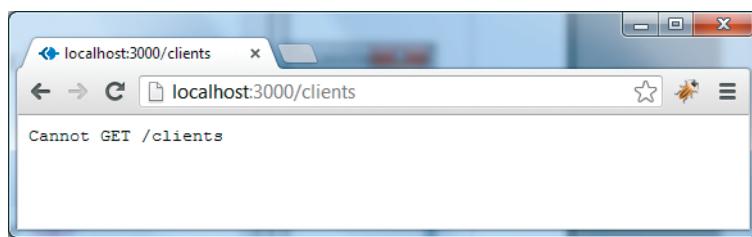
Utilisation d'un nom de variable facultatif (:id est absent)



La route existe et la page est affichée, tandis que la valeur récupérée pour la variable `:id` dans le programme est `undefined`. Sans indiquer le "?" dans la définition de la route, on obtient une erreur de routage, car la variable `:id` est devenue obligatoire.

Figure 15-18

Utilisation d'une route non définie



Vérifier les valeurs possibles d'un nom de variable

Pour l'instant, la seule contrainte que l'on sait vérifier sur un nom de variable consiste à s'assurer de sa présence (en utilisant le caractère "?" qui suit le nom de la variable dans l'écriture de la route, cf. section précédente).

Il est possible d'effectuer des vérifications plus spécifiques, telles que s'assurer que les valeurs sont entières, comprises entre telle valeur et telle autre, etc. On utilise pour cela la méthode `app.param()` fournie par Express. Cette méthode et son utilisation sont décrites dans le chapitre 17, « Objets app, req et res utilisés par Express », section « Gérer le format des variables dans les URL ».

Ordre de priorité des routes

Lors de l'écriture des routes, celles-ci se cumulent dans le fichier `app.js`. Il est alors possible qu'une URL utilisée puisse correspondre à plusieurs routes. Dans ce cas, Express prend la convention d'utiliser la première route trouvée satisfaisante, pour envoyer la réponse au navigateur.

Supposons que les routes soient définies comme ceci. Deux routes peuvent correspondre à l'URL `http://localhost:3000/abcde`.

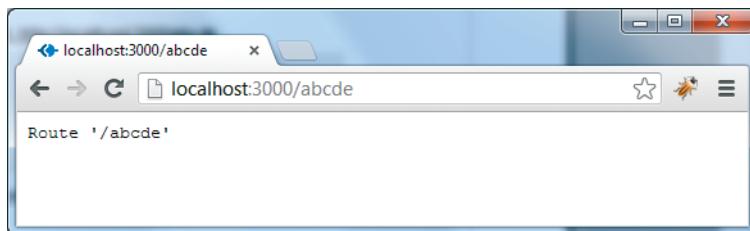
Définitions de deux routes accessibles pour l'URL `http://localhost:3000/abcde`

```
app.get("/abcde", function(req, res) {  
    res.end("Route '/abcde'");  
});  
  
app.get(/abcde/, function(req, res) {  
    res.end("Route /abcde/");  
});
```

La première route sera accessible uniquement à l'URL `http://localhost:3000/abcde`, tandis que la seconde sera accessible à toute URL comportant la chaîne "`abcde`", donc par exemple pour l'URL `http://localhost:3000/abcde`. Les deux routes permettent donc de récupérer l'URL `http://localhost:3000/abcde`.

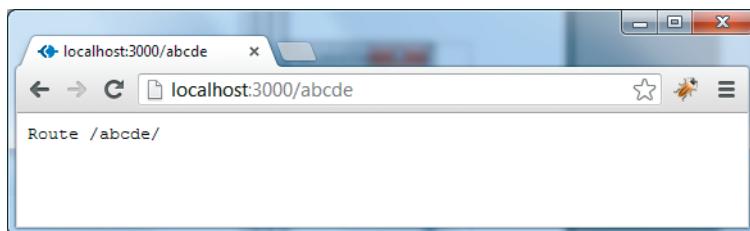
Mais si vous exécutez ce programme, vous verrez que seule la première route écrite dans le fichier sera utilisée par Express. Ainsi, en conservant l'ordre d'écriture des routes, il s'affiche :

Figure 15-19
Priorité des routes :
la première qui correspond
est prise en compte.



Ce qui montre que seule la première route a été activée. Mais si on inverse l'ordre des deux routes dans le fichier, on obtient alors :

Figure 15-20
Priorité des routes :
la première qui correspond
est prise en compte.



À nouveau, c'est toujours la première route qui satisfait l'URL qui est prise en compte. Maintenant, supposons que l'on conserve les routes dans le dernier ordre indiqué (`/abcde/`, puis `"/abcde"`) mais on souhaite que, si l'URL `"/abcde"` est utilisée, la seconde route soit utilisée plutôt que la première. Comment procéder ? Il suffit d'utiliser le paramètre `next` du middleware.

Utiliser le paramètre `next` dans le middleware pour accéder à la seconde route

```
app.get('/abcde/', function(req, res, next) {
  if (req.url == '/abcde') next();
  res.end("Route /abcde/");
});

app.get('/abcde', function(req, res) {
  res.end("Route '/abcde'");
});
```

Si l'URL est `"/abcde"`, on appelle la méthode `next()` qui enchaîne vers le middleware suivant. Finalement, la seconde route n'est activée qu'à partir de la première route, au moyen de `next()`.

Pour plus d'explications sur l'enchaînement des routes et des middlewares, consultez la section « Bien utiliser les middlewares », située en fin de chapitre.

Indiquer une route qui récupère les cas d'erreur

Il peut arriver que l'on se trompe en tapant une URL de notre site, ce qui conduit à l'erreur 404 indiquant que la page n'est pas trouvée. On peut alors créer une route qui récupère tous ces cas d'erreur. Si on ne le fait pas, le mécanisme standard implémenté dans Express se met alors en action et affiche son propre message d'erreur.

Pour cela, il suffit que la dernière URL écrite dans les routes soit suffisamment générale pour qu'elle corresponde à toutes les formes d'URL susceptibles d'être utilisées. On utilise donc une expression régulière telle que `/.*/` signifiant n'importe quels caractères en nombre quelconque.

Utiliser une route pour les cas d'erreur

```
app.get('/clients', function(req, res) {
  res.end("GET /clients");
});

app.get('/*', function(req, res) {
  res.writeHead(404);
  res.end("Erreur 404 : URL " + req.url + " inconnue");
});
```

La première route correspond à une route normale qui affiche un résultat dans le navigateur. La seconde (et dernière) route est celle qui récupère les cas d'erreur. En effet, si aucune des routes précédentes ne fournit la réponse, celle-ci est obligatoirement activée.

Pour être encore plus générale, la dernière route devrait être associée à tous les types de méthodes HTTP : `GET`, `POST`, `PUT` et `DELETE`. Pour cela, il suffit d'utiliser la méthode `app.all()` au lieu de seulement `app.get()`.

Traiter toutes les méthodes HTTP dans les cas d'erreur

```
app.get("/clients", function(req, res) {
  res.end("GET /clients");
});

app.all(/.*/, function(req, res) {
  res.writeHead(404);
  res.end("Erreur 404 : URL " + req.url + " inconnue");
});
```

On peut aussi utiliser `app.use()` qui correspond à l'activation d'un middleware indépendant de la route utilisée. Comme il est écrit en dernier, il sera activé si la réponse n'a pas encore été envoyée.

Utiliser `app.use()` pour traiter les cas d'erreur

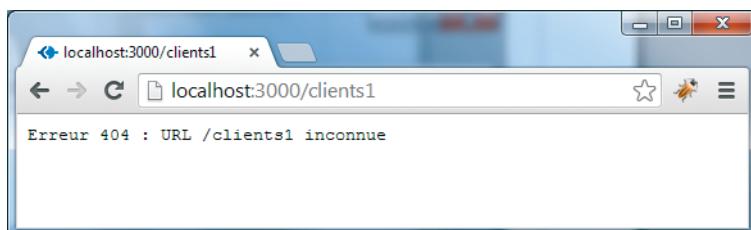
```
app.get("/clients", function(req, res) {
  res.end("GET /clients");
};

app.use(function(req, res) {
  res.writeHead(404);
  res.end("Erreur 404 : URL " + req.url + " inconnue");
});
```

Par exemple, en utilisant la route `GET /clients1` qui ne correspond pas à une URL connue, la route par défaut s'active.

Figure 15-21

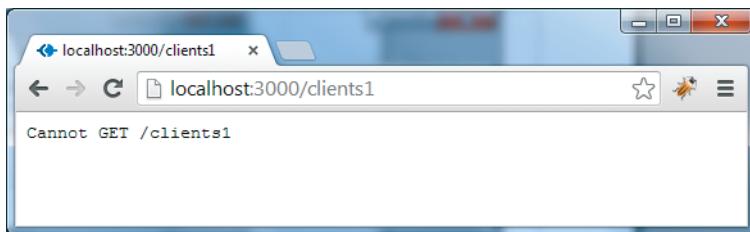
Affichage de la route par défaut définie dans le programme



Si on ne définit pas une route par défaut, le message d'erreur interne à Express s'affiche.

Figure 15–22

Affichage de la route par défaut définie dans Express



Cette fonctionnalité sera utilisée dans les cas où l'on souhaite rendre plus présentables les messages d'erreur affichés par Express. Mais, dans tous les cas, un message d'erreur sera affiché si on accède à une ressource non trouvée.

Organiser l'écriture des routes en créant des modules séparés

Les routes que nous avons écrites sont, pour l'instant, regroupées au sein du fichier principal de notre application, à savoir le fichier `app.js`. Cette façon de procéder peut convenir si le nombre de routes est peu élevé, ce qui est le cas pour le moment. Comment organiser notre code dans le cas où le nombre de routes devient plus conséquent ?

Nous nous inspirons ici de la technique utilisée par Express pour créer les routes par défaut de l'application générée précédemment par celui-ci.

L'application de base créée par Express comporte un répertoire `routes` qui contient les modules `index.js` et `users.js`. Ces deux modules correspondent au middleware associé à la route (c'est-à-dire la fonction de callback exécutée lorsque la route est activée), et sont chargés dans le module `app.js` au moyen des instructions `require()` suivantes.

Chargement des modules associés aux traitements des routes (dans `app.js`)

```
var routes = require('./routes');           // charge le module ./routes/index.js
var user = require('./routes/user');         // charge le module ./routes/user.js
```

La première instruction `require()` ne fait pas mention du nom du module mais seulement du répertoire dans lequel il se trouve, car par défaut c'est le module `index.js` qui est dans ce cas chargé.

Regardons le contenu de ces deux modules.

Fichier index.js (dans le répertoire routes)

```
exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

Fichier user.js (dans le répertoire routes)

```
exports.list = function(req, res){
  res.send("respond with a resource");
};
```

Chacun de ces modules définit un middleware qui envoie la réponse lorsque la route est activée. Les routes sont définies dans le fichier `app.js` de la façon suivante.

Déclaration des routes dans app.js

```
var routes = require("./routes");           // charge le module ./routes/index.js
var user = require("./routes/user");        // charge le module ./routes/user.js
app.get("/", routes.index);
app.get("/users", user.list);
```

Le premier appel à `app.get("/", routes.index)` associe la route `GET /` à la fonction de callback représentée par la fonction `index()` du module `index.js`.

Le second appel à `app.get("/users", user.list)` associe la route `GET /users` à la fonction de callback représentée par la fonction `list()` du module `user.js`.

En se basant sur ce principe, on peut définir autant de modules que nécessaire pour effectuer le traitement des routes.

Pour aller plus loin que ce qui est proposé par Express, on peut souhaiter ne plus inclure les définitions des routes dans le fichier `app.js`. Créons un fichier `routes.js` situé dans le répertoire principal de l'application (au même niveau que `app.js`). Ce fichier va contenir la définition des routes.

Fichier routes.js (dans le répertoire principal de l'application)

```
var routes = require("./routes/");
var user = require("./routes/user");

module.exports = function(app) {
  app.get("/", routes.index);
  app.get("/users", user.list);
}
```

Du fait qu'il existe le fichier `routes.js` et également le répertoire `routes`, on précise `"../routes/"` (terminé par un `"/"`) dans l'instruction `require()` sinon il y aurait ambiguïté entre les deux. On pourrait aussi écrire `require("./routes/index.js")` pour montrer que c'est ce fichier `index.js` qui est chargé.

La fonction utilisée dans le fichier `routes.js` est déclarée comme fonction principale du module.

Le fichier `routes.js` est chargé dans le module `app.js`. La variable `app` lui est transmise afin qu'il puisse y accéder.

Fichier app.js

```
/**  
 * Module dependencies  
 */  
  
var express = require('express');  
var http = require('http');  
var path = require('path');  
var util = require('util');  
  
var app = express();  
var routes = require("./routes.js");  
routes(app);  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.favicon());  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.methodOverride());  
app.use(app.router);  
app.use(express.static(path.join(__dirname, 'public')));  
app.use(express.static(path.join(__dirname, 'forms')));  
  
// development only  
if ('development' == app.get('env')) {  
    app.use(express.errorHandler());  
}  
  
http.createServer(app).listen(app.get('port'), function(){  
    console.log('Express server listening on port ' + app.get('port'));  
});
```

Remarquons que la variable `routes` est ici inutile car elle ne sert que pour appeler la fonction principale du module en lui transmettant le paramètre `app`. On peut donc remplacer les deux lignes :

```
var routes = require("./routes.js");
routes(app);
```

Par celle-ci :

```
require("./routes.js")(app);
```

Vérifions que les routes `GET /` et `GET /users` fonctionnent toujours.

Figure 15–23
Utilisation de la route `GET /`

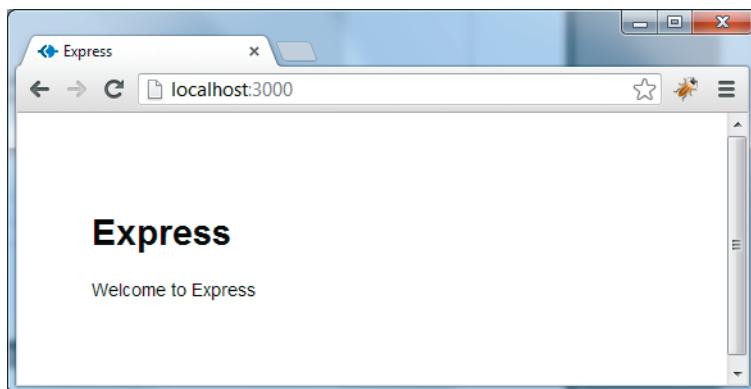
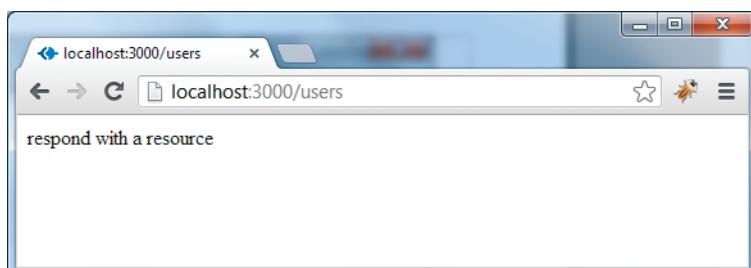


Figure 15–24
Utilisation de la route
`GET /users`



Organiser l'écriture des routes en utilisant REST

Lorsqu'on souhaite utiliser REST (voir la section « Architecture REST » dans ce chapitre), il est possible de diminuer le nombre de lignes à écrire (concernant les routes) en utilisant un module externe qui simplifie la tâche, à savoir le module

`express-resource`. Il permet de créer automatiquement les routes associées à l'accès à une ressource, celle-ci pouvant être un objet ou un ensemble d'objets, manipulés par notre application (par exemple, des clients, des voitures, etc.).

Installons ce module au moyen de la commande `npm install express-resource`, tapée depuis le répertoire principal de notre application Express (ici, `testexpress`).

Figure 15–25

Installation du module
`express-resource`

```
C:\Users\Eric\Documents\Node.js\testexpress>npm install express-resource
npm WARN package.json application-name@0.0.1 No README.md file found
npm http GET https://registry.npmjs.org/express-resource
npm http 200 https://registry.npmjs.org/express-resource
npm http GET https://registry.npmjs.org/express-resource/-/express-resource-0.tgz
npm http 200 https://registry.npmjs.org/express-resource/-/express-resource-0.tgz
npm http GET https://registry.npmjs.org/lingo
npm http GET https://registry.npmjs.org/methods/0.0.1
npm http GET https://registry.npmjs.org/debug
npm http 304 https://registry.npmjs.org/methods/0.0.1
npm http 304 https://registry.npmjs.org/debug
npm WARN package.json methods@0.0.1 No README.md file found!
npm http 200 https://registry.npmjs.org/lingo
npm http GET https://registry.npmjs.org/lingo/-/lingo-0.0.5.tgz
npm http 200 https://registry.npmjs.org/lingo/-/lingo-0.0.5.tgz
express-resource@0.0.1 node_modules\express-resource
  └── methods@0.0.1
    ├── debug@0.7.2
    └── lingo@0.0.5

C:\Users\Eric\Documents\Node.js\testexpress>
```

L'installation de ce module permet de gérer les routes REST définies dans le tableau 15–3. Chacune des routes est associée à une méthode portant un nom défini par le module `express-resource`.

Tableau 15–3 Routes créées par `express-resource`

Route	Méthode	Signification
<code>GET /clients</code>	<code>index</code>	Affiche la liste des clients.
<code>GET /clients/new</code>	<code>new</code>	Affiche le formulaire de création d'un client.
<code>POST /clients</code>	<code>create</code>	Crée le client.
<code>GET /clients/:id</code>	<code>show</code>	Affiche le client représenté par son id (sans permettre sa modification).
<code>GET /clients/:id/edit</code>	<code>edit</code>	Affiche le client représenté par son id (en permettant sa modification).
<code>PUT /clients/:id</code>	<code>update</code>	Met à jour le client représenté par son id.
<code>DELETE /clients/:id</code>	<code>destroy</code>	Supprime le client représenté par son id.

Grâce à ces routes, on a ainsi accès à l'ensemble des opérations que l'on peut souhaiter effectuer sur un objet `client`: **CREATE**, **READ**, **UPDATE** et **DELETE** (CRUD en abrégé). Notez que lorsqu'on souhaite visualiser un client en permettant sa modification ou non, la route est presque la même dans les deux cas : on ajoute simplement `/edit` à la fin de l'URL de la route pour indiquer la possibilité de modification.

Remarquez que les routes qui mettent à jour le modèle de données (insertion, modification ou suppression) ne sont pas accessibles par la méthode `GET`, afin de ne pas autoriser ces actions depuis un navigateur non autorisé. Les seules routes accessibles à la méthode `GET` sont celles qui affichent des informations sans pouvoir les modifier.

De plus, le nom de la méthode HTTP utilisée par `express-resource` symbolise bien le traitement que l'on souhaite effectuer : `GET` pour une lecture de un ou plusieurs clients, `POST` pour créer un client, `PUT` pour mettre à jour le client et `DELETE` pour le supprimer.

En lisant le tableau 15-3, on peut en déduire (sur la première ligne de celui-ci) que si l'on entre l'URL `http://localhost:3000/clients`, cela activera une méthode nommée `index`. Cette méthode devra être écrite par nous-mêmes et réalisera le traitement consistant à afficher la liste des clients. Il en est de même pour les autres routes de la liste, mais certaines d'entre elles ne seront accessibles que depuis un formulaire, afin d'utiliser les méthodes `POST`, `PUT` ou `DELETE` qui ne sont pas accessibles autrement.

Activer les routes REST dans le fichier app.js

Le fichier principal de notre application Express (fichier `app.js`) doit être modifié pour prendre en compte ces nouvelles routes associées aux clients. Si l'on désire gérer d'autres types de ressources, il suffira de les gérer comme l'on gère les clients suivants.

Fichier `app.js` prenant en compte les routes REST définies par `express-resource`

```
/**  
 * Module dependencies  
 */  
  
var express = require('express');  
var http = require('http');  
var path = require('path');  
var util = require('util');  
var resource = require('express-resource');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);
```

```
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public'))));
app.use(express.static(path.join(__dirname, 'forms'))));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.resource("clients", require("./clients.js"));

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Nous insérons le module `express-resource`, ajouté obligatoirement avant de créer l'application Express `app` (sinon rien ne fonctionne). Puis nous utilisons la nouvelle méthode `app.resource()`, définie par `express-resource` sur l'objet `app`. La méthode `app.resource()` prend deux paramètres.

- Le premier paramètre correspond au nom des objets manipulés, ici des clients. Il est usuel de mettre ce nom au pluriel, ceci symbolisant le fait qu'on gère un ensemble d'objets. Ce nom sera celui utilisé dans les URL des routes, qui commenceront obligatoirement par ce nom, par exemple `http://localhost:3000/clients` (ceci quelle que soit la méthode HTTP utilisée).
- Le second paramètre correspond à un objet contenant les méthodes appelées par `express-resource` lors du traitement des routes. Ce sont les méthodes précédemment listées dans le tableau : `index`, `new`, `create`, `show`, `edit`, `update` et `destroy`. Dans notre cas, ces méthodes sont définies dans un module externe `clients.js` qui exporte leurs valeurs via l'objet `module.exports`.

Examinons maintenant le contenu du module `clients.js`, qui correspond au traitement effectué sur chacune des routes.

Écrire le traitement associé aux routes

Le traitement effectué sur chacune des routes doit être inscrit dans un objet JavaScript transmis en second paramètre de la méthode `app.resource()`. Si on utilise un nouveau module contenant le traitement des routes, il suffit d'écrire ces méthodes en

tant que propriétés de l'objet `module.exports`. L'instruction `require(fichier)` utilisée dans `app.js` retournera donc un objet contenant les méthodes attendues : `index`, `new`, `create`, `show`, `edit`, `update` et `destroy`.

Fichier clients.js (dans testexpress)

```
exports.index = function(req, res) {
    console.log(req.params);
    res.setHeader("Content-Type", "text/html; charset=utf-8");
    var html = "<p>Route GET /clients</p>";
    html += "<p>Méthode index - Afficher la liste des clients</p>";
    res.end(html);
};

exports.new = function(req, res) {
    console.log(req.params);
    res.setHeader("Content-Type", "text/html; charset=utf-8");
    var html = "<p>Route GET /clients/new</p>";
    html += "<p>Méthode new - Afficher le formulaire de création
        d'un nouveau client</p>";
    res.end(html);
};

exports.create = function(req, res) {
    console.log(req.params);
    res.setHeader("Content-Type", "text/html; charset=utf-8");
    var html = "<p>Route POST /clients</p>";
    html += "<p>Méthode create - Créer un nouveau client</p>";
    res.end(html);
};

exports.show = function(req, res) {
    console.log(req.params);
    res.setHeader("Content-Type", "text/html; charset=utf-8");
    var html = "<p>Route GET /clients/:id</p>";
    html += "<p>Méthode show - Afficher le client dont l'id est "
        + req.params.client + "</p>";
    res.end(html);
};

exports.edit = function(req, res) {
    console.log(req.params);
    res.setHeader("Content-Type", "text/html; charset=utf-8");
    var html = "<p>Route GET /clients/:id/edit</p>";
    html += "<p>Méthode edit - Afficher le formulaire de modification du client
        dont l'id est " + req.params.client + "</p>";
    res.end(html);
};
```

```
exports.update = function(req, res) {
  console.log(req.params);
  res.setHeader("Content-Type", "text/html; charset=utf-8");
  var html = "<p>Route PUT /clients/:id</p>";
  html += "<p>Méthode update - Mettre à jour le client dont l'id est "
    + req.params.client + "</p>";
  res.end(html);
};

exports.destroy = function(req, res) {
  console.log(req.params);
  res.setHeader("Content-Type", "text/html; charset=utf-8");
  var html = "<p>Route DELETE /clients/:id</p>";
  html += "<p>Méthode destroy - Supprimer le client dont l'id est "
    + req.params.client + "</p>";
  res.end(html);
};
```

Nous avons implémenté dans l'objet `module.exports` (également nommé `exports`) les méthodes `index`, `new`, `create`, `show`, `edit`, `update` et `destroy`. Chacune de ces méthodes correspond à un middleware permettant l'envoi de la réponse. Ces middlewares prennent donc en paramètres les objets `req` et `res`, présents dans les middlewares.

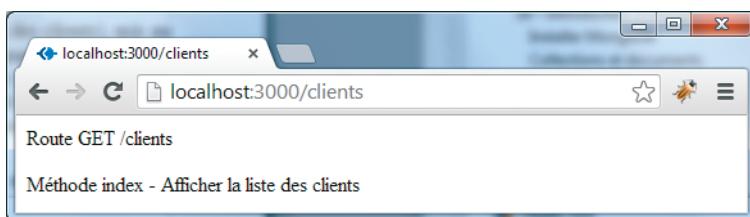
Nous affichons dans chacune de ces méthodes le contenu de l'objet `req.params`. Il contient les champs `"client"` et `"format"`.

- Le champ `"client"` correspond au nom des objets gérés par la route (ici, des clients), mis au singulier. Il suffit donc d'enlever la lettre `"s"` du premier argument indiqué dans `app.resource()` pour obtenir le nom de ce champ. Si cet argument n'est pas mis au pluriel (bien que ce soit recommandé), le nom du champ spécifié dans `req.params` est alors le même que celui mis en premier argument de `app.resource()`. La valeur récupérée dans `req.params.client` sera égale à la valeur transmise dans la variable `:id` dans l'URL.
- Le champ `"format"` correspond à une extension éventuelle indiquée en fin d'URL, par exemple `".txt"`, `".pdf"` ou `".json"` (liste non exhaustive). La valeur récupérée dans `req.params.format` sera donc `"txt"`, `"pdf"` ou `"json"` selon ce qui a été indiqué dans l'URL. Un traitement différent pourra alors être effectué selon le format indiqué dans l'URL. Pour plus d'informations sur ce composant `:format`, reportez-vous à la section suivante.

Activons certaines des URL fournies par REST afin de voir les résultats affichés.

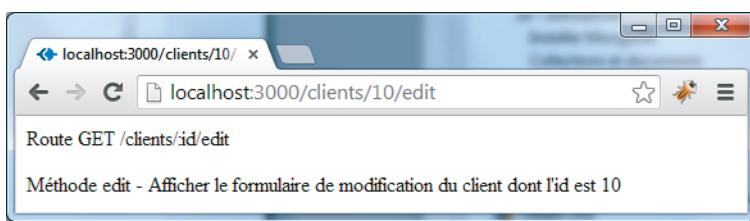
Pour la route `GET /clients` :

Figure 15–26
Utilisation de la route
`GET /clients`



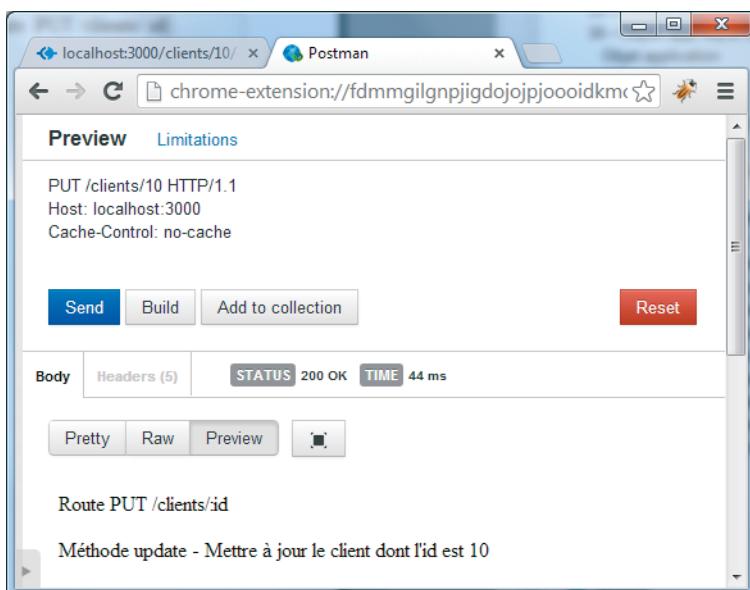
Pour la route `GET /clients/10/edit` :

Figure 15–27
Utilisation de la route
`GET /clients/10/edit`



Pour les routes utilisant les méthodes `POST`, `PUT` et `DELETE`, il faut utiliser un formulaire HTML ou passer par l'outil Postman de Chrome. Par exemple, pour tester la route `PUT /clients/:id` :

Figure 15–28
Utilisation de la requête PUT
avec Postman



Remarquons que si une ou plusieurs méthodes ne sont pas définies (parmi `index`, `new`, `create`, `show`, `edit`, `update` ou `destroy`), la route correspondante n'existera pas. Si l'on essaie d'y accéder tout de même, Express nous retourne un message d'erreur indiquant que la route n'existe pas. Il est donc tout à fait possible de définir uniquement certaines méthodes et pas les autres.

Créer des services web en utilisant le composant :format dans les routes

Le composant (ou variable) `:format` est défini par `express-resource` dans les URL des routes, en tant que composant optionnel. S'il est présent, il est indiqué en fin d'URL et précédé de `".`. Par exemple, on pourrait utiliser les URL suivantes.

- `http://localhost:3000/clients.xml` : permet d'accéder à la liste de clients, au format XML.
- `http://localhost:3000/clients.json` : permet d'accéder à la liste de clients, au format JSON.
- `http://localhost:3000/clients` : sans extension, permet d'accéder à la liste de clients, au format HTML.

Le composant `:format` sera utilisé lorsqu'on souhaite créer des services web qui retournent des informations sous forme XML ou JSON. La même forme d'URL pourra aussi retourner un résultat HTML, il suffira pour cela de ne pas indiquer d'extension dans l'URL utilisée.

Réécrivons la méthode `index` précédente afin qu'elle prenne en compte les formats HTML, XML et JSON.

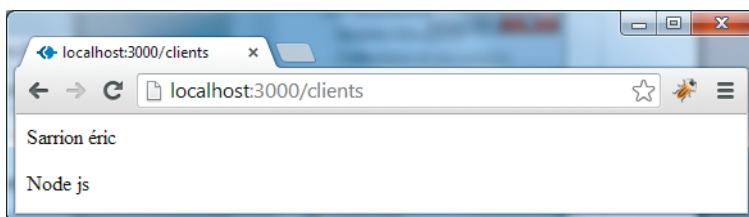
Méthode index renvoyant du HTML, du XML ou du JSON

```
exports.index = function(req, res) {
  if (!req.params.format) {
    // pas de format indiqué : c'est donc du HTML que l'on souhaite renvoyer
    res.setHeader("Content-Type", "text/html; charset=utf-8");
    var html = "<p>Sarrion éric</p><p>Node js</p>";
    res.end(html);
  } else if (req.params.format == "xml") {
    // format XML
    res.setHeader("Content-Type", "text/xml");
    var xml = "<résultat>";
    xml += "<p>Sarrion éric</p><p>Node js</p>";
    xml += "</résultat>";
    res.end(xml);
  } else if (req.params.format == "json") {
    // format JSON
    res.setHeader("Content-Type", "application/json; charset=utf-8");
    var json = "[ { p : 'Sarrion éric' }, { p : 'Node js' } ]";
    res.end(json);
  }
};
```

Nous spécifions pour chaque format de données le type du contenu (HTML, XML ou JSON) et son encodage ("utf-8"). Puis la valeur renvoyée est créée au format demandé et envoyée vers le client au moyen de `res.end()`.

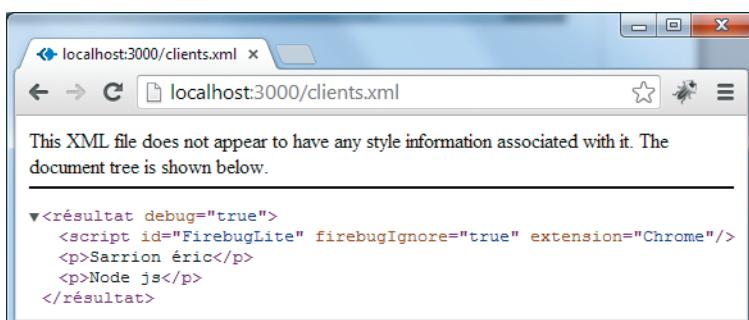
Par exemple, pour un affichage en HTML :

Figure 15–29
Affichage en HTML



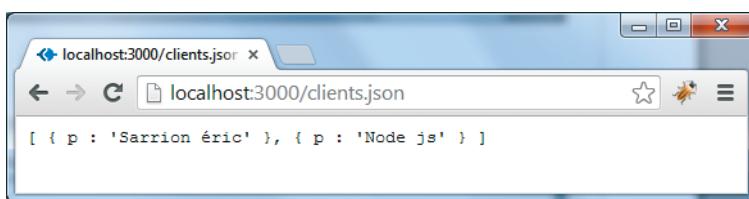
Pour un retour en XML :

Figure 15–30
Affichage sous forme XML



Enfin, pour un retour en JSON :

Figure 15–31
Affichage sous forme JSON



On voit que le format des éléments renvoyés varie selon l'extension fournie dans l'URL. Cette faculté apportée par le composant `:format` d'Express permet de réaliser des services web pour nos applications.

Bien utiliser les middlewares

On a vu que lors de la définition d'une route par `app.get()`, `app.post()`, etc., Express positionnait la route correspondante dans l'objet `app.routes` (voir la section « Objet `app.routes` défini par Express »).

Objet `app.stack` défini par Express

Lors de la création d'un middleware au moyen de `app.use()`, Express l'insère dans l'objet `app.stack`, qui est un tableau d'objets `{ route, handle }`, dans lequel :

- la propriété `route` correspond à l'URL qui active ce middleware, par exemple `"/clients"` (indépendamment de la requête utilisée) ;
- la propriété `handle` correspond à un objet `Function`, qui est une fonction de callback qui sera appelée pour traiter l'URL. Chaque fonction correspond à un middleware.

Affichons le contenu de l'objet `app.stack` lorsque tous les middlewares d'Express ont été positionnés.

Afficher le contenu de `app.stack`

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
, connect = require('connect')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, util = require('util')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.favicon());  
app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));
```

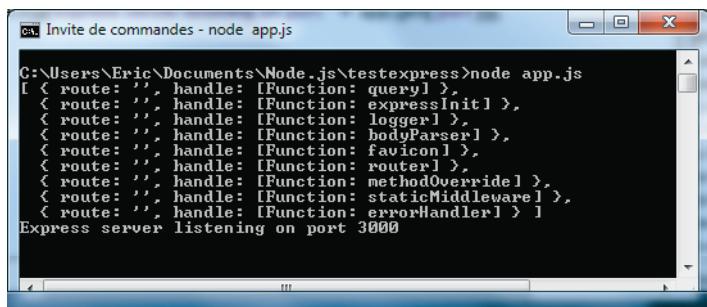
```
// development only
if ('development' == app.get('env')) {
    app.use(express.errorHandler());
}

console.log(util.inspect(app.stack));

http.createServer(app).listen(app.get('port'), function(){
    console.log('Express server listening on port ' + app.get('port'));
});
```

Nous avons simplement ajouté l'instruction `console.log()` qui affiche le contenu de l'objet `app.stack` (à l'aide de la méthode `util.inspect()` du module `util`).

Figure 15–32
Objet `app.stack` par défaut



Les propriétés `route` sont ici positionnées à `""` (chaîne vide) car les méthodes `app.use()` utilisées ne font pas usage d'une URL lors de leur écriture. Les middlewares correspondants sont déclenchés quelle que soit l'URL utilisée.

Ajoutons les deux middlewares `use1` et `use2` associés à l'URL `"/clients"`.

Ajouter deux middlewares associés à l'URL /clients

```
/***
 * Module dependencies
 */

var express = require('express')
, connect = require('connect')
, routes = require('./routes')
, user = require('./routes/user')
, http = require('http')
, util = require('util')
, path = require('path');
```

```

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.favicon());
app.use(app.router);
app.use(express.methodOverride());
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.use("/clients", function use1(req, res, next) {
  console.log("use1");
  next();
  console.log("use1 fin");
});

app.use("/clients", function use2(req, res, next) {
  console.log("use2");
  next();
  console.log("use2 fin");
});

console.log(util.inspect(app.stack));

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});

```

Figure 15-33
Objet app.stack modifié

```

C:\Users\Eric\Documents\Node.js\testexpress>node app.js
[ { route: '/', handle: [Function: query] },
  { route: '/', handle: [Function: expressInit] },
  { route: '/', handle: [Function: logger] },
  { route: '/', handle: [Function: bodyParser] },
  { route: '/', handle: [Function: favicon] },
  { route: '/', handle: [Function: router] },
  { route: '/', handle: [Function: methodOverride] },
  { route: '/', handle: [Function: staticMiddleware] },
  { route: '/clients', handle: [Function: use1] },
  { route: '/clients', handle: [Function: use2] } ]
Express server listening on port 3000

```

Remarquons que les middlewares associés aux routes définies par `app.get()`, `app.post()`, etc., sont insérés dans l'objet `app.routes`, tandis que les middlewares définis par `app.use()` sont insérés dans l'objet `app.stack`.

Définissons la route `GET /clients` et affichons les objets `app.stack` et `app.routes` dans la console du serveur.

Afficher les objets app.stack et app.routes

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
  , connect = require('connect')  
  , routes = require('./routes')  
  , user = require('./routes/user')  
  , http = require('http')  
  , util = require('util')  
  , path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.favicon());  
app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
  app.use(express.errorHandler());  
}  
  
app.use("/clients", function use1(req, res, next) {  
  console.log("use1");  
  next();  
  console.log("use1 fin");  
});  
  
app.use("/clients", function use2(req, res, next) {  
  console.log("use2");  
});
```

```

next();
console.log("use2 fin");
});

app.get("/clients", function getclients1(req, res, next) {
  res.end("GET /clients OK");
});

console.log(util.inspect(app.stack));
console.log(util.inspect(app.routes));

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});

```

Figure 15–34

Objets app.stack et app.routes

```

C:\Users\Eric\Documents\Node.js\testexpress>node app.js
[ { route: '', handle: [Function: query] },
  { route: '', handle: [Function: expressInit] },
  { route: '', handle: [Function: logger] },
  { route: '', handle: [Function: bodyParser] },
  { route: '', handle: [Function: favicon] },
  { route: '', handle: [Function: router] },
  { route: '', handle: [Function: methodOverride] },
  { route: '', handle: [Function: staticMiddleware] },
  { route: '', handle: [Function: errorHandler] },
  { route: '/clients', handle: [Function: use1] },
  { route: '/clients', handle: [Function: use2] } ]
  < get:
    [ { path: '/clients',
        method: 'get',
        callbacks: [Object],
        keys: [],
        regexp: '^/clients$/i' } ]
  Express server listening on port 3000

```

Exécutons maintenant la route `GET /clients` dans un navigateur.**Figure 15–35**Objets app.stack et app.routes
après exécution de la route GET
`/clients`

```

C:\Users\Eric\Documents\Node.js\testexpress>node app.js
[ { route: '', handle: [Function: query] },
  { route: '', handle: [Function: expressInit] },
  { route: '', handle: [Function: logger] },
  { route: '', handle: [Function: bodyParser] },
  { route: '', handle: [Function: favicon] },
  { route: '', handle: [Function: router] },
  { route: '', handle: [Function: methodOverride] },
  { route: '', handle: [Function: staticMiddleware] },
  { route: '', handle: [Function: errorHandler] },
  { route: '/clients', handle: [Function: use1] },
  { route: '/clients', handle: [Function: use2] } ]
  < get:
    [ { path: '/clients',
        method: 'get',
        callbacks: [Object],
        keys: [],
        regexp: '^/clients$/i' } ]
  Express server listening on port 3000

```

On obtient une trace de l'appel de la route sur la console du serveur, mais on ne voit pas l'appel de nos deux middlewares `use1` et `use2` que nous avons pourtant définis sur l'URL `/clients`. Ceci est une conséquence de l'appel du middleware `app.router` tel qu'il est effectué dans notre application.

Nous donnons un aperçu plus complet dans la section suivante.

Influence du middleware app.router

Lors de la création d'une application Express par défaut, le middleware `app.router` est inscrit de façon automatique dans le code de notre programme de base. Ce middleware effectue la gestion des routes et il est obligatoire. Cependant, s'il n'est pas utilisé directement comme ici, Express l'insère automatiquement dès qu'il rencontre la définition d'une route par `app.get()`, `app.post()`, etc. On peut donc supprimer la ligne d'utilisation du middleware `app.router` dans le programme.

Supprimons la ligne correspondante dans le fichier et relançons le programme. On affiche de nouveau l'URL `/clients` dans le navigateur.

Suppression de l'utilisation du middleware app.router

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
, connect = require('connect')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, util = require('util')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.favicon());  
// app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));
```

```
// development only
if ('development' == app.get('env')) {
    app.use(express.errorHandler());
}

app.use("/clients", function use1(req, res, next) {
    console.log("use1");
    next();
    console.log("use1 fin");
});

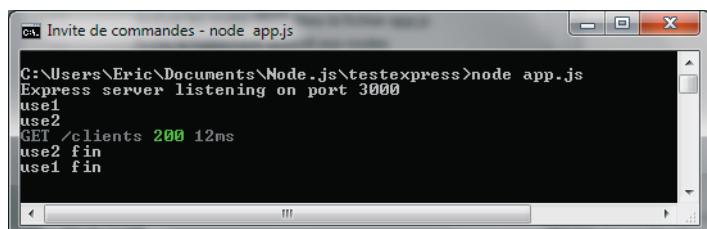
app.use("/clients", function use2(req, res, next) {
    console.log("use2");
    next();
    console.log("use2 fin");
});

app.get("/clients", function getclients1(req, res, next) {
    res.end("GET /clients OK");
});

http.createServer(app).listen(app.get('port'), function(){
    console.log('Express server listening on port ' + app.get('port'));
});
```

Nous avons défini les middlewares `use1` et `use2`, puis la route `GET /clients` retournant la réponse au navigateur, en ayant supprimé l'appel explicite au middleware `app.router`.

Figure 15–36
Les middlewares `use1` et `use2` sont appelés.



On voit maintenant les appels aux middlewares `use1` et `use2`. On obtient le même résultat en plaçant les middlewares `use1` et `use2` avant le middleware `app.router`.

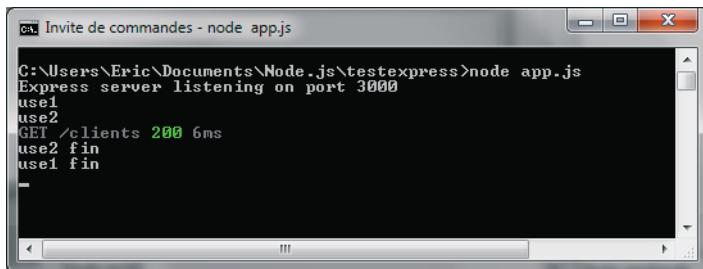
Déclarer les middlewares `use1` et `use2` avant le middleware `app.router`

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
, connect = require('connect')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, util = require('util')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.favicon());  
  
app.use("/clients", function use1(req, res, next) {  
  console.log("use1");  
  next();  
  console.log("use1 fin");  
});  
  
app.use("/clients", function use2(req, res, next) {  
  console.log("use2");  
  next();  
  console.log("use2 fin");  
});  
  
app.use(app.router);  
  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
  app.use(express.errorHandler());  
}  
  
app.get("/clients", function getclients1(req, res, next) {  
  res.end("GET /clients OK");  
});
```

```
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Figure 15-37

Les middlewares `use1` et `use2` sont appelés.



En résumé, le middleware `app.router` permet de prioriser (faire passer en premier) les routes définies au moyen de `app.get()`, `app.post()`, `app.put()` et `app.delete()`. Dès que `app.router` est mis en place, les middlewares définis à la suite par `app.use()` ne sont traités qu'après les traitements de ces routes, et à la condition que la réponse n'ait pas encore été rendue par une de ces routes (par `res.end()`). Pour qu'un middleware défini par `app.use()` soit exécuté dans tous les cas, il suffit de le déclarer avant la mise en place du middleware `app.router`. Rappelons que ce middleware `app.router` est mis en place soit par l'appel explicite à `app.use(app.router)`, soit automatiquement par Express dès que l'on rencontre l'appel à `app.get()`, `app.post()`, `app.put()` ou `app.delete()`.

Enchaînement des middlewares avec `next()`

L'exemple précédent montre clairement le rôle de la fonction `next()`, permettant d'enchaîner les middlewares les uns avec les autres. Les middlewares appelés par `next()` sont empilés, puis ils sont dépilerés (dans le même sens qu'une pile). Ainsi, le middleware `use1` finit de s'exécuter après que le middleware `use2` a terminé son exécution.

De plus, si un middleware renvoie une réponse au serveur (par `res.end()`), cela arrête l'enchaînement d'appels des middlewares qui suivent, comme on peut le voir sur l'exemple qui suit.

Arrêter l'enchaînement des middlewares en renvoyant la réponse du serveur

```
/**
 * Module dependencies
 */

var express = require('express')
, connect = require('connect')
, routes = require('./routes')
```

```
, user = require('./routes/user')
, http = require('http')
, util = require('util')
, path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.favicon());
// app.use(app.router);
app.use(express.methodOverride());
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get("/clients", function getclients1(req, res, next) {
  res.end("GET /clients OK");
});

app.use("/clients", function use1(req, res, next) {
  console.log("use1");
  next();
  console.log("use1 fin");
});

app.use("/clients", function use2(req, res, next) {
  console.log("use2");
  next();
  console.log("use2 fin");
});

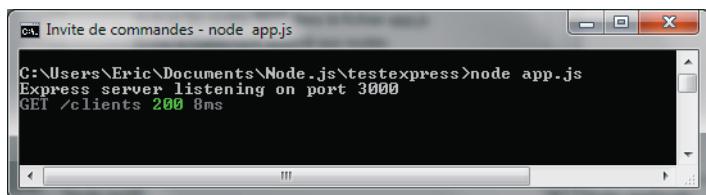
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Le middleware `app.router` est ici mis en commentaires bien que cela n'ait pas d'incidence car il est automatiquement ajouté par Express dès que celui-ci rencontre l'instruction `app.get()` qui suit.

La route `GET /clients` retourne une réponse au navigateur, ce qui empêche les middlewares suivants `use1` et `use2` de continuer à s'exécuter.

Figure 15–38

L'envoi de la réponse empêche les middlewares qui suivent de s'exécuter.



Si on modifie le middleware associé à la route `GET /clients`, en appelant `next()` plutôt que d'envoyer la réponse, Express passe le contrôle au middleware `use1` qui suit.

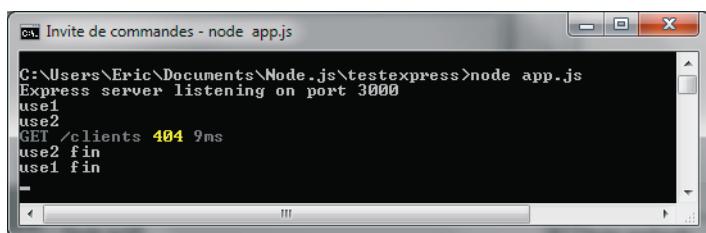
Appeler next() au lieu de renvoyer la réponse

```
app.get("/clients", function getclients1(req, res, next) {
  // res.end("GET /clients OK");
  next();
});
```

Le résultat est maintenant complètement différent.

Figure 15–39

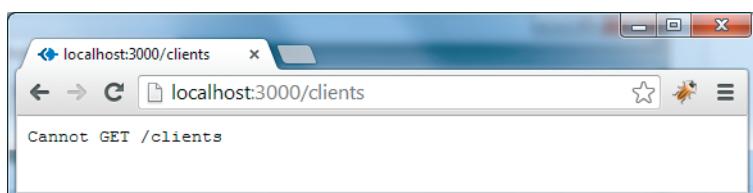
L'utilisation de `next()` active le middleware suivant.



Tous les middlewares sont enchaînés, mais comme aucun d'eux ne retourne une réponse au navigateur, Express affiche son message d'erreur (erreur 404).

Figure 15–40

Aucune réponse n'est envoyée au navigateur : erreur 404 (page non trouvée).



Enfin, il est intéressant de remarquer un dernier point concernant la méthode `next()`. On sait qu'elle permet de passer au middleware suivant, sauf lorsqu'elle est utilisée dans une méthode `app.get()`, `app.post()`, etc. En effet, dans ce cas, elle indique de

passer au middleware associé à la même route qui suit (qui n'est pas forcément le suivant dans le code).

Tableau 15–4 Méthode next()

Méthode	Signification
next()	Indique de passer au middleware suivant. Si la fonction <code>next()</code> est utilisée dans une définition de route (dans <code>app.get()</code> , <code>app.post()</code> , etc.), elle indique alors de passer au middleware suivant défini dans la même route ou une route qui suit. Si ce middleware n'existe pas, passe au middleware suivant défini par <code>app.use()</code> . Il faut noter que n'importe quel middleware (de route ou pas) peut retourner la réponse du serveur, et dans ce cas cela arrête l'enchaînement des middlewares.

Voyons cela sur l'exemple qui suit.

Utiliser next() pour passer au middleware suivant associé à la même route

```
/** 
 * Module dependencies
 */

var express = require('express')
, connect = require('connect')
, routes = require('./routes')
, user = require('./routes/user')
, http = require('http')
, util = require('util')
, path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.favicon());
// app.use(app.router);
app.use(express.methodOverride());
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}
```

```
app.get("/clients", function getclients1(req, res, next) {
  console.log("getclients1");
  next();
  console.log("getclients1 fin");
});

app.use("/clients", function use1(req, res, next) {
  console.log("use1");
  next();
  console.log("use1 fin");
});

app.use("/clients", function use2(req, res, next) {
  console.log("use2");
  next();
  console.log("use2 fin");
});

app.get("/clients", function getclients2(req, res, next) {
  console.log("getclients2");
  next();
  console.log("getclients2 fin");
});

app.get("/clients", function getclients3(req, res, next) {
  console.log("getclients3");
  res.end("getclients3");
  console.log("getclients3 fin");
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Nous avons d'abord défini une route `GET /clients` (fonction `getclients1`) qui effectue l'appel à `next()`. Les deux middlewares `use1` et `use2` qui suivent sont ignorés et la prochaine route `GET /clients` est activée, à travers la fonction `getclients2`. Cette route effectue également un appel à `next()`, et la prochaine route `GET /clients` est activée à travers la fonction `getclients3`. Cette dernière route retourne enfin le résultat.

En supposant maintenant que la dernière route `getclients3` ne renvoie pas la réponse (par `res.end()`), mais effectue plutôt un appel à `next()`, les middlewares `use1` et `use2` redeviennent exécutés car la réponse n'a pas encore été envoyée. Express appelle ces middlewares au cas où l'un d'eux donne une réponse.

Figure 15–41

Utilisation de next() pour enchaîner les middlewares

```
C:\Users\Eric\Documents\Node.js\testexpress>node app.js
Express server listening on port 3000
getclients1
getclients2
getclients3
GET /clients 200 7ms
getclients3 fin
getclients2 fin
getclients1 fin
```

Ne pas effectuer de réponse dans les routes GET

```
/**
 * Module dependencies
 */

var express = require('express')
, connect = require('connect')
, routes = require('./routes')
, user = require('./routes/user')
, http = require('http')
, util = require('util')
, path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.favicon());
// app.use(app.router);
app.use(express.methodOverride());
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get("/clients", function getclients1(req, res, next) {
  console.log("getclients1");
  next();
})
```

```
console.log("getclients1 fin");
});

app.use("/clients", function use1(req, res, next) {
  console.log("use1");
  next();
  console.log("use1 fin");
});

app.use("/clients", function use2(req, res, next) {
  console.log("use2");
  next();
  console.log("use2 fin");
});

app.get("/clients", function getclients2(req, res, next) {
  console.log("getclients2");
  next();
  console.log("getclients2 fin");
});

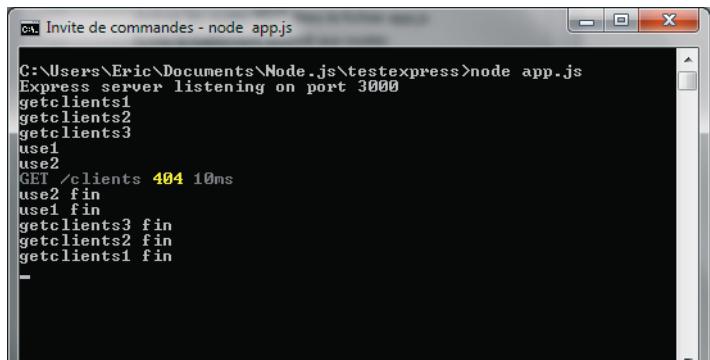
app.get("/clients", function getclients3(req, res, next) {
  console.log("getclients3");
  // res.end("getclients3");
  next();
  console.log("getclients3 fin");
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

L'appel à `next()` a remplacé l'appel à `res.end()` dans la dernière route.

Figure 15–42

Les middlewares associés aux routes sont appelés en premier, puis les middlewares définis par `app.use()` (car aucune réponse n'a été renvoyée au navigateur avant).



Utiliser la fonction next("route")

Dans la section « Utiliser la route en tant que middleware » étudiée en début de chapitre, on a vu qu'il est possible d'affecter plusieurs middlewares en même temps dans une même déclaration de route. Par exemple :

Utiliser plusieurs middlewares dans la même définition de route

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
  , connect = require('connect')  
  , routes = require('./routes')  
  , user = require('./routes/user')  
  , http = require('http')  
  , util = require('util')  
  , path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.favicon());  
app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
  app.use(express.errorHandler());  
}  
  
app.get("/clients", function getclients1(req, res, next) {  
  console.log("getclients1");  
  next();  
  console.log("getclients1 fin");  
}, function getclients2(req, res, next) {  
  console.log("getclients2");  
  next();  
  console.log("getclients2 fin");  
}, function getclients3(req, res, next) {  
  console.log("getclients3");  
});
```

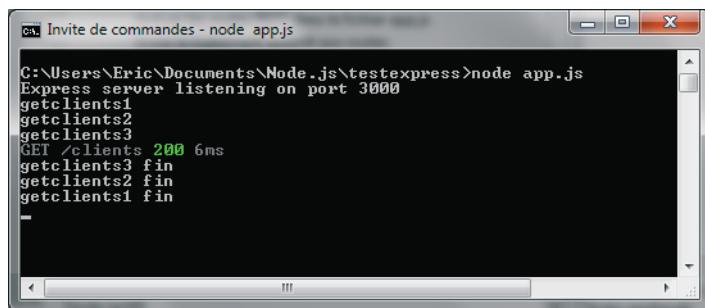
```
res.end("getclients3");
console.log("getclients3 fin");
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Nous avons ici trois middlewares définis dans la route `GET /clients`. Les fonctions `next()` passent au middleware suivant de la même route, ce qui provoque leur enchaînement jusqu'au dernier qui retourne le résultat. On aurait pu aussi écrire trois appels à la méthode `app.get()`, ce qui aurait produit le même résultat.

Figure 15–43

Utilisation de `next()` dans les middlewares d'une même route



Modifions le programme en ajoutant un nouveau middleware pour la route `GET /clients`, mais après les premiers traitements. Par ailleurs, la fonction `getclients3()` effectue `next()` au lieu de retourner la réponse du serveur.

Ajouter un nouvelle route

```
/**
 * Module dependencies
 */

var express = require('express')
, connect = require('connect')
, routes = require('./routes')
, user = require('./routes/user')
, http = require('http')
, util = require('util')
, path = require('path');
```

```
var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.favicon());
app.use(app.router);
app.use(express.methodOverride());
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
    app.use(express.errorHandler());
}

app.get("/clients", function getclients1(req, res, next) {
    console.log("getclients1");
    next();
    console.log("getclients1 fin");
}, function getclients2(req, res, next) {
    console.log("getclients2");
    next();
    console.log("getclients2 fin");
}, function getclients3(req, res, next) {
    console.log("getclients3");
    next();
    console.log("getclients3 fin");
});

app.get("/clients", function getclients4(req, res, next) {
    console.log("getclients4");
    res.end("getclients4");
    console.log("getclients4 fin");
});

http.createServer(app).listen(app.get('port'), function(){
    console.log('Express server listening on port ' + app.get('port'));
});
```

On obtient maintenant le résultat attendu suivant :

Figure 15–44
Enchaînement de middlewares

```
C:\Users\Eric\Documents\Node.js\testexpress>node app.js
Express server listening on port 3000
getclients1
getclients2
getclients3
getclients4
GET /clients 200 6ms
getclients4 fin
getclients3 fin
getclients2 fin
getclients1 fin
-
```

Les quatre middlewares s'enchaînent les uns à la suite des autres. La question est la suivante : ne pourrait-on pas briser cet enchaînement et provoquer, par exemple, le passage de `getclients2` à `getclients4`, sans passer par `getclients3` ?

Express le permet en indiquant `next("route")` au lieu de simplement `next()`. L'instruction `next("route")` indique de quitter le traitement actuel de la route (effectué dans `app.get()`, `app.post()`, etc.) et de passer à la même route suivante (définie dans une autre méthode `app.get()`, `app.post()`, etc.).

Tableau 15–5 Méthode `next("route")`

Méthode	Signification
<code>next("route")</code>	Quitte le traitement actuel de la route pour donner la main à la même route qui suit (si elle existe), définie dans une autre méthode <code>app.get()</code> , <code>app.post()</code> , etc. Si aucune route n'est trouvée à la suite, donne la main au middleware suivant défini par <code>app.use()</code> .

Utiliser `next("route")`

```
/***
 * Module dependencies
 */

var express = require('express')
, connect = require('connect')
, routes = require('./routes')
, user = require('./routes/user')
, http = require('http')
, util = require('util')
, path = require('path');

var app = express();
```

```
// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.favicon());
app.use(app.router);
app.use(express.methodOverride());
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get("/clients", function getclients1(req, res, next) {
  console.log("getclients1");
  next();
  console.log("getclients1 fin");
}, function getclients2(req, res, next) {
  console.log("getclients2");
  next("route");
  console.log("getclients2 fin");
}, function getclients3(req, res, next) {
  console.log("getclients3");
  next();
  console.log("getclients3 fin");
});

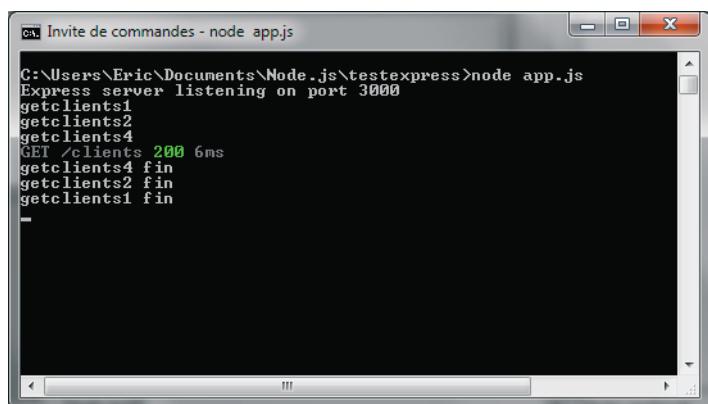
app.get("/clients", function getclients4(req, res, next) {
  console.log("getclients4");
  res.end("getclients4");
  console.log("getclients4 fin");
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

La fonction `getclients2()` appelle maintenant `next("route")`, ce qui empêche le traitement de `getclients3()` de s'effectuer.

Figure 15–45

Utilisation de next("route")
pour « sauter » un middleware



The screenshot shows a terminal window titled "Invite de commandes - node app.js". The window displays the following text:

```
C:\Users\Eric\Documents\Node.js\testexpress>node app.js
Express server listening on port 3000
getclients1
getclients2
getclients4
GET /clients 200 6ms
getclients4 fin
getclients2 fin
getclients1 fin
-
```


16

Envoyer la réponse du serveur

Jusqu'à présent, nous avons envoyé des éléments au navigateur, mais de façon sommaire en utilisant la méthode `res.end()` offerte par Node. Express permet plus de souplesse dans l'envoi de la réponse, en offrant des méthodes spécifiques selon le type de données renvoyées (HTML, JSON, fichiers, etc.).

Dans ce chapitre, nous examinons les différents moyens de retourner des informations du serveur vers le navigateur client. Les informations retournées peuvent être de différents types.

- Le code HTTP de la réponse : 200 si OK, 404 si page non trouvée, 500 si erreur sur le serveur, etc.
- L'en-tête HTTP de la réponse : la clé "`Content-Type`" est l'une des plus utilisées, mais il en existe d'autres et il est possible d'en créer des nouvelles.
- Le corps de la réponse : ce sont les informations que l'utilisateur verra affichées, contrairement au code et à l'en-tête HTTP que l'utilisateur ne voit pas directement.

Pour commencer, étudions comment retourner un code HTTP indiquant si la requête au serveur s'est bien déroulée.

Retourner un code HTTP

Le code HTTP fourni dans la réponse permet d'indiquer le résultat global de la requête effectuée au serveur.

Les différentes catégories de code HTTP

Le protocole HTTP fournit plusieurs valeurs de codes possibles, réparties en cinq grandes catégories : 100, 200, 300, 400 et 500. Chacune de ces catégories regroupe plusieurs codes spécifiant des différences dans le résultat, par exemple dans la catégorie 400, on trouve :

- 404, qui correspond à une page non trouvée ;
- 405, qui signifie que l'on a essayé d'utiliser une méthode HTTP non autorisée ;
- etc.

Tableau 16–1 Catégories des codes HTTP fournis en réponse

Code	Signification
>= 100	Retourné pour indiquer des informations provisoires, en attendant d'avoir la vraie réponse du serveur (avec les codes qui suivent dans ce tableau).
>= 200	Retourné pour indiquer que la réponse est OK.
>= 300	Retourné pour indiquer une redirection vers une autre adresse.
>= 400	Retourné pour indiquer une erreur du client, par exemple : <ul style="list-style-type: none"> - 404 page non trouvée ; - 405 méthode HTTP non autorisée. Dans ce cas, le client a demandé l'accès à une ressource inconnue, ce qui provoque l'erreur.
>= 500	Retourné pour indiquer une erreur du serveur, en général une erreur d'exécution.

Le code HTTP retourné par le serveur est géré par Express et dépend des routes que nous avons préalablement écrites dans notre application.

Toutefois, dans le traitement de chaque route, on peut indiquer le code que l'on souhaite renvoyer, et ainsi outrepasser le code renvoyé par défaut dans Express.

Utiliser le code HTTP pour retourner une valeur

```
/**
 * Module dependencies
 */

var express = require('express');
var http = require('http');
var path = require('path');
var util = require('util');

var app = express();
```

```
// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public'))));
app.use(express.static(path.join(__dirname, 'forms'))));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get("/clients", function(req, res) {
  res.end("GET /clients");
});

app.all("/clients", function(req, res) {
  res.writeHead(405);
  res.end("Erreur 405 : Méthode " + req.method + " inconnue");
});

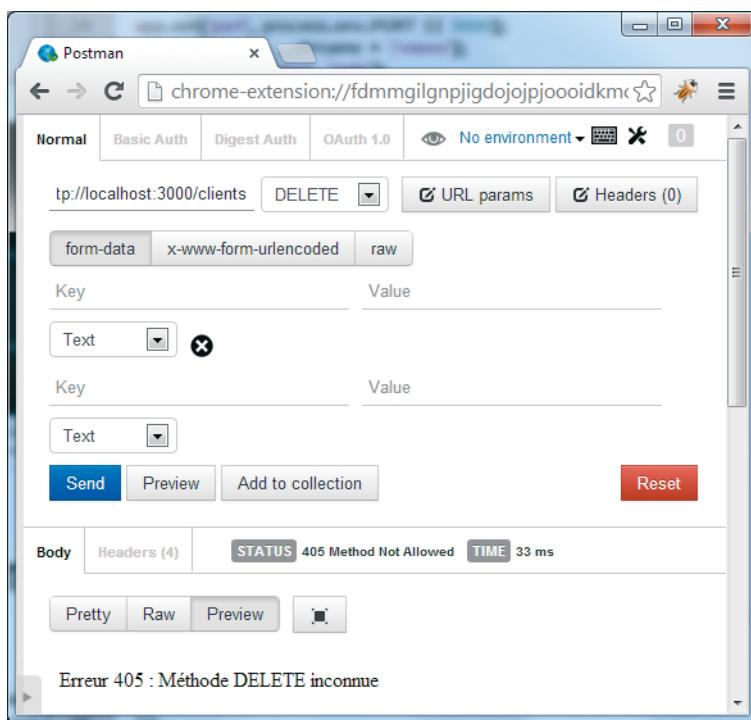
app.all(/.*/, function(req, res) {
  res.writeHead(404);
  res.end("Erreur 404 : " + req.method + req.url + " : route inconnue");
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

- La première route `GET /clients` affiche la route utilisée. Un code HTTP valant 200 est automatiquement généré par Express, sans aucune intervention de notre part.
- La deuxième route affiche une erreur de méthode HTTP (erreur 405) sur les URL `/clients`, pour les méthodes autres que `GET`. En effet, les routes `GET /clients` sont déjà traitées par la route précédente, qui renvoie la réponse.
- Enfin, la troisième et dernière route affiche un message d'erreur pour toutes les autres routes (indépendamment de la méthode et de l'URL utilisées).

Affichons, par exemple, le retour lorsqu'on utilise la route `DELETE /clients`, interceptée par la seconde route.

Figure 16–1
Utilisation de la route DELETE /clients avec Postman



On utilise l'outil Postman de Chrome afin de générer la méthode `DELETE` sans avoir à écrire de formulaire HTML.

Utiliser la méthode `res.status(statusCode)`

On utilise ici la méthode `res.writeHead(statusCode)` pour retourner le code HTTP. Mais la méthode `res.writeHead()` définie dans Node permet également d'utiliser un second paramètre (non utilisé pour l'instant) pour renvoyer un ou plusieurs en-têtes HTTP.

Express a défini en interne la méthode `res.status(statusCode)` qui retourne simplement le code HTTP, sans les en-têtes. Le programme précédent peut donc également s'écrire :

Utiliser `res.status()` pour retourner le code HTTP

```
/**  
 * Module dependencies.  
 */
```

```
var express = require('express');
var http = require('http');
var path = require('path');
var util = require('util');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public'))));
app.use(express.static(path.join(__dirname, 'forms')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get("/clients", function(req, res) {
  res.end("GET /clients");
});

app.all("/clients", function(req, res) {
  res.status(405);
  res.end("Erreur 405 : Méthode " + req.method + " inconnue");
});

app.all(/.*/, function(req, res) {
  res.status(404);
  res.end("Erreur 404 : " + req.method + req.url + " : route inconnue");
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

La tableau 16-2 récapitule le fonctionnement de la nouvelle méthode `res.status()`.

Tableau 16-2 Méthode `res.status()`

Méthode	Signification
<code>res.status(statusCode)</code>	Retourne dans la réponse le code HTTP indiqué.

Retourner un en-tête au navigateur

Node a défini la méthode `res.setHeader(name, value)` qui permet de positionner l'en-tête `name` à la valeur `value`. On a souvent utilisé cette méthode pour positionner l'en-tête "`Content-Type`" afin d'indiquer le format d'affichage des données renvoyées (simple texte, HTML, XML ou JSON).

On peut également utiliser cette méthode pour retourner de nouveaux en-têtes. Express, par exemple, utilise l'en-tête "`X-Powered-By`" positionné à la valeur "`Express`" afin de laisser une trace dans la réponse envoyée au navigateur.

Il est de coutume de commencer un nouvel en-tête par la chaîne "`X-`", et de séparer chacun des mots par "`-`", le mot suivant commençant alors par une majuscule.

Afin d'améliorer la méthode `res.setHeader()` proposée par Node, Express a ajouté la nouvelle méthode `res.set()`. Elle peut accepter les deux paramètres `name` et `value` comme la méthode `res.setHeader()` de Node, mais également un objet dans lequel on peut définir des couples `name : value`.

Tableau 16-3 Méthode `res.set()`

Méthode	Signification
<code>res.set(name, value)</code>	Positionne l'en-tête <code>name</code> à la valeur <code>value</code> (identique à <code>res.setHeader(name, value)</code> définie dans Node).
<code>res.set(obj)</code>	Positionne les en-têtes définis dans l'objet <code>obj</code> . Cela permet de définir plusieurs en-têtes simultanément.

Voyons sur un exemple comment utiliser la méthode `res.set(obj)`.

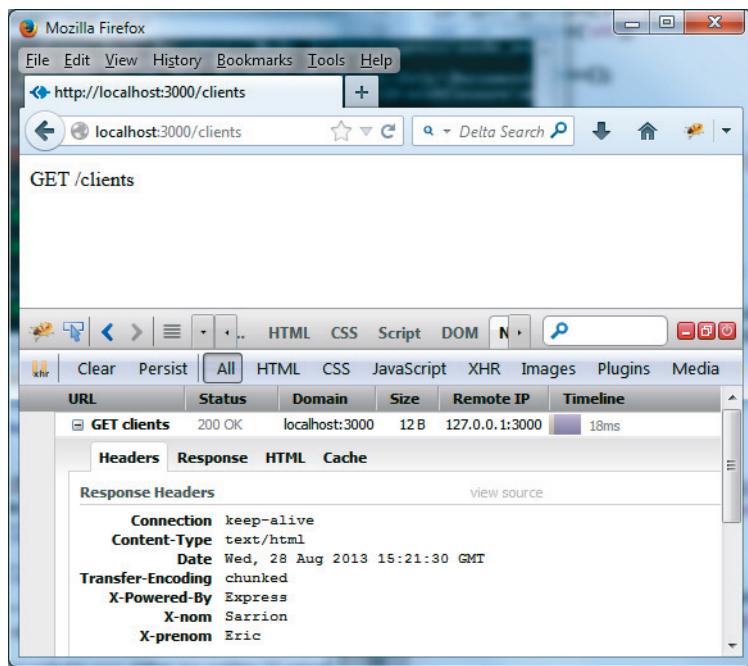
Utiliser `res.set(obj)` pour définir les en-têtes "Content-Type", "X-nom" et "X-prenom"

```
app.get("/clients", function(req, res) {
  res.set({
    "Content-Type" : "text/html",
    "X-nom" : "Sarrion",
    "X-prenom" : "Eric"
  });
  res.end("GET /clients");
});
```

On voit grâce à Firebug que les trois en-têtes ont été positionnés dans la réponse envoyée au navigateur.

Figure 16–2

Utilisation de `res.set()` pour positionner un en-tête



Retourner le corps de la réponse

Le corps de la réponse correspond à la partie visible destinée à l'utilisateur. C'est ce qui sera affiché dans le navigateur. Le code HTTP et les en-têtes vus précédemment ne sont que des éléments de programmation du serveur, servant à sa gestion et non visibles des utilisateurs.

Node permet de retourner différents types de données, comme on l'a vu dans les précédents chapitres : HTML, XML, JSON et fichiers statiques. Express a ajouté de nouvelles méthodes définies sur l'objet `res`, afin de retourner plus facilement ces types de données, et a ajouté des nouvelles fonctionnalités comme l'affichage des vues définies dans des fichiers externes.

Retourner du code HTML encodé en UTF-8

Par défaut, Node retourne le résultat en HTML, mais sans utiliser l'encodage UTF-8. Pour retourner du HTML encodé en UTF-8 avec Node, il faut modifier l'en-tête "`Content-Type`" pour le positionner à la valeur "`text/html; charset=utf-8`". Le texte envoyé par `res.end(html)` sera alors correctement encodé en UTF-8.

Express considère à l'inverse que vous souhaitez prioritairement envoyer du HTML encodé en UTF-8. Il effectue donc pour nous le positionnement correct de l'en-tête "Content-Type" à la valeur "text/html; charset=utf-8". On utilise pour cela la nouvelle méthode `res.send(html)`.

Tableau 16–4 Méthode `res.send()`

Méthode	Signification
<code>res.send(html)</code>	Retourne le code HTML indiqué dans la réponse du serveur. L'en-tête "Content-Type" est positionné automatiquement à la valeur "text/html; charset=utf-8".

Utiliser `res.send()` pour retourner du HTML

```
app.get("/clients", function(req, res) {
  res.send("<h1> Liste affichée des clients </h1>");
});
```

Figure 16–3
Retourner du code HTML
au navigateur



En utilisant uniquement la méthode `res.end()` de Node, on devrait alors écrire :

Utiliser `res.end()` de Node pour retourner du HTML encodé en UTF-8

```
app.get("/clients", function(req, res) {
  res.setHeader("Content-type", "text/html; charset=utf-8");
  res.end("<h1> Liste affichée des clients </h1>");
});
```

Si le positionnement correct de l'en-tête "Content-Type" n'est pas fait, le code HTML s'affiche mais pas dans le bon encodage. Mettons la ligne correspondante en commentaires :

Utiliser `res.end()` de Node pour retourner du HTML, non encodé en UTF-8

```
app.get("/clients", function(req, res) {
  // res.setHeader("Content-type", "text/html; charset=utf-8");
  res.end("<h1> Liste affichée des clients </h1>");
});
```

Figure 16–4

Code HTML retourné sans encodage



On voit ici l'intérêt d'utiliser la méthode `res.send()` plutôt que la méthode `res.end()`. La méthode `res.send()` nous évite de positionner nous-mêmes le type des données renvoyées (par défaut, du HTML), et l'encodage à utiliser (par défaut, "`utf-8`").

Si plusieurs méthodes `res.send()` se trouvent dans le middleware qui envoie la réponse, seul le premier appel est traité. Les suivants ne sont pas pris en compte.

Retourner du texte simple

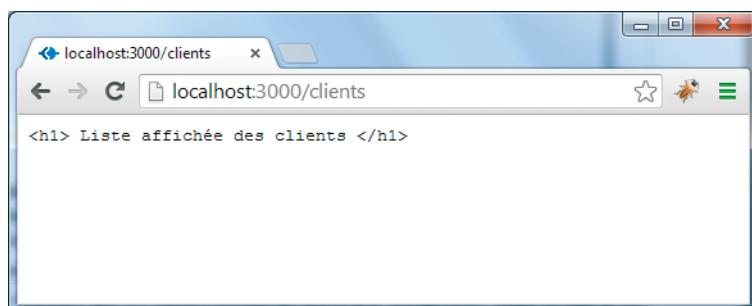
Il peut arriver que l'on souhaite que les balises HTML ne soient pas interprétées. Dans ce cas, il suffit d'indiquer que le "`Content-Type`" est "`text/plain`" (au lieu de "`text/html`", par défaut). L'encodage (`charset`) doit également être indiqué en "`utf-8`".

Afficher du texte non pris en compte comme du HTML

```
app.get("/clients", function(req, res) {
  res.setHeader("Content-type", "text/plain; charset=utf-8");
  res.send("<h1> Liste affichée des clients </h1>");
});
```

Figure 16–5

Retourner du texte non interprété en HTML au navigateur



Retourner du JSON

Retourner du JSON est très fréquent dans les sites web actuels, notamment pour fournir une API permettant d'interroger le serveur (services web).

On utilise pour cela la méthode `res.json(obj)` fournie par Express, qui retourne sous forme JSON l'objet passé en paramètre.

Tableau 16–5 Méthode `res.json()`

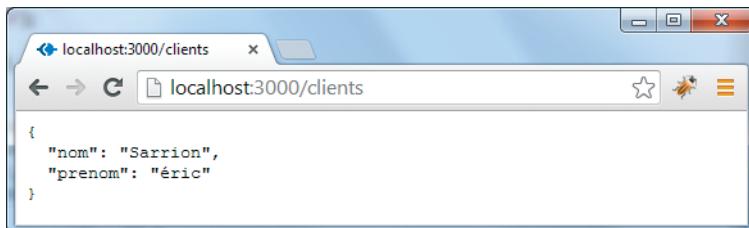
Méthode	Signification
<code>res.json(obj)</code>	Retourne l'objet indiqué sous forme JSON. L'en-tête "Content-Type" est positionné automatiquement à la valeur "application/json". Si l'on souhaite utiliser des caractères UTF-8, il faut l'indiquer au moyen de <code>res.charset = "utf-8"</code> .

Envoyer du JSON en utilisant `res.json()`

```
app.get("/clients", function(req, res) {
  res.charset = "utf-8";
  res.json({ nom : "Sarrion", prenom : "éric" });
});
```

L'instruction `res.charset = "utf-8"` permet de positionner l'encodage en UTF-8 sans utiliser l'en-tête "Content-Type". Cette instruction sera utile pour afficher correctement les caractères accentués présents dans l'objet JSON.

Figure 16–6
Retourner du JSON
au navigateur



Voici l'équivalent, sans utiliser la méthode `res.json()` offerte par Express, mais en utilisant seulement les fonctionnalités de Node :

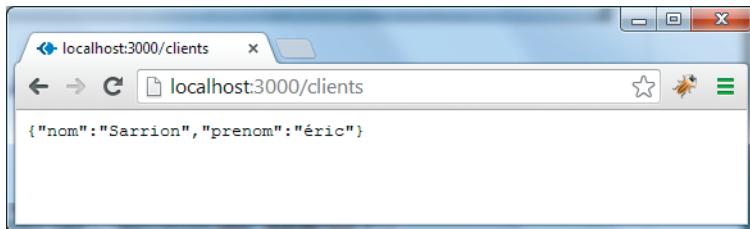
Envoyer du JSON sans utiliser `res.json()`

```
app.get("/clients", function(req, res) {
  res.setHeader("Content-type", "application/json; charset=utf-8");
  res.end(JSON.stringify({ nom : "Sarrion", prenom : "éric" }));
});
```

Nous indiquons que le "Content-Type" est "application/json", avec "charset=utf-8". Le texte à afficher est transformé en chaîne de caractères au moyen de `JSON.stringify()`.

Figure 16–7

Retourner du JSON au navigateur



Retourner des fichiers statiques

Un fichier statique représente un fichier qui peut être utilisé sans aucune transformation par le navigateur. Il s'agit en général d'un fichier texte, d'une image, d'une vidéo, etc. Il suffit de saisir l'URL du fichier dans le navigateur et celui-ci se charge de le récupérer sur le serveur, puis de l'utiliser, par exemple en l'affichant.

Le middleware `static` permet d'utiliser facilement des fichiers statiques avec Express. On a vu l'utilisation de ce middleware dans Connect. Express l'a repris en interne, mais au lieu de l'utiliser sur le module `connect`, on l'utilise maintenant sur le module `express`.

Le fichier `app.js` créé par Express contient l'utilisation de ce middleware, en permettant que tous les fichiers statiques se trouvant dans le répertoire `public` soient automatiquement pris en compte par Express.

Prise en compte des fichiers statiques se trouvant dans le répertoire public

```
app.use(express.static(path.join(__dirname, 'public')));
```

Le répertoire `public` est concaténé avec le répertoire principal de l'application Express, mémorisé dans la variable `__dirname` définie dans Node.

Le répertoire `public` est créé par Express et contient trois sous-répertoires `images`, `javascripts` et `stylesheets`. Ces sous-répertoires permettront d'accéder facilement aux différents fichiers de notre application.

Il est bien sûr possible d'ajouter de nouveaux répertoires qui contiendront ce genre de fichiers statiques. Il suffit d'utiliser le middleware `static` pour chacun des répertoires ajoutés.

Retourner des fichiers dynamiques

Lorsqu'on désire afficher un fichier particulier qui n'est pas dans les répertoires définis par le middleware `static` (par exemple, un fichier créé dynamiquement), on utilise la méthode `res.sendFile(name)` qui retourne le fichier au navigateur, qui pourra ensuite l'afficher.

Une variante de cette méthode permet de renvoyer ce fichier vers le client, mais pour être seulement sauvegardé sur son disque (sans visualisation dans le navigateur comme le permet `res.sendFile()`). C'est la méthode `res.download(name)`. Une autorisation de l'utilisateur est demandée afin d'effectuer la sauvegarde sur le disque de l'utilisateur.

Tableau 16–6 Méthodes `res.sendFile()` et `res.download()`

Méthode	Signification
<code>res.sendFile(name)</code>	Retourne le fichier représenté par <code>name</code> afin de l'afficher dans le navigateur.
<code>res.download(name)</code>	Retourne le fichier représenté par <code>name</code> afin de l'enregistrer sur le disque du client. Une autorisation du client est demandée avant de le sauvegarder.

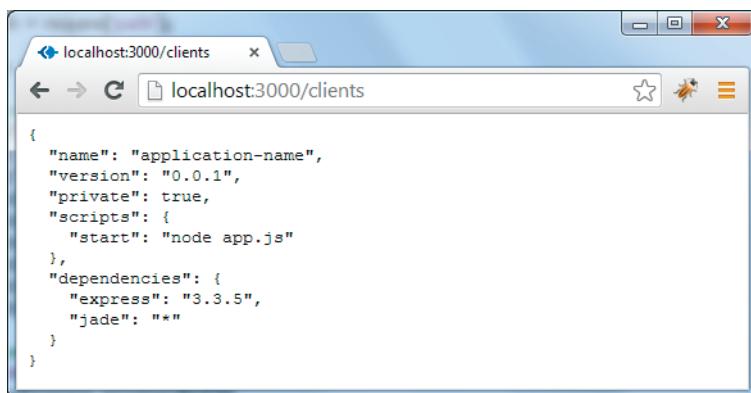
Utilisons la méthode `res.sendFile()` afin d'afficher le contenu du fichier `package.json` se trouvant dans le répertoire principal de l'application créée avec Express.

Utiliser la méthode `res.sendFile()`

```
app.get("/clients", function(req, res) {
  res.sendFile("package.json");
});
```

Figure 16–8

Envoi d'un fichier au serveur



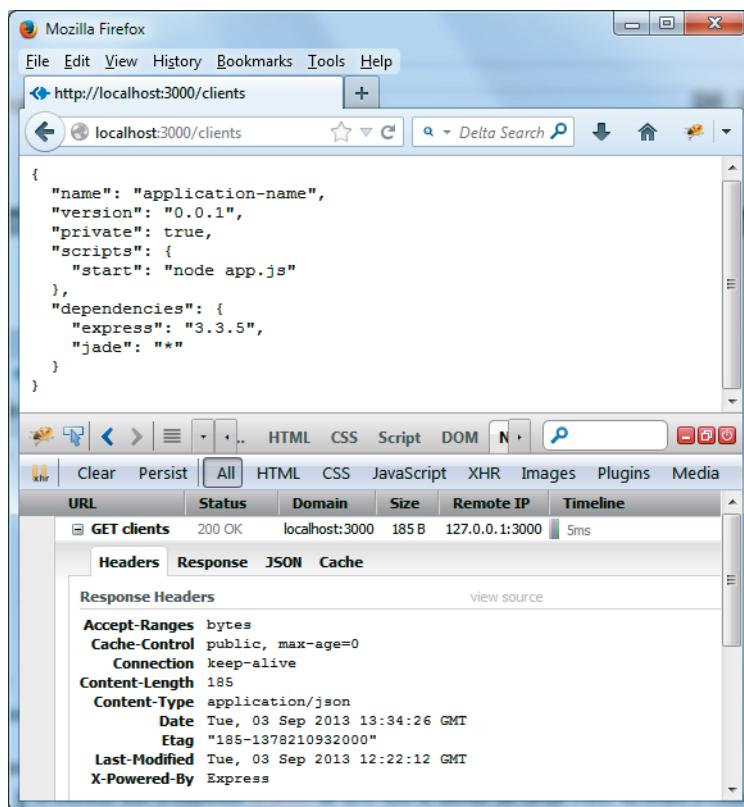
Le contenu du fichier `package.json` est affiché dans le navigateur.

La différence avec le middleware `static` est qu'ici les fichiers à visualiser ne doivent pas obligatoirement se trouver dans des répertoires prédéfinis. Ils peuvent être n'importe où et même être créés dynamiquement.

Selon l'extension du fichier à transférer, l'en-tête "Content-Type" est modifié pour tenir compte du type de contenu. Ainsi, pour le fichier précédent, on obtient dans Firefox :

Figure 16–9

En-tête utilisé pour retourner du JSON au navigateur



L'en-tête "Content-Type" est ici positionné à la valeur "application/json" par Express.

Retourner une vue

Une des fonctionnalités les plus intéressantes d'Express est de permettre de construire des vues qui seront affichées à la demande. Les vues sont des fichiers externes qui peuvent contenir du code HTML, et auxquelles on peut également transmettre des paramètres depuis le code JavaScript.

Pour cela, Express a créé la méthode `res.render(name)`, dans laquelle `name` représente le fichier associé à la vue. Ce fichier devra se trouver dans le répertoire associé aux vues, défini par l'instruction `app.set('views', __dirname + '/views')` se trouvant dans le fichier `app.js`.

Définir le répertoire dans lequel se trouvent les vues du programme

```
app.set('views', __dirname + '/views');
```

Une autre fonctionnalité intéressante d'Express est de pouvoir définir le format des vues que l'on souhaite utiliser. En effet, les vues peuvent être écrites dans différents formats, tels que JADE, EJS, etc. Un format correspond à un langage dans lequel la vue est écrite (qui n'est pas forcément écrite en HTML). Express effectue une transformation de la vue en code HTML, ce qui est nécessaire pour que le navigateur puisse afficher la page reçue du serveur.

Lors de la création de l'application Express, le format par défaut utilisé dans les vues est JADE. Cela est défini dans le code du fichier `app.js` au moyen de l'instruction `app.set('view engine', 'jade')` et permet, si on le souhaite, de ne pas indiquer l'extension du fichier lors de l'appel à la méthode `res.render(name)`.

Indiquer que le format des vues est JADE

```
app.set('view engine', 'jade');
```

Utilisons Express pour afficher une vue écrite avec JADE. Pour cela, Express propose d'utiliser la méthode `res.render(name)` qui affiche la vue `name` se trouvant dans le répertoire des vues (positionné par l'instruction `app.set('views', __dirname + '/views')`).

Tableau 16-7 Méthodes gérant les vues dans Express

Méthode	Signification
<code>app.set("views", directory)</code>	Indique dans quel répertoire les vues sont stockées.
<code>app.set("view engine", format)</code>	Indique l'extension (format) des noms de fichiers associés aux vues : "jade", "ejs", etc. Ceci permet de ne pas indiquer d'extension dans le nom de fichier lui-même, lors de l'utilisation de la méthode <code>res.render(name)</code> .
<code>res.render(name, [obj])</code>	Affiche la vue ayant le nom de fichier <code>name</code> se trouvant dans le répertoire des vues. Un objet optionnel peut être transmis contenant des informations utilisées par la vue. Si <code>name</code> ne contient pas d'extension, l'extension du fichier correspond à celle indiquée lors de l'appel de la méthode <code>app.set("view engine", format)</code> .

Utiliser la méthode res.render()

```
app.get("/clients", function(req, res) {  
  res.render("clients.jade");  
});
```

Les instructions précédentes indiquent que lorsque la route `GET /clients` est invoquée, la vue `clients.jade` doit être affichée en résultat.

La vue `clients.jade` est la suivante :

Fichier clients.jade (dans le répertoire views de l'application)

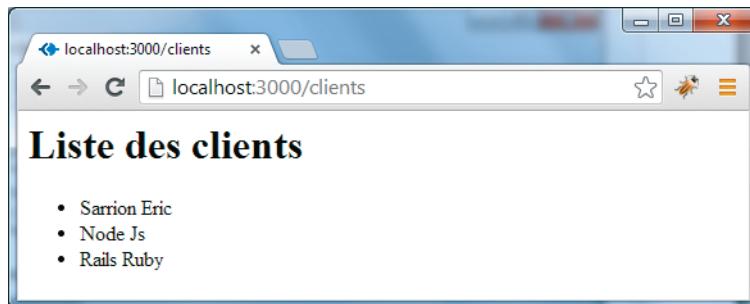
```
h1 Liste des clients  
ul  
  li Sarrion Eric  
  li Node Js  
  li Rails Ruby
```

Ne soyez pas surpris par la façon dont le fichier JADE est écrit. En effet, il est écrit en utilisant la syntaxe JADE. Au lieu de mettre les balises entourées des caractères `<` et `>` comme en HTML, on les utilise directement par leur nom. De plus, l'indentation des lignes indique l'imbrication des balises. N'oubliez pas que même si ce fichier est écrit dans un format non HTML, il sera finalement traduit en HTML par Express.

Affichons l'URL `http://localhost:3000/clients` dans le navigateur.

Figure 16-10

Retourner une vue écrite en JADE au navigateur



La liste des clients s'affiche correctement. Une amélioration serait de la transmettre en paramètres de la vue, plutôt que les écrire un par un dans la vue elle-même.

Pour cela, il faut modifier l'appel de la méthode `res.render()` en lui transmettant un second argument qui est un objet JSON qui contiendra cette liste de clients. La vue devra récupérer cette liste et l'afficher.

Transmettre la liste des clients à la vue

```
app.get("/clients", function(req, res) {
  res.render("clients.jade", { clients : [
    { nom : "Sarrion", prenom : "Eric" },
    { nom : "Node", prenom : "Js" },
    { nom : "Rails", prenom : "Ruby" }
  ]});
});
```

La liste des clients est un objet `{ clients : value }`, dans lequel `value` est un tableau d'objets `{ nom, prenom }`. Regardons comment la vue peut récupérer ce paramètre.

Récupérer la liste des clients dans la vue (fichier clients.jade)

```
h1 Liste des clients
ul
  li #{clients[0].nom} #{clients[0].prenom}
  li #{clients[1].nom} #{clients[1].prenom}
  li #{clients[2].nom} #{clients[2].prenom}
```

La clé `clients` utilisée dans l'objet transmis correspond à une variable du même nom `clients` dans la vue. La valeur associée à la clé `clients` étant un tableau, on accède à chaque élément au moyen de `clients[index]`. Chaque élément du tableau étant lui-même un objet `{ nom, prenom }`, on accède au nom et au prénom de chaque client au moyen de `clients[index].nom` et `clients[index].prenom` respectivement.

La notation `#{expression JavaScript}` est en fait le moyen d'afficher sous forme de chaîne de caractères le résultat de l'expression JavaScript indiquée. On souhaite ici afficher le nom suivi du prénom, séparés par un espace, le tout dans un élément ``. Remarquez qu'on aurait pu également écrire chacune des lignes de la liste de la façon suivante.

Autre façon d'écrire les éléments de liste (fichier clients.jade)

```
h1 Liste des clients
ul
  li #{clients[0].nom + " " + clients[0].prenom}
  li #{clients[1].nom + " " + clients[1].prenom}
  li #{clients[2].nom + " " + clients[2].prenom}
```

En effet, l'expression JavaScript mise entre les accolades peut comporter des instructions JavaScript quelconques, sachant que l'on manipule des chaînes de caractères.

Une autre façon d'écrire ces mêmes lignes est la suivante. Il suffit d'indiquer immédiatement après la balise `li` le signe `=`, signifiant d'évaluer l'expression JavaScript qui suit.

Utiliser le signe `=` pour évaluer le code JavaScript qui suit (fichier clients.jade)

```
h1 Liste des clients
ul
  li= clients[0].nom + " " + clients[0].prenom
  li= clients[1].nom + " " + clients[1].prenom
  li= clients[2].nom + " " + clients[2].prenom
```

Si vous oubliez le signe `=`, le code JavaScript qui suit n'est pas évalué et il sera considéré comme du texte qui sera affiché tel quel dans la page.

Enfin, rendons plus efficace le code de la vue. Plutôt que d'écrire nous-mêmes chacun des éléments du tableau, faisons une boucle pour parcourir le tableau des clients.

Utiliser une boucle each pour lister les clients (fichier clients.jade)

```
h1 Liste des clients
ul
  each client in clients
    li= client.nom + " " + client.prenom
```

La variable `clients` est la même que précédemment, et elle correspond au tableau de clients transmis. On parcourt ce tableau et on récupère dans une variable `client`, chacun des clients à afficher. La notation `li=` est la même que celle étudiée précédemment.

Le langage offert par JADE peut être approfondi sur <http://jade-lang.com/>. Dans un chapitre suivant, nous étudierons un autre langage utilisable dans les vues, à savoir le format EJS (*Embedded JavaScript*).

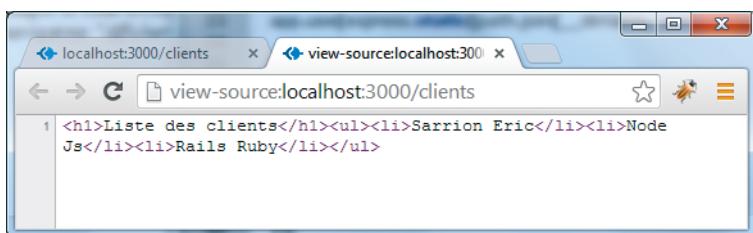
Permettre un affichage lisible du code HTML de la page

Dans tous nos exemples, et en particulier le dernier, vous verrez que le code HTML de la page est optimisé de façon à tenir sur le moins de caractères possibles. Les espaces et les retours chariots inutiles sont supprimés automatiquement par Express, ce qui rend la page HTML plus rapide à charger, mais aussi rend le code HTML généré difficile à lire !

Par exemple, voici le code HTML généré par l'affichage de la liste de clients précédente (obtenu par le menu du navigateur « Afficher le code source de la page »).

Figure 16–11

Code source de la page HTML
retournée par le serveur



Pour rendre cet affichage plus lisible, il suffit de l'indiquer à Express, en utilisant l'instruction `app.locals.pretty = true` dans le fichier `app.js`.

Rendre le code HTML plus lisible (fichier app.js)

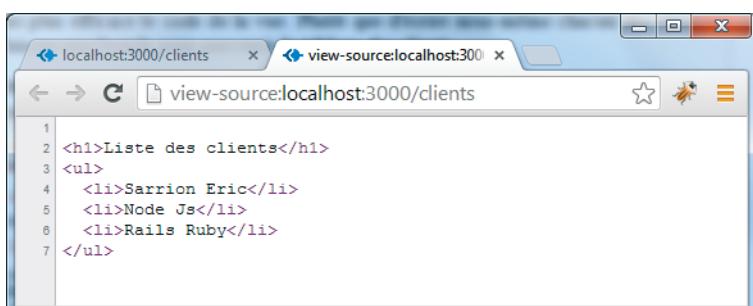
```
app.locals.pretty = true;
```

Une fois cette instruction exécutée, le code HTML de la page est maintenant mieux disposé.

Figure 16–12

Figure 10-12

Code source plus lisible



Rediriger vers une autre URL

Il est également possible de rediriger vers une autre URL, qui doit être accessible avec une méthode de type `GET`, sinon elle est inaccessible (pour des raisons de sécurité). L'URL peut être sur notre site ou vers un site externe.

Tableau 16-8 Méthode res.redirect(url)

Méthode	Signification
<code>res.redirect(url)</code>	Redirige vers l'URL indiquée, qui doit être accessible par la méthode GET. L'URL peut être interne ou externe : - <code>res.redirect("/admin")</code> effectue un appel interne à GET /admin ; - <code>res.redirect("admin")</code> effectue un appel interne à GET admin, relativement à l'emplacement actuel ; - <code>res.redirect("http://yahoo.com")</code> redirige vers le site externe yahoo.com.

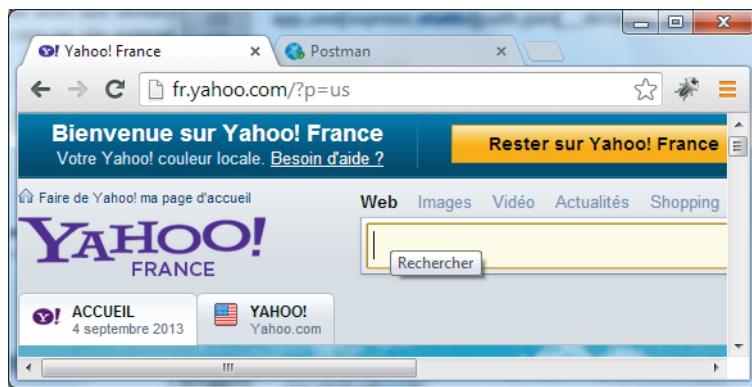
Effectuer une redirection vers yahoo.com

```
app.get("/clients", function(req, res) {  
    res.redirect("http://yahoo.com");  
});
```

Lorsque l'URL `/clients` sera entrée dans le navigateur, on sera automatiquement redirigé vers le site <http://yahoo.com>.

Figure 16–13

La page yahoo.com est affichée.



17

Objets app, req et res utilisés par Express

Dans les précédents chapitres, nous avons abondamment utilisé les objets `app`, `req` et `res` mis à notre disposition par Express. Le but ici est de découvrir les principales fonctionnalités proposées par chacun de ces objets, en particulier celles que nous n'avons pas encore décrites.

Objet app : gérer l'application Express

L'objet `app` est créé par le retour de l'appel à la méthode `express()`. Il sert à gérer l'application créée avec Express.

Créer l'objet app à partir de `express()`

```
var express = require("express");
var app = express();
```

On a vu que l'on pouvait utiliser, sur cet objet, principalement les méthodes `app.get()`, `app.post()`, `app.delete()`, `app.put()`, `app.all()` et `app.use()`, décrites dans les chapitres précédents. Ces méthodes permettent de créer des middlewares et de gérer le routage au moyen de REST.

Parmi les nouvelles méthodes que l'on va étudier dans ce chapitre, on trouve :

- la possibilité de gérer le format des variables dans les URL ;
- la possibilité de connaître le code HTML généré par une vue, même si elle est dynamique ;
- la possibilité de partager des objets avec les vues.

Étudions maintenant ces nouvelles possibilités.

Gérer le format des variables dans les URL

On a vu au chapitre 15, « Routage des requêtes avec Express », l'utilisation des variables lors de la définition des URL dans les routes. L'intérêt des variables est de pouvoir récupérer leur valeur dans le code de la fonction de callback traitant la route. Par exemple, la valeur de la variable `:id` est disponible dans l'objet `req.params.id`. Toute autre variable indiquée dans l'URL sera accessible de la même façon, en tant que propriété de l'objet `req.params`.

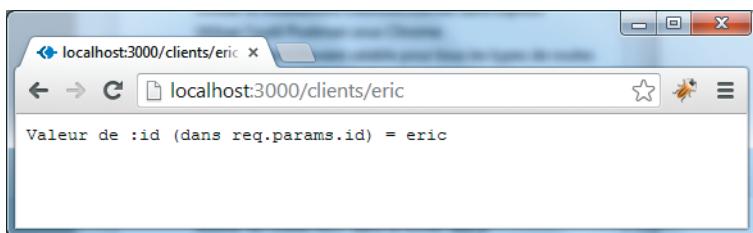
Par exemple, pour afficher la valeur de la variable `:id` dans la page, en utilisant une route `GET /clients/:id`, on écrira :

Afficher la valeur de la variable `:id`

```
app.get("/clients/:id", function(req, res) {  
    res.end("Valeur de :id (dans req.params.id) = " + req.params.id);  
});
```

Si on utilise l'URL `http://localhost:3000/clients/eric` dans un navigateur, la valeur "eric" est transmise et affichée.

Figure 17-1
Affichage du contenu
de la variable `:id`



À ce stade, toutes les valeurs pour la variable `:id` sont possibles, car aucun contrôle n'est effectué sur les valeurs transmises. Si on souhaite, par exemple, que les valeurs de cette variable `:id` soit uniquement entières, la vérification n'est pas effectuée pour l'instant. D'où l'idée d'ajouter un contrôle sur les valeurs des variables.

Pour cela, Express met la méthode `app.param(name, callback)` à notre disposition. Elle permet de définir un middleware pour lequel la fonction de callback sera appelée chaque fois que la variable `name` sera utilisée dans une URL.

Tableau 17–1 Méthode app.param(name, callback)

Méthode	Signification
<code>app.param(name, callback)</code>	Appelle la fonction de callback pour chaque URL qui contient la variable <code>name</code> indiquée. La fonction de callback se comporte comme un middleware et doit donc utiliser <code>next()</code> pour passer au middleware suivant.

Voici un exemple d'utilisation de cette méthode.

Utilisation de app.param(name, callback)

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.favicon());  
app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
    app.use(express.errorHandler());  
}  
  
app.param("id", function(req, res, next, id){  
    console.log("Valeur du paramètre id : " + id);  
    console.log("Egalement accessible dans req.params.id : " + req.params.id);  
})
```

```

    next();
});

app.get("/clients/:id", function(req, res) {
  res.end("Valeur de :id (dans req.params.id) = " + req.params.id);
});

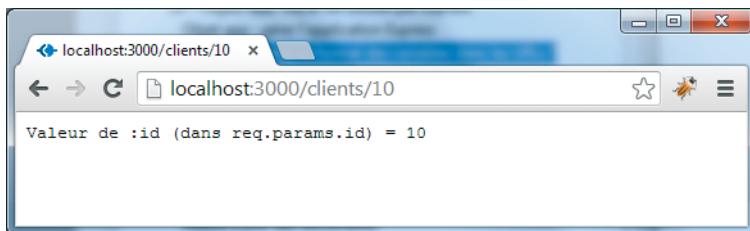
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});

```

Nous utilisons ici la variable `:id` dont nous affichons la valeur lorsque la route `GET /clients/:id` est invoquée. L'utilisation du middleware défini dans `app.param()` permet d'afficher dans la console la valeur de la variable `:id`, transmise en dernier argument du middleware.

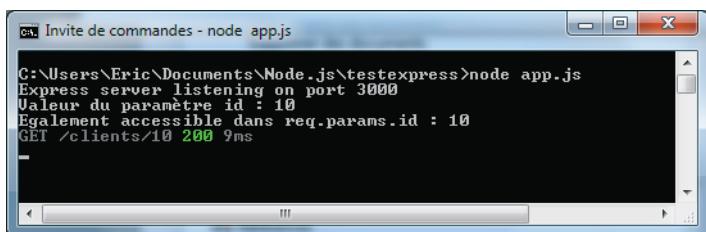
Par exemple, en utilisant l'URL `http://localhost:3000/clients/10` dans la barre d'adresses du navigateur :

Figure 17-2
Affichage du contenu
de la variable `:id`



Tandis que la console du serveur affiche maintenant :

Figure 17-3
Affichage de la console
du serveur



L'intérêt de cette technique est de pouvoir effectuer un traitement (dans `app.param()`) avant que celui de la requête soit effectué (dans `app.get()`). Par exemple, si on souhaite interdire les URL pour lesquelles la variable `:id` n'est pas numérique, on écrira :

Autoriser uniquement les valeurs entières pour la variable :id

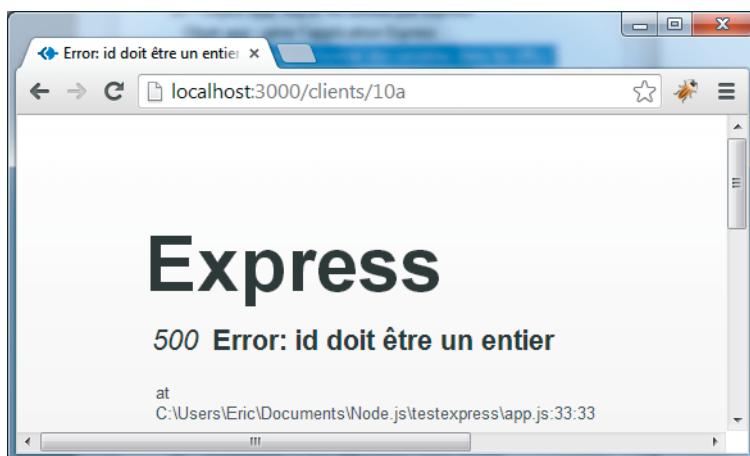
```
app.param("id", function(req, res, next, id){  
    console.log("Valeur du paramètre id : " + id);  
    console.log("Également accessible dans req.params.id : " + req.params.id);  
    if (!id.match(/\d+/)) next(new Error("id doit être un entier"));  
    else next();  
});
```

Si la valeur de :id n'est pas entière, une exception est générée au moyen de `next(new Error())`. Tandis que si tout se déroule correctement, on passe au middleware suivant avec `next()`.

En saisissant l'URL `http://localhost:3000/clients/10a` dans la barre d'adresses du navigateur, donc en indiquant une variable :id non numérique :

Figure 17–4

Interdire les valeurs non numériques pour :id

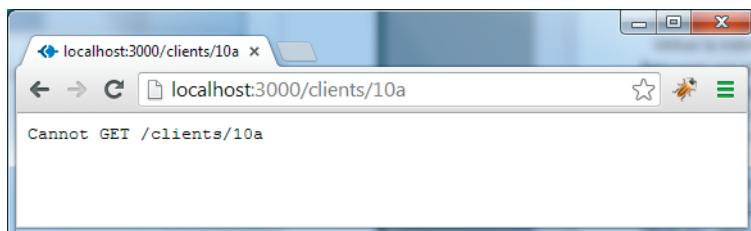


On peut remplacer avantageusement `next(new Error())` par `next("route")`, ce qui provoque le passage à une autre route qui suit, si une telle route existe. Si elle n'existe pas, il y a alors une erreur de routage générée par Express.

Utiliser `next("route")` au lieu de `next(new Error())`

```
app.param("id", function(req, res, next, id){  
    console.log("Valeur du paramètre id : " + id);  
    console.log("Également accessible dans req.params.id : " + req.params.id);  
    if (!id.match(/\d+/)) next("route");  
    else next();  
});
```

Figure 17–5
La route n'est pas trouvée.



Remarquez qu'on utilise `next("route")` et non pas `next()`, sinon on passerait au middleware suivant qui est celui défini dans la méthode `app.get()`, ce qui empêcherait toute forme de validation.

Récupérer et modifier le code HTML généré par une vue

On sait comment afficher le résultat d'une vue dans le navigateur au moyen de la méthode `res.render()`. La méthode `app.render()` définie sur l'objet `app` (au lieu de l'objet `res`) permet de définir une fonction de callback qui récupère le code HTML généré et peut éventuellement le modifier avant de l'envoyer dans la réponse du serveur.

Tableau 17–2 Méthode `app.render(name, [obj], callback)`

Méthode	Signification
<code>app.render(name, [obj], callback)</code>	Retourne le code HTML de la vue <code>name</code> dans la fonction de callback, qui est de la forme <code>callback(err, html)</code> . Ce code HTML peut être modifié et retourné dans la réponse par <code>res.send(html)</code> . Le paramètre <code>obj</code> est un objet facultatif permettant de transmettre des données à la vue.

Dans l'exemple qui suit, on affiche la vue `clients.jade`, dans laquelle on modifie le nom et le prénom du client "Node Js" en "Admin JavaScript".

Code de la vue `clients.jade` (dans le répertoire `views`)

```
h1 Liste des clients
ul
  each client in clients
    li= client.nom + " " + client.prenom
```

Le code de la vue correspond à celui que nous avions utilisé dans un précédent chapitre.

Utiliser app.render() pour modifier le code HTML de la vue

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.favicon());  
app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
    app.use(express.errorHandler());  
}  
  
app.locals.pretty = true;  
  
app.get("/clients", function(req, res) {  
    app.render("clients.jade", { clients : [  
        { nom : "Sarrion", prenom : "Eric" },  
        { nom : "Node", prenom : "Js" },  
        { nom : "Rails", prenom : "Ruby" }  
    ], function(err, html) {  
        console.log(html);  
        html = html.replace("Node", "Admin").replace("Js", "JavaScript");  
        res.send(html);  
    });  
});  
  
http.createServer(app).listen(app.get('port'), function(){  
    console.log('Express server listening on port ' + app.get('port'));  
});
```

Nous affichons le code HTML de la vue d'origine dans la console du serveur, puis nous modifions le nom et le prénom comme indiqué. Le nouveau code HTML est alors retourné vers le navigateur au moyen de `res.send()`. Les nouvelles valeurs s'affichent alors dans celui-ci.

Dans la console du serveur, on affiche le code HTML d'origine :

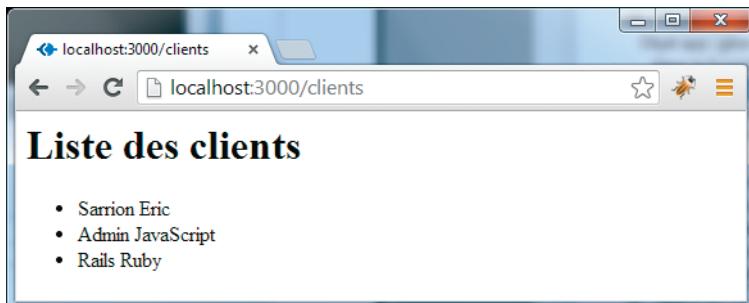
Figure 17–6
Code HTML d'origine

```
C:\Users\Eric\Documents\Node.js\testexpress>node app.js
Express server listening on port 3000

<h1>Liste des clients</h1>
<ul>
  <li>Sarrion Eric</li>
  <li>Node Js</li>
  <li>Rails Ruby</li>
</ul>
GET /clients 200 447ms - 112b
```

Dans l'écran du navigateur, on affiche le code HTML modifié :

Figure 17–7
Page affichée dans le navigateur, contenant le code HTML modifié



Partager des objets avec les vues

Express permet de transmettre des paramètres aux vues, comme nous l'avons fait dans l'exemple précédent. Il peut de plus utiliser des variables globales qui sont utilisables dans les vues.

Le positionnement d'un objet global se fait au moyen de la méthode `app.set(name, value)`, dans laquelle `name` est le nom de l'objet et `value` représente sa valeur.

La récupération dans la vue s'effectue au moyen de l'objet `settings` défini par Express. Les propriétés de cet objet correspondent aux noms des objets `name` positionnés par `app.set(name, value)`.

Voici un exemple d'utilisation, dans lequel on positionne en global le titre de la page et le texte associé à un lien permettant le retour à la page d'accueil, que l'on insère en bas de page.

Utiliser app.set() pour partager des variables avec les vues

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'jade');  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.favicon());  
app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
    app.use(express.errorHandler());  
}  
  
app.locals.pretty = true ;  
  
app.get("/clients", function(req, res) {  
    app.set("infos", {  
        titre : "Liste des clients",  
        bottom : "Revenir à la page d'accueil"  
    });  
    app.render("clients.jade", { clients : [  
        { nom : "Sarrion", prenom : "Eric" },  
        { nom : "Node", prenom : "Js" },  
        { nom : "Rails", prenom : "Ruby" }  
    ]}, function(err, html) {  
        console.log(html);  
        html = html.replace("Node", "Admin").replace("Js", "JavaScript");  
        res.send(html);  
    });  
});  
  
http.createServer(app).listen(app.get('port'), function(){  
    console.log('Express server listening on port ' + app.get('port'));  
});
```

Nous définissons l'objet `infos` qui contient les propriétés `titre` et `bottom`. Cet objet est utilisé dans la vue `clients.jade` de la façon suivante :

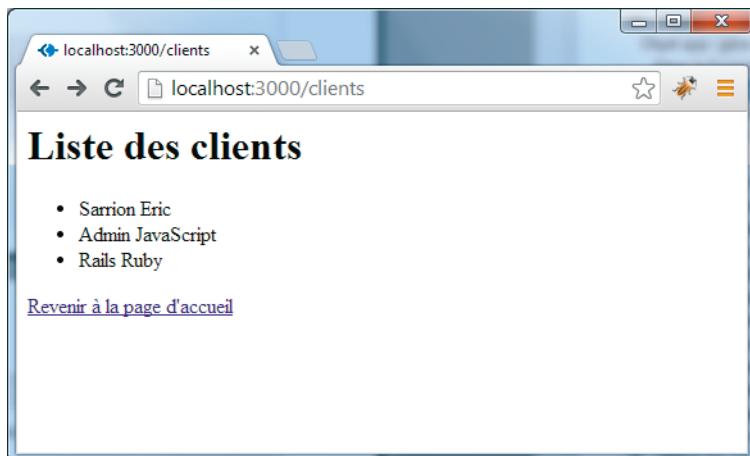
Utiliser l'objet `settings` pour récupérer des valeurs dans la vue

```
h1 #{settings.infos.titre}
ul
  each client in clients
    li= client.nom + " " + client.prenom
a(href="/") #{settings.infos.bottom}
```

Les propriétés `titre` et `bottom` de l'objet `infos` sont accessibles dans la vue au moyen de `settings.infos.titre` et `settings.infos.bottom`.

Figure 17–8

Transmission d'informations à la vue par `app.set()`



Objet `req` : gérer la requête reçue

L'objet `req` est de classe `http.IncomingMessage` définie par Node. Express a ajouté à l'objet `req` certaines propriétés permettant d'obtenir des informations sur la requête reçue, telles que :

- informations transmises par l'utilisateur (par exemple, les saisies effectuées dans un formulaire) ;
- informations sur la route utilisée par la requête (adresse IP, protocole utilisé, etc.)

Examinons ces deux types d'informations dans les sections suivantes.

Récupérer les informations transmises par les utilisateurs

Dans les chapitres précédents, nous avons utilisé différents mécanismes pour récupérer des données transmises par les utilisateurs et qui étaient positionnées dans les objets `req.query` ou `req.body`.

Tableau 17–3 Objets `req.query` et `req.body`

Objet	Signification
<code>req.query</code>	Lorsque les données récupérées de l'utilisateur sont transmises en mode <code>GET</code> , <code>req.query</code> est un objet contenant ces données. Par exemple, <code>req.query.nom</code> vaut <code>"Sarrion"</code> si on a transmis <code>nom=Sarrion</code> dans l'URL. Le middleware <code>express.query()</code> doit être utilisé pour que cela fonctionne.
<code>req.body</code>	Lorsque les données récupérées de l'utilisateur sont transmises en mode <code>POST</code> , <code>req.body</code> est un objet contenant ces données. Par exemple, <code>req.body.nom</code> vaut <code>"Sarrion"</code> si on a transmis <code>nom=Sarrion</code> dans l'en-tête HTTP. Le middleware <code>express.bodyParser()</code> doit être utilisé pour que cela fonctionne.

On a également vu qu'il était possible de transmettre des données dans des variables lors de l'écriture de la route. Les valeurs de ces variables sont dans l'objet `req.params`, chaque variable étant une propriété de cet objet.

Tableau 17–4 Objet `req.params`

Objet	Signification
<code>req.params</code>	Objet contenant les variables transmises dans l'URL de la route. Par exemple, si on utilise la route <code>POST /clients/:id</code> , la valeur de la variable <code>:id</code> sera accessible dans <code>req.params.id</code> . Aucun middleware n'est nécessaire pour que cela fonctionne.

En plus de ces objets, Express définit la méthode `req.param(name)` sur l'objet `req`. Cette méthode permet de rechercher quelle est la valeur de la propriété `name` définie dans `req.params` (en premier), puis `req.body` (en deuxième) ou `req.query` (en dernier). Si la propriété `name` n'est pas trouvée dans l'un de ces trois objets, la méthode `req.param()` retourne `undefined`.

Tableau 17–5 Méthode `req.param(name)`

Méthode	Signification
<code>req.param(name)</code>	Retourne la valeur de la propriété <code>name</code> dans l'un des trois objets (par ordre de priorité décroissante) ; <code>req.params</code> , <code>req.body</code> , puis <code>req.query</code> . Les middlewares <code>express.query()</code> et <code>express.bodyParser()</code> doivent être utilisés préalablement.

L'utilisation de `req.param(name)` permet de s'affranchir du mode de transmission des données (`GET`, `POST`, `PUT` ou `DELETE`) vu que la méthode `req.param(name)` cherche dans les trois emplacements possibles successifs (`req.params`, `req.body` et `req.query`).

Récupérer les informations sur la route utilisée

Des propriétés sont disponibles sur l'objet `req` afin d'obtenir des informations sur la route utilisée.

Tableau 17–6 Propriétés associées à la route définie sur l'objet `req`

Propriété	Signification
<code>req.url</code>	URL utilisée, incluant la partie <code>query</code> située après le "?" (par exemple, <code>/clients?id=10</code>)
<code>req.ip</code>	Adresse IP de l'utilisateur
<code>req.path</code>	URL utilisée, sans la partie <code>query</code> (par exemple, <code>/clients</code>)
<code>req.method</code>	Type de la méthode HTTP utilisée (<code>GET</code> , <code>POST</code> , etc.)
<code>req.host</code>	Nom du serveur, ou adresse IP de celui-ci
<code>req.protocol</code>	Protocole utilisé (<code>http</code>)
<code>req.xhr</code>	Booléen indiquant si la requête provient d'une requête Ajax (<code>XMLHttpRequest</code> , en abrégé <code>xhr</code>).

Voici un exemple d'utilisation de ces propriétés.

Afficher les informations stockées dans l'objet `req`

```
/**
 * Module dependencies
 */

var express = require('express')
  , routes = require('./routes')
  , user = require('./routes/user')
  , http = require('http')
  , path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.logger('dev'));
```

```
app.use(express.query());
app.use(express.bodyParser());
app.use(express.favicon());
app.use(app.router);
app.use(express.methodOverride());
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

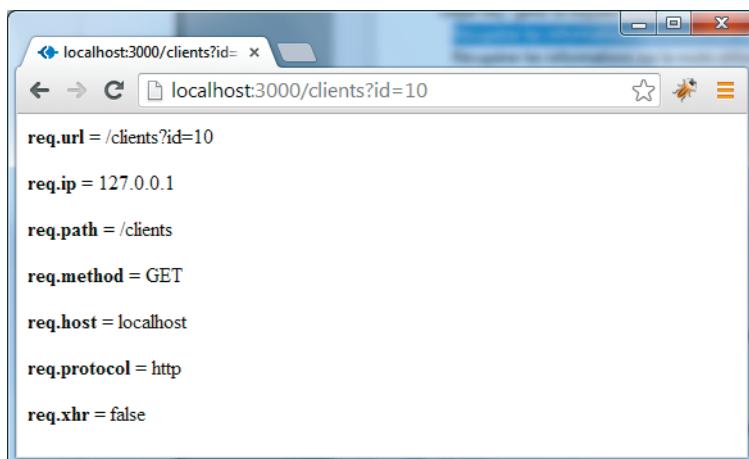
app.get("/clients", function(req, res) {
  var html = "";
  html += "<p><b>req.url</b> = " + req.url + "</p>";
  html += "<p><b>req.ip</b> = " + req.ip + "</p>";
  html += "<p><b>req.path</b> = " + req.path + "</p>";
  html += "<p><b>req.method</b> = " + req.method + "</p>";
  html += "<p><b>req.host</b> = " + req.host + "</p>";
  html += "<p><b>req.protocol</b> = " + req.protocol + "</p>";
  html += "<p><b>req.xhr</b> = " + req.xhr + "</p>";
  res.send(html);
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Si on utilise l'URL `http://localhost:3000/clients?id=10` :

Figure 17–9

Composants de l'objet req



Objet res : gérer la réponse à envoyer

L'objet `res` sert à retourner au navigateur la réponse du serveur. Le chapitre 16 est consacré à ce sujet, et nous n'avons rien de plus à ajouter ici.

18

Créer les vues avec EJS

On a vu dans les chapitres précédents comment écrire des vues à l'aide d'un langage spécialisé appelé JADE. Express l'utilise par défaut, mais il est possible d'en utiliser d'autres. Le langage utilisé dans les vues est connu par Express :

- au moyen de l'extension du fichier de la vue (par exemple, `.jade`, `.ejs`, etc.) ;
- grâce à l'instruction `app.set("view engine", format)`, qui permet dans ce cas de ne pas indiquer l'extension lors de l'utilisation de la vue par la méthode `res.render(name)`.

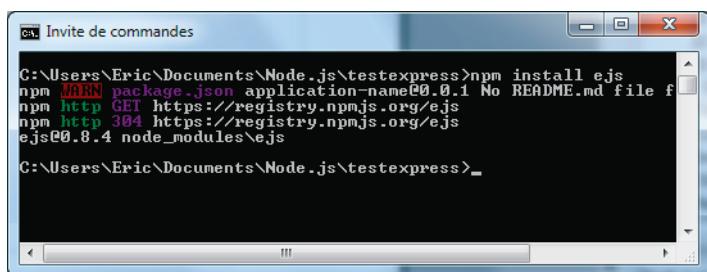
Dans ce chapitre, nous allons voir comment utiliser le langage EJS(*Embedded JavaScript*). Ce langage permet d'écrire les vues en incorporant directement du code JavaScript dans le code HTML de la vue, d'où le nom de ce langage.

Si vous vous demandez quel langage il vaut mieux utiliser pour écrire les vues (JADE, EJS, etc.), sachez que tous ces langages offrent les mêmes possibilités, à savoir écrire facilement des vues qui seront affichées dans le navigateur. EJS est peut-être plus simple que JADE pour une première approche, et c'est pourquoi nous l'étudions ici.

Installer EJS

EJS n'est pas livré en standard avec Express, contrairement à JADE. Il faut donc installer le module correspondant, au moyen de la commande `npm install ejs`.

Figure 18-1
Installation du module ejs



Une fois ce module installé, les fichiers d'extension EJS seront reconnus par Express.

Une première vue avec EJS

Pour se familiariser avec EJS, écrivons une vue qui affiche les nombres de 0 à 9 au moyen d'une boucle `for`. Le code du fichier `app.js` est le même que celui utilisé précédemment, à savoir une route `GET /clients` qui active la vue `clients.ejs`.

Fichier app.js

```
/**
 * Module dependencies
 */

var express = require('express')
, routes = require('./routes')
, user = require('./routes/user')
, http = require('http')
, path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.logger('dev'));
app.use(express.query());
app.use(express.bodyParser());
app.use(express.favicon());
app.use(app.router);
app.use(express.methodOverride());
app.use(express.static(path.join(__dirname, 'public')));
```

```
// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get("/clients", function(req, res) {
  res.render("clients.ejs");
});

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Fichier clients.ejs (dans le répertoire views)

```
<h1> Nombres de 0 à 9 </h1>
<% for (var i = 0; i < 10; i++) { %>
<%= i %> <br>
<% } %>
```

Le code HTML de la vue est inséré dans la page sans modification. En revanche, le code JavaScript est entouré des balises `<%` et `%>`, ceci indiquant d'évaluer le code JavaScript figurant entre ces balises. Si on souhaite afficher une valeur ou un résultat, il suffit de l'entourer entre `<%=` et `%>`.

Notez que l'accolade ouvrante doit se trouver sur la même ligne que l'instruction `for`, sinon une erreur de syntaxe se produit.

Figure 18–2
Affichage des nombres
de 0 à 9 avec EJS



Les nombres de 0 à 9 se sont affichés dans la page, chacun étant séparé du suivant par un retour à la ligne effectué grâce à la balise
.

Transmettre des paramètres à la vue

Plutôt que d'afficher une liste de nombres, affichons une liste de clients se trouvant dans le tableau `clients`.

Transmettre un tableau de clients à la vue

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'ejs');  
app.use(express.logger('dev'));  
app.use(express.query());  
app.use(express.bodyParser());  
app.use(express.favicon());  
app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
  app.use(express.errorHandler());  
}  
  
app.get("/clients", function(req, res) {  
  res.render("clients.ejs", { clients : [  
    { nom : "Sarrion", prenom : "Eric" },  
    { nom : "Node", prenom : "Js" },  
    { nom : "Rails", prenom : "Ruby" }  
  ]});  
});
```

```
    ]});  
});  
  
http.createServer(app).listen(app.get('port'), function(){  
  console.log('Express server listening on port ' + app.get('port'));  
});
```

La méthode `res.render(name, obj)` permet d'indiquer un paramètre `obj` pour lequel on définit ici la propriété `clients`, qui est un tableau de clients sous forme d'objets `{ nom, prenom }`. C'est la même syntaxe que celle utilisée pour transmettre des paramètres à des vues écrites en JADE.

Le fichier de la vue récupère ces informations et les affiche de la façon suivante.

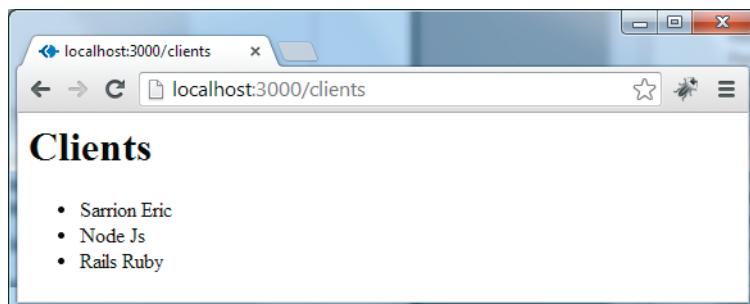
Afficher la liste de clients avec EJS (fichier `clients.ejs` dans le répertoire `views`)

```
<h1> Clients </h1>  
<ul>  
  <% clients.forEach(function(client) { %>  
  <li><%= client.nom %> <%= client.prenom %></li>  
  <% }); %>  
</ul>
```

L'objet `clients` est utilisé directement dans la vue, en tant que tableau JavaScript. Dès que l'on souhaite afficher du code HTML, on ferme la balise du code JavaScript, que l'on réouvre par la suite pour effectuer de nouveaux traitements JavaScript.

Figure 18–3

Transmission de données
à une vue EJS



Pour transmettre à la vue un objet plus complexe, il suffit d'ajouter de nouvelles propriétés à l'objet transmis. Par exemple, transmettons le titre affiché dans la fenêtre.

Transmettre le titre de la fenêtre en paramètre à la vue

```
/**  
 * Module dependencies  
 */  
  
var express = require('express')  
, routes = require('./routes')  
, user = require('./routes/user')  
, http = require('http')  
, ejs = require('ejs')  
, path = require('path');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'ejs');  
app.use(express.logger('dev'));  
app.use(express.query());  
app.use(express.bodyParser());  
app.use(express.favicon());  
app.use(app.router);  
app.use(express.methodOverride());  
app.use(express.static(path.join(__dirname, 'public')));  
  
// development only  
if ('development' == app.get('env')) {  
    app.use(express.errorHandler());  
}  
  
app.get("/clients", function(req, res) {  
    res.render("clients.ejs", {  
        titre : "Liste des clients",  
        clients : [  
            { nom : "Sarrion", prenom : "Eric" },  
            { nom : "Node", prenom : "Js" },  
            { nom : "Rails", prenom : "Ruby" }  
        ]});  
};  
  
console.log(ejs);  
  
http.createServer(app).listen(app.get('port'), function(){  
    console.log('Express server listening on port ' + app.get('port'));  
});
```

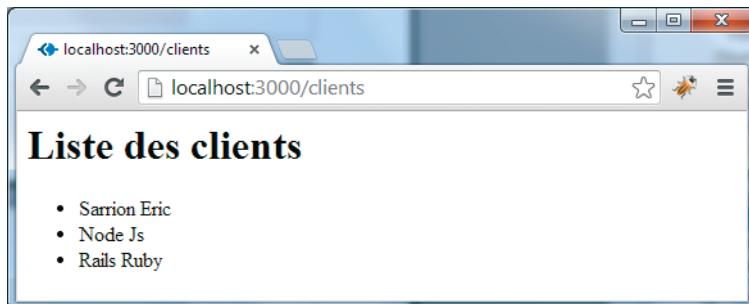
La vue exploite le titre transmis en utilisant directement le nom de la propriété (ici, `titre`).

Afficher le titre de la fenêtre transmis en paramètre (fichier clients.ejs dans le répertoire views)

```
<h1> <%= titre %> </h1>
<ul>
  <% clients.forEach(function(client) { %>
    <li><%= client.nom %> <%= client.prenom %></li>
  <% }); %>
</ul>
```

Figure 18–4

Affichage d'un titre transmis en paramètre



Remarquons que, selon le format des vues utilisé (EJS ou JADE), seul le code de la vue est différent, tandis que le code de la route (écrit ici dans `app.js`) est le même quel que soit le format des vues. Ceci permet d'écrire un code dans `app.js` indépendamment du format des vues.

De plus, comme nous l'avions vu dans le chapitre précédent (section « Récupérer et modifier le code HTML généré par une vue »), il est possible de modifier le code HTML généré par la vue en utilisant une fonction de callback en dernier paramètre de la méthode `app.render()`.

Cas pratique : utiliser plusieurs vues dans une application

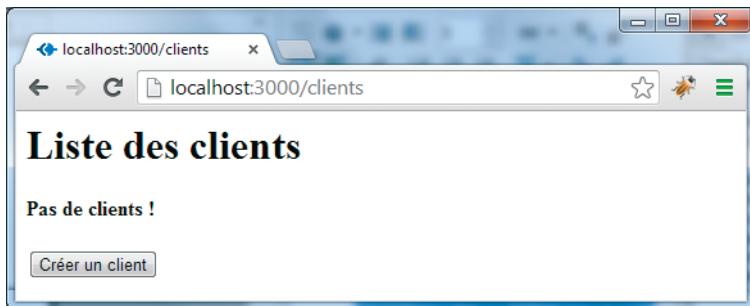
On souhaite ici montrer comment utiliser plusieurs vues dans une application gérant un ensemble de clients. Les clients sont stockés en mémoire sur le serveur car pour l'instant, nous ne savons pas les mémoriser dans une base de données. L'utilisation d'une base de données MongoDB avec Node dans une application Express sera étudiée dans la partie 3, chapitre « Utiliser la base de données MongoDB avec Node ».

Cinématique de l'application

On souhaite gérer une liste de clients, identifiés ici par leur nom et leur prénom. Les quatre opérations traditionnelles sont possibles, à savoir créer, lire, modifier et supprimer un client.

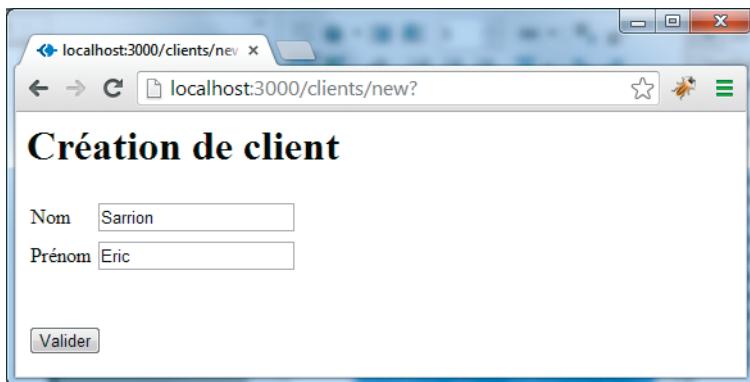
Au début, aucun client n'est présent, on peut donc seulement en créer un.

Figure 18–5
Vue contenant la liste des clients (vide pour l'instant)



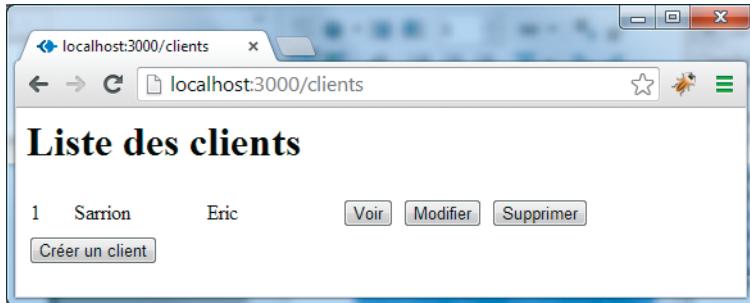
Après avoir cliqué sur le bouton Crée un client, un formulaire s'affiche.

Figure 18–6
Vue associée à la création d'un client



Remplissons les champs Nom et Prénom comme indiqué, puis validons. La page d'accueil s'affiche de nouveau, elle contient maintenant ce nouveau client.

Figure 18–7
Vue affichée lorsqu'un client a été créé.



Le chiffre situé en face de chaque nom représente une clé unique d'identification (clé primaire) gérée automatiquement par notre programme. Elle commence à 1 et elle

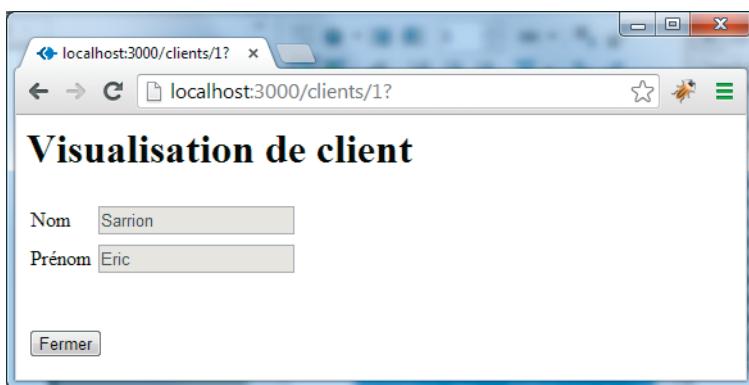
Ce document est la propriété exclusive de Alexandre Mbiam (alexmbiam@gmail.com) - 15 avril 2016 à 11:53

est réinitialisée à chaque démarrage du serveur, tout comme la liste des clients affichés. L'utilisation d'une base de données permettra de garder ces informations persistantes, même en cas de redémarrage du serveur.

Après le nom et le prénom de chaque client, figurent trois boutons permettant d'effectuer les actions de base sur chacun des clients : Voir, Modifier ou Supprimer.

Si on clique sur le bouton Voir :

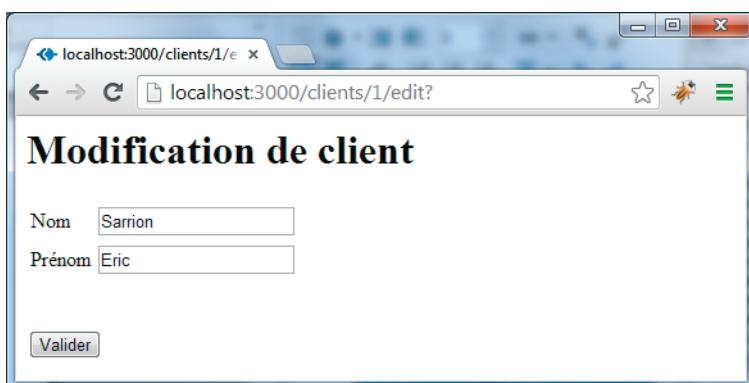
Figure 18–8
Visualisation d'un client sans modification possible



Le formulaire de visualisation affiche le client correspondant, sans possibilité de le modifier.

Si on clique sur le bouton Modifier :

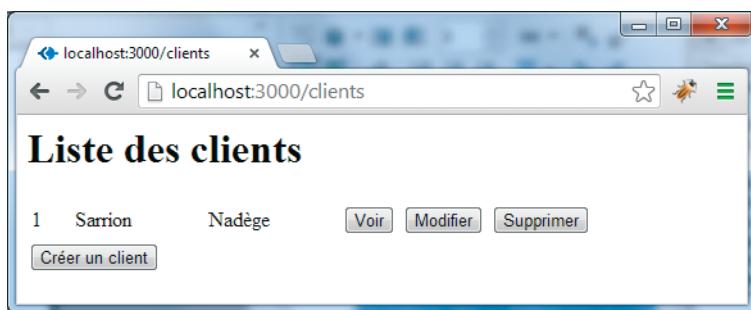
Figure 18–9
Visualisation d'un client avec modification possible



Le formulaire est presque le même que celui permettant de visualiser le client, mais on peut maintenant modifier le nom et le prénom. Par exemple, remplaçons le prénom Eric par Nadège, puis cliquons sur Valider.

Figure 18-10

Affichage de la liste des clients avec le client modifié

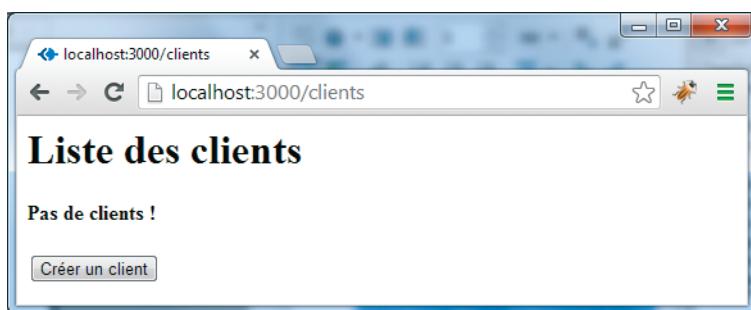


Le nouveau prénom a remplacé l'ancien.

Si on clique enfin sur le bouton Supprimer :

Figure 18-11

Suppression du client, la liste redevenant vide.



Le client correspondant a été supprimé, donc la liste s'affiche maintenant vide.

Programmes de l'application

Cet exemple est un bon exercice pour utiliser le module `express-resource` que nous avions introduit au chapitre 15, « Routage des requêtes avec Express », section « Organiser l'écriture des routes en utilisant REST ». Ce module permet d'organiser facilement l'accès à une ressource (ici, des clients) et d'effectuer les quatre opérations classiques (CRUD).

Le programme sera organisé de la façon suivante.

- Un programme principal qui sera le fichier `app.js`, incluant les différents middlewares nécessaires.
- Un fichier `clients.js` qui contiendra les routes du programme. Ce fichier sera inclus dans le fichier `app.js`.

- Différents fichiers associés aux vues de l'application, situés dans le répertoire `views/clients`, de façon à écrire des vues spécifiques gérant les clients.

Fichier app.js

```
/**  
 * Module dependencies  
 */  
  
var express = require('express');  
var http = require('http');  
var path = require('path');  
var util = require('util');  
var resource = require('express-resource');  
  
var app = express();  
  
// all environments  
app.set('port', process.env.PORT || 3000);  
app.set('views', __dirname + '/views');  
app.set('view engine', 'ejs');  
app.use(express.favicon());  
app.use(express.logger('dev'));  
app.use(express.bodyParser());  
app.use(express.methodOverride());  
app.use(app.router);  
app.use(express.static(path.join(__dirname, 'public')));  
app.use(express.static(path.join(__dirname, 'forms')));  
  
// development only  
if ('development' == app.get('env')) {  
    app.use(express.errorHandler());  
}  
  
app.resource("clients", require("./clients.js"));  
  
http.createServer(app).listen(app.get('port'), function(){  
    console.log('Express server listening on port ' + app.get('port'));  
});
```

Nous avons mis en évidence les instructions principales de ce fichier, à savoir l'inclusion du module `express-resource` et son association avec le fichier `clients.js` contenant les routes. L'inclusion du middleware `methodOverride` est également importante car elle permet de définir l'attribut `method` des formulaires aux valeurs `PUT` et `DELETE` en plus de `GET` et `POST` (voir le chapitre 13, « Utiliser les middlewares définis dans Connect »).

Le fichier `clients.js` contenant la définition des routes est le suivant.

Fichier clients.js (dans le répertoire principal de l'application, même niveau que app.js)

```
var clients = [];  
  
exports.index = function(req, res) {  
    res.render("clients/clients.ejs", { clients : clients });  
};  
  
exports.new = function(req, res) {  
    res.render("clients/new.ejs");  
};  
  
exports.create = function(req, res) {  
    var nom = req.param("nom");  
    var prenom = req.param("prenom");  
    var id = clients.length ? clients[clients.length - 1].id + 1 : 1;  
    var client = { id : id, nom : nom, prenom : prenom };  
    clients.push(client);  
    res.redirect("/clients");  
};  
  
exports.show = function(req, res) {  
    var id = req.params.client;  
    var client = clients.filter(function(client) {  
        return (client.id == id);  
    })[0];  
    res.render("clients/show.ejs", { client : client });  
};  
  
exports.edit = function(req, res) {  
    var id = req.params.client;  
    var client = clients.filter(function(client) {  
        return (client.id == id);  
    })[0];  
    res.render("clients/edit.ejs", { client : client });  
};  
  
exports.update = function(req, res) {  
    var nom = req.param("nom");  
    var prenom = req.param("prenom");  
    var id = req.params.client;  
    clients.forEach(function(client) {  
        if (client.id == id) {  
            client.nom = nom;  
            client.prenom = prenom;  
        }  
    });  
    res.redirect("/clients");  
};
```

```
exports.destroy = function(req, res) {
  var id = req.params.client;
  clients = clients.filter(function(client) {
    return (client.id != id);
  });
  res.redirect("/clients");
};
```

La méthode `filter()`, parfois utilisée sur le tableau `clients`, permet de retourner un nouveau tableau contenant uniquement les éléments pour lesquels la fonction de callback retourne `true`. Par exemple, pour rechercher un client dont l'identifiant `id` est précisé, on récupère le client lui-même en accédant à l'indice 0 du tableau retourné (vu que le tableau retourné contient un seul élément).

Ces quatre vues sont situées dans le répertoire des vues de l'application, identifié par l'instruction `app.set('views', __dirname + '/views')` incluse dans `app.js`. De plus, comme les méthodes `res.render()` utilisées dans `app.js` préfixent chaque nom de fichier par la chaîne "`clients/`", cela permet d'avoir toutes les vues de l'application situées dans le répertoire `views/clients`.

Le mécanisme interne à `express-resource` associe les routes aux méthodes définies dans le fichier `clients.js`. Nous avions expliqué ceci dans le chapitre 15, « Routage des requêtes avec Express », section « Organiser l'écriture des routes en utilisant REST ».

Le tableau 18-1 présente chacune de ces méthodes.

Tableau 18-1 Méthodes définies dans clients.js

Méthode	Signification
<code>exports.index</code>	On affiche la liste des clients. Pour cela, on affiche la vue <code>clients.ejs</code> à laquelle on transmet le tableau des clients déjà présents (vide si aucun).
<code>exports.new</code>	On affiche la vue de création de nouveau client. Lors du clic sur le bouton de validation, la méthode <code>exports.create</code> sera activée pour créer le client dans la mémoire du serveur.
<code>exports.create</code>	Méthode appelée pour créer un nouveau client. On attribue à chaque client un identifiant <code>id</code> unique incrémenté de 1 à chaque création de client.
<code>exports.show</code>	Méthode appelée lorsqu'on souhaite visualiser un client sans le modifier. Elle affiche la vue <code>show.ejs</code> permettant de visualiser le client transmis en paramètre, repéré par son identifiant <code>id</code> . Pour afficher le nom et le prénom, on recherche dans la liste des clients celui qui possède le même identifiant <code>id</code> . Le client trouvé est transmis en paramètre à la vue <code>show.ejs</code> .
<code>exports.edit</code>	Méthode appelée pour modifier le client transmis dans la route (variable <code>:id</code>). Elle affiche la vue <code>edit.ejs</code> . Cette méthode est similaire à la précédente, mais la vue correspondante permet de modifier les informations affichées, contrairement à la précédente. Après validation dans la vue, on devra activer la méthode <code>update()</code> qui modifie le client avec les valeurs transmises.

Tableau 18–1 Méthodes définies dans clients.js (suite)

Méthode	Signification
<code>exports.update</code>	Méthode effectuant la modification du nom ou prénom du client. Une fois effectuée, on retourne à l'affichage de la liste des clients, avec les valeurs modifiées.
<code>exports.destroy</code>	Méthode effectuant la suppression d'un client dans la liste en mémoire. Elle est appelée directement depuis la vue principale, en cliquant sur le bouton Supprimer situé sur la ligne d'un client.

Cette application possède les quatre vues suivantes :

- `clients.ejs` pour afficher la liste des clients ;
- `new.ejs` permet d'afficher les informations dans le but de créer un client ;
- `show.ejs` pour afficher les informations sur le client, sans modification possible ;
- `edit.ejs` est similaire à `show.ejs`, mais permet de modifier le client et de valider sa modification.

Fichier clients.ejs (dans views/clients)

```
<h1> Liste des clients </h1>
<% if (!clients.length) { %>
  <b> Pas de clients ! </b>
  <br><br>
<% } else { %>
  <table>
    <% clients.forEach(function(client) { %>
      <tr>
        <td width=30><%= client.id %></td>
        <td width=100><%= client.nom %></td>
        <td width=100><%= client.prenom %></td>
        <td>
          <form style="display:inline" action="/clients/<%= client.id %>">
            <input type="submit" value="Voir" />
          </form>
        </td>
        <td>
          <form style="display:inline" action="/clients/<%= client.id %>/edit">
            <input type="submit" value="Modifier" />
          </form>
        </td>
        <td>
          <form style="display:inline" action="/clients/<%= client.id %>">
            <input type="post" />
            <input type="submit" value="Supprimer" />
            <input type="hidden" name="_method" value="delete" />
          </form>
        </td>
      </tr>
    <% } %>
  </table>
<% } %>
```

```
</tr>
<% } ); %>
</table>
<% } %>

<form action="/clients/new">
  <input type="submit" value="Créer un client" />
</form>
```

La liste des clients est transmise à la vue, qui l'affiche au moyen d'une boucle `forEach()`. Les boutons d'action sont insérés chacun dans un formulaire, afin d'activer la route indiquée dans l'attribut `action` du formulaire. Le style `display` positionné à `inline` dans le formulaire permet de mettre les boutons successivement sur une seule ligne.

Le bouton Supprimer est particulier, car on souhaite activer la route `DELETE /clients/:id`. On utilise le middleware `methodOverride` pour activer une route possédant la méthode `DELETE`, en l'indiquant dans un champ caché du formulaire. La méthode `POST` est alors ajoutée dans le formulaire afin que ce champ caché soit transmis correctement.

Fichier `new.ejs` (dans `views/clients`)

```
<h1> Création de client <h1>

<form action="/clients" method="post">

  <table>
    <tr>
      <td>Nom</td>
      <td><input type="text" name="nom" /></td>
    </tr>
    <tr>
      <td>Prénom</td>
      <td><input type="text" name="prenom" /></td>
    </tr>
  </table>

  <br>
  <input type="submit" value="Valider" />

</form>
```

La méthode `POST` est activée afin que la méthode `create()` soit appelée dans le fichier `clients.js`, via la route `POST /clients`.

Voici maintenant la description du fichier `new.ejs` permettant de créer un nouveau client.

Fichier show.ejs (dans views/clients)

```
<h1> Visualisation de client <h1>

<form action="/clients" method="get">

<table>
<tr>
<td>Nom</td>
<td><input type="text" name="nom" disabled value = "<%=client.nom %>" /></td>
</tr>
<tr>
<td>Prénom</td>
<td><input type="text" name="prenom" disabled value = "<%=client.prenom %>" /></td>
</tr>
</table>

<br>
<input type="submit" value="Fermer" />

</form>
```

Fichier edit.ejs (dans views/clients)

```
<h1> Modification de client <h1>

<form action="/clients/<%=client.id %>" method="post">

<table>
<tr>
<td>Nom</td>
<td><input type="text" name="nom" value = "<%=client.nom %>" /></td>
</tr>
<tr>
<td>Prénom</td>
<td><input type="text" name="prenom" value = "<%=client.prenom %>" /></td>
</tr>
</table>

<br>
<input type="hidden" name="_method" value="put" />
<input type="submit" value="Valider" />

</form>
```

On voit dans la vue `edit.ejs` l'utilisation du middleware `methodOverride`, grâce à l'utilisation du champ caché indiquant le nom de la méthode HTTP utilisée (ici, `PUT`).

Ajouter des styles dans les vues

Il est évidemment possible de styler les éléments HTML présents dans nos vues. On peut insérer les styles dans chacune des vues de notre application, au moyen de l'une des deux méthodes suivantes :

- en utilisant la balise `<style type="text/css">` et en définissant les styles à l'intérieur de celle-ci ;
- en incluant une feuille de styles au moyen de la balise `<link>`. Le fichier des styles devra se trouver dans un répertoire défini avec le middleware `static`, permettant ainsi d'y accéder directement sans définir de route. Express a créé pour cela le répertoire `public/stylesheets` dans lequel se trouve déjà le fichier `style.css`.

Mettons en œuvre cette seconde méthode. Ajoutons la balise `<link>` dans la vue `clients.ejs`, afin de redéfinir l'aspect de la liste des clients affichés.

Ajout de la balise `<link>` dans la vue `clients.ejs`

```
<link rel="stylesheet" href="/stylesheets/style.css" />

<h1> Liste des clients </h1>
<% if (!clients.length) { %>
  <b> Pas de clients ! </b>
  <br><br>
<% } else { %>
  <table>
    <% clients.forEach(function(client) { %>
      <tr>
        <td width=30><%= client.id %></td>
        <td width=100><%= client.nom %></td>
        <td width=100><%= client.prenom %></td>
        <td>
          <form style="display:inline" action="/clients/<%= client.id %>">
            <input type="submit" value="Voir" />
          </form>
        </td>
        <td>
          <form style="display:inline" action="/clients/<%= client.id %>/edit">
            <input type="submit" value="Modifier" />
          </form>
        </td>
        <td>
          <form style="display:inline" action="/clients/<%= client.id %>" method="post">
            <input type="submit" value="Supprimer" />
            <input type="hidden" name="_method" value="delete" />
          </form>
        </td>
      </tr>
    <% } >
  </table>
</div>
```

```

<% }); %>
</table>
<% } %>

<form action="/clients/new">
  <input type="submit" value="Créer un client" />
</form>

```

Nous incluons le fichier `style.css` se trouvant dans le répertoire `stylesheets`, lui-même situé dans le répertoire `public` de l'application Express. Le répertoire `public` est accessible car il est défini par le middleware `static` utilisé par `app.js`.

Définissons de nouveaux styles dans le fichier `style.css` :

Fichier `style.css` (dans le répertoire `public/stylesheets`)

```

body {
  background-color : gainsboro;
}
input[type=submit] {
  font-family : arial;
  font-size : 15px;
  font-weight : bold;
}
td {
  color : red;
  font-family : arial;
  font-size : 15px;
  font-weight : bold;
}

```

L'aspect de la vue `clients.ejs` est maintenant le suivant.

Figure 18-12

Utilisation des styles CSS dans les vues



Pour utiliser des fonctionnalités de styles plus approfondies (par exemple, le concept de *layouts*), utilisez le module `ejs-locals` disponible sur `npm`.

TROISIÈME PARTIE

Utiliser la base de données **MongoDB** avec **Node**

19

Introduction à MongoDB

Node peut utiliser les bases de données traditionnelles, par exemple MySQL. Toutefois, on préfère l'associer à une base de données de type NoSQL comme MongoDB, ce qui signifie qu'on n'utilisera plus de requêtes SQL pour interagir.

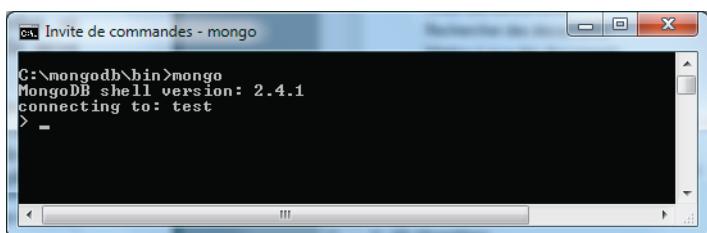
MongoDB est une base de données robuste orientée document, dans laquelle on stocke ce que l'on appelle des « documents », c'est-à-dire une structure de données de n'importe quel type, comme les informations écrites sur une feuille de papier (qui équivaut alors à un document). Plusieurs feuilles de papier, correspondant ainsi à plusieurs documents, forment ce qui est appelé une « collection ».

Installer MongoDB

MongoDB est accessible sur le site <http://www.mongodb.org>. Téléchargez la version qui correspond à votre système d'exploitation. Utilisez ensuite la procédure d'installation indiquée sur la page <http://docs.mongodb.org/manual/installation>, qui fournit tous les détails pour chacune des plates-formes disponibles (Windows, Linux, etc.).

Une fois MongoDB installé, ouvrez un interpréteur de commandes et lancez l'utilitaire `mongo` se trouvant dans le répertoire `bin` de MongoDB. La fenêtre de la figure 19-1 s'affiche alors, permettant d'agir sur les documents et collections sauvegardés dans MongoDB.

Figure 19–1
Lancement de l'utilitaire mongo



Dans la suite de ce chapitre, nous allons interagir avec MongoDB au moyen de l'utilitaire `mongo` précédemment lancé.

Documents et collections

Les documents et les collections sont les éléments principaux d'une base de données MongoDB. Expliquons plus en détail ces deux notions.

Définir un document

Un document est un ensemble de données, chacune des données pouvant avoir un type différent des autres. Un document correspondrait, pour une base de données de type SQL, à un enregistrement d'une table. En effet, un enregistrement peut contenir diverses données, chacune d'entre elles pouvant être de différents types. Par exemple, le nom et le prénom sont des chaînes de caractères, tandis que l'âge d'un client est un entier, et sa date de naissance est un objet `Date`.

Pour représenter un document, MongoDB utilise la notation JSON. Ainsi, un document contenant le nom et le prénom d'un client, pourrait s'écrire de la façon suivante.

Un document MongoDB décrivant le nom et le prénom d'un client

```
var client1 = { nom : "Sarrion", prenom : "Eric" };
```

Le nom et le prénom ont été associés à des valeurs précises. Un second document décrivant un autre client pourrait être :

Un autre document décrivant un second client

```
var client2 = { nom : "Node", prenom : "Js" };
```

Un document correspond donc à un objet JavaScript écrit sous forme JSON. On peut aussi l'écrire sous la forme suivante, car les espaces et les retours à la ligne peuvent être utilisés lors de son écriture.

Autre forme d'écriture du document

```
var client1 = {  
    nom : "Sarrion",  
    prenom : "Eric"  
};
```

Lors du stockage d'un document dans une base de données MongoDB, le format JSON est converti au format BSON, qui est une représentation binaire du format JSON. Ceci permet de compacter les informations stockées et ainsi de gagner de l'espace de stockage.

La valeur associée à une propriété dans un document peut être de n'importe quel type JavaScript, par exemple :

- une chaîne de caractères (comme celles définies précédemment) ;
- un nombre quelconque (entier ou flottant) ;
- un tableau ;
- une date ;
- un autre objet (c'est-à-dire un autre document).

On voit donc que l'on peut définir n'importe quel type de document à l'aide de la notation JSON.

Toutefois, MongoDB a défini des contraintes pour l'écriture de documents, en particulier concernant leur taille maximale. Celle-ci ne pourra pas excéder, au format compressé BSON, seize mégaoctets (16 Mo).

De plus, les noms des propriétés de l'objet ne peuvent pas commencer par le caractère "\$", ni contenir le caractère "..". Enfin, la propriété `_id` ne peut pas être utilisée car elle est réservée par MongoDB pour affecter une clé primaire unique à chaque document inséré dans la base de données.

Définir une collection

Un document ne peut être sauvegardé que dans une collection, qui sera donc un ensemble de documents. Les documents d'une collection peuvent être de différents types, contrairement aux enregistrements d'une table SQL qui sont tous du même format. Hormis cette différence entre ces deux formes de stockage, on peut considérer qu'une collection correspond à une table d'une base de données relationnelle.

Utiliser l'exécutable mongo

L'utilitaire `mongo` fourni par MongoDB permet d'exécuter des commandes MongoDB :

- soit directement dans l'interpréteur de commandes (*shell*) ouvert au moyen de la commande `mongo` (elle-même exécutée dans un interpréteur de commandes) ;
- soit en exécutant des fichiers JavaScript contenant des instructions MongoDB, au moyen de la commande `mongo nomfichier.js` tapée dans un interpréteur de commandes.

Ces deux façons de procéder produisent les mêmes résultats finaux. Les instructions (indiquées dans le fichier JavaScript) ou les commandes (saisies directement dans le shell) sont pratiquement identiques. Les seules différences proviennent du fait que le shell autorise de taper certaines commandes sous une forme simplifiée, avec une syntaxe non JavaScript.

Par exemple, pour lister les bases de données créées dans MongoDB, on peut écrire aussi bien dans le shell que dans un fichier JavaScript exécuté par la commande `mongo`, l'instruction `db.adminCommand("listDatabases")` qui retourne un objet contenant la liste des bases de données présentes dans MongoDB. L'instruction `show dbs` tapée directement dans le shell fournit les mêmes informations. Mais cette instruction ne fonctionne pas dans un programme JavaScript.

Exécution dans un shell

Exécutons l'instruction `db.adminCommand("listDatabases")` dans un shell ouvert par la commande `mongo`.

Figure 19-2

Affichage de la liste des bases de données présentes par l'utilitaire `mongo`

```

C:\Users\Eric\Documents\Node.js\testexpress>mongo
MongoDB shell version: 2.4.1
connecting to: test
> db.adminCommand("listDatabases")
{
  "databases" : [
    {
      "name" : "test",
      "totalSize" : 0,
      "ok" : 1
    }
  ]
}
>

```

Dans la fenêtre de la figure 19-2, nous avons lancé l'utilitaire `mongo`, qui ouvre un shell permettant de taper des commandes MongoDB. Remarquez que, par défaut, ce shell nous connecte directement à la base de données "`test`", vide lors de la première utilisation.

Nous avons ensuite demandé à lister les bases de données MongoDB présentes dans le système, au moyen de l'instruction `db.adminCommand("listDatabases")`. L'objet `db`

utilisé pour exécuter l'instruction `adminCommand("listDatabases")` est un objet global automatiquement créé par l'utilitaire `mongo`, et qui permet d'accéder aux instructions spécifiques à MongoDB.

Il nous est retourné un objet JavaScript contenant les propriétés `databases`, `totalSize` et `ok`. La propriété `databases` est un tableau contenant la liste des bases de données présentes dans le système, ici aucune. En effet, bien que le shell nous connecte directement à la base de données `test`, elle ne s'affiche pas dans la liste car étant vide, elle n'est pas prise en compte par MongoDB. Il faudra donc attendre d'insérer un document dans une collection de cette base de données pour qu'elle apparaisse dans le résultat de cette commande.

L'intérêt d'utiliser le shell associé à MongoDB est que cette instruction peut s'écrire de manière plus concise. En effet, plutôt que de taper cette ligne, on peut simplement taper la commande `show dbs` dans le shell.

Exécution dans un fichier JavaScript

Plutôt que d'utiliser le shell ouvert lors de la commande `mongo`, indiquons les instructions MongoDB dans un fichier JavaScript et exécutons-le au moyen de la commande `mongo`. Le fichier JavaScript peut se trouver dans n'importe quel répertoire, par exemple dans le répertoire `testmongo` créé pour cette occasion.

Fichier `testmongo.js` (dans le répertoire `testmongo`)

```
var dbs = db.adminCommand("listDatabases");
printjson(dbs);
```

L'utilitaire `mongo` permet d'afficher des résultats au moyen des deux instructions `print()` ou `printjson()`. Remarquez que nous ne sommes pas encore dans un programme Node, donc l'instruction traditionnelle de Node `console.log()` ne fonctionne pas ici.

L'association de MongoDB avec un programme Node sera étudiée dans le prochain chapitre. Nous étudions dans ce chapitre les fonctionnalités de MongoDB indépendantes d'un serveur.

Pour exécuter ce fichier JavaScript dans MongoDB, il suffit de l'indiquer dans la ligne de commandes en lançant la commande `mongo`. On tape donc la commande `mongo testmongo.js` dans un interpréteur de commandes.

Figure 19–3

Exécution d'un fichier
JavaScript par l'utilitaire mongo

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
{
  "databases" : [
  ],
  "totalSize" : 0,
  "ok" : 1
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

Les mêmes informations que celles obtenues au moyen du shell apparaissent dans la fenêtre de commandes, montrant la similitude des deux approches.

En revanche, l'instruction `show dbs` exécutée dans un fichier JavaScript provoque ici une erreur, car ce n'est pas une syntaxe JavaScript valide.

Voyons maintenant comment manipuler concrètement des documents dans une collection, à savoir effectuer les quatre opérations de base : création, lecture, mise à jour et suppression (CRUD). Mais avant, commençons par établir une connexion à la base de données.

Établir une connexion à la base de données

Dans les exemples précédents, nous avons constaté que notre programme (shell ou fichier JavaScript) provoquait une connexion automatique à la base de données `test` par défaut. L'objet `db` créé par MongoDB permet un accès direct à cette base de données `test` pour y effectuer nos opérations. D'où la question : « Comment effectuer une connexion à une autre base de données que celle utilisée par défaut ? ».

Il est bien entendu possible de définir autant de bases de données qu'on le souhaite. MongoDB fournit la méthode `connect(name)` permettant de se connecter à la base de données dont le nom est `name`.

On peut se connecter à une base de données qui n'existe pas encore. Elle sera visible dans la liste des bases de données uniquement lorsqu'un premier document aura été inséré dans une collection.

Par exemple, connectons-nous à une nouvelle base de données nommée `mydb`.

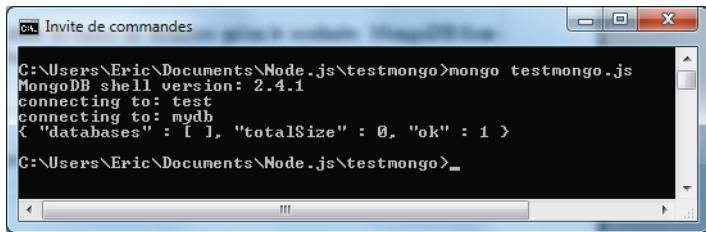
Connexion à la base de données mydb

```
db = connect("mydb");
var dbs = db.adminCommand("listDatabases");
printjson(dbs);
```

Nous nous connectons à la base de données `mydb`, puis nous affichons la liste des bases de données disponibles.

Figure 19–4

Connexion à la base de données `mydb`



```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
  "databases": [
    {
      "name": "mydb",
      "ok": 1,
      "totalSize": 0
    }
  ]
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

Nous voyons effectivement la connexion à la base de données par défaut `test`, puis la connexion à `mydb`. Cependant, l'affichage de la liste des bases de données au moyen de l'instruction `db.adminCommand("listDatabases")` ne produit aucun affichage, étant donné que ces bases de données sont vides pour l'instant. Nous créons donc un premier document dans la nouvelle base de données `mydb`.

Créer des documents

La création d'un document se fait dans une collection. Un document est un objet JavaScript, décrit le plus souvent au format JSON.

Une fois la connexion à la base de données effectuée (si cela est nécessaire, sinon on est par défaut connecté à la base de données `test`), on accède à la collection nommée `nomCollection` au moyen de l'objet `db.nomCollection`. Par exemple, la collection `clients` associée aux clients sera accessible par l'objet `db.clients`. La méthode `insert(doc)` définie par MongoDB sur chacune des collections permet d'insérer le document représenté par l'objet `doc` dans la collection. Ainsi, pour insérer un document `client` dans la collection `clients`, il suffira d'utiliser la méthode `db.clients.insert(client)`.

Tableau 19–1 Méthode `insert()`

Méthode	Signification
<code>db.nomCollection.insert(doc)</code>	Insère le document dans la collection indiquée.

Utilisons la méthode `insert()` pour insérer un client dans la collection `clients`.

Insertion d'un document client dans la collection clients

```
db = connect("mydb");

var dbs = db.adminCommand("listDatabases");
print("\nAvant insertion d'un document");
printjson(dbs);

var client1 = {
  nom : "Sarrion",
  prenom : "Eric"
};

db.clients.insert(client1);

var dbs = db.adminCommand("listDatabases");
print("\nApres insertion d'un document");
printjson(dbs);
```

L'objet `client1` est ici défini par les propriétés `nom` et `prenom`, valant respectivement "Sarrion" et "Eric". Quelle que soit la structure du document, celui-ci est inséré dans la collection au moyen de l'instruction `db.clients.insert(doc)`.

Figure 19-5

Insertion d'un document dans la collection clients

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
Avant insertion d'un document
{
  "databases" : [
    {}
  ],
  "totalSize" : 0,
  "ok" : 1
}
Apres insertion d'un document
{
  "databases" : [
    {
      "name" : "mydb",
      "sizeOnDisk" : 83886080,
      "empty" : false
    }
  ],
  "totalSize" : 83886080,
  "ok" : 1
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

On peut faire les constatations suivantes.

- Avant insertion du document dans la collection, aucune base de données n'est présente (la propriété `databases` de l'objet retourné par `db.adminCommand("listDatabases")` est un tableau vide).
- Après insertion du document dans la collection, la base de données `mydb` est créée (visible dans la propriété `databases`).

Ce document est la propriété exclusive de Alexandre Mbiam (alexmbiam@gmail.com) - 15 avril 2016 à 11:53

Si l'on souhaite insérer plusieurs documents, il suffit d'effectuer plusieurs appels à l'instruction `insert()`. Par exemple, insérons ci-dessous les deux clients Rails Ruby et Node Js.

Insertion de deux documents dans la collection clients

```
db = connect("mydb");

var client1 = {
  nom : "Rails",
  prenom : "Ruby"
};

db.clients.insert(client1);

var client2 = {
  nom : "Node",
  prenom : "Js"
};

db.clients.insert(client2);

var dbs = db.adminCommand("listDatabases");
print("\nAprès insertion d'un document");
printjson(dbs);
```

Suite à ces différentes instructions, la base de données `mydb` contient maintenant plusieurs documents dans la collection `clients`. Il est intéressant de savoir comment lire les documents stockés dans la base de données, ce que nous verrons dans la prochaine section.

Remarquons qu'on peut insérer dans une collection des documents ayant des formats différents. Contrairement à une base de données relationnelle, le format des enregistrements insérés dans la base de données n'a pas besoin d'être indiqué lors de la création de la collection et peut évoluer d'un document à l'autre.

Rechercher des documents

Une fois les documents insérés dans la base de données, il est possible d'effectuer des recherches afin de retourner ceux qui nous intéressent. La méthode `find()` utilisée sur la collection permet de les récupérer.

La méthode `find(query, projection)` prend les deux paramètres (optionnels) `query` et `projection`, que nous allons étudier dans les sections suivantes. Voyons comment les utiliser à travers des exemples.

Tableau 19–2 Méthode `find()`

Méthode	Signification
<code>db.nomCollection.find(query, projection)</code>	<p>Recherche les documents dans la collection indiquée, qui correspondent à l'éventuel objet <code>query</code> indiqué en paramètre, et les retourne dans un tableau (appelé « curseur »). Les clés rentrées dans les documents sont celles indiquées dans l'objet <code>projection</code> (par défaut, toutes les clés des documents trouvés sont rentrées si l'objet <code>projection</code> n'est pas indiqué).</p> <p>L'objet <code>query</code> permet de spécifier les documents qui seront sélectionnés dans la collection.</p> <p>L'objet <code>projection</code> permet d'indiquer les attributs des documents qui seront rentrés ou non, par exemple :</p> <ul style="list-style-type: none"> - <code>{ nom : 1 }</code> pour indiquer que l'on souhaite rentrer uniquement le nom ; - <code>{ nom : 0 }</code> pour indiquer que l'on souhaite tout rentrer sauf le nom. <p>On ne peut pas mixer les attributs ayant les valeurs 0 et 1. On ne peut donc pas écrire <code>projection { nom : 1, prenom : 0 }</code>. Le seul attribut qui peut s'utiliser avec la valeur 0 ou 1 avec les autres attributs est l'attribut <code>_id</code>.</p>
<code>db.nomCollection.findOne(query, projection)</code>	<p>Recherche uniquement le premier document qui satisfait les conditions spécifiées dans <code>query</code>. Les paramètres <code>query</code> et <code>projection</code> sont les mêmes que ceux explicités dans la méthode <code>find()</code> précédente.</p> <p>Si aucun document n'est trouvé, la méthode renvoie <code>null</code>.</p>

Rechercher tous les documents de la collection

L'utilisation la plus simple de la méthode `find()` consiste à ne spécifier aucun paramètre. Dans ce cas, elle recherche tous les documents de la collection.

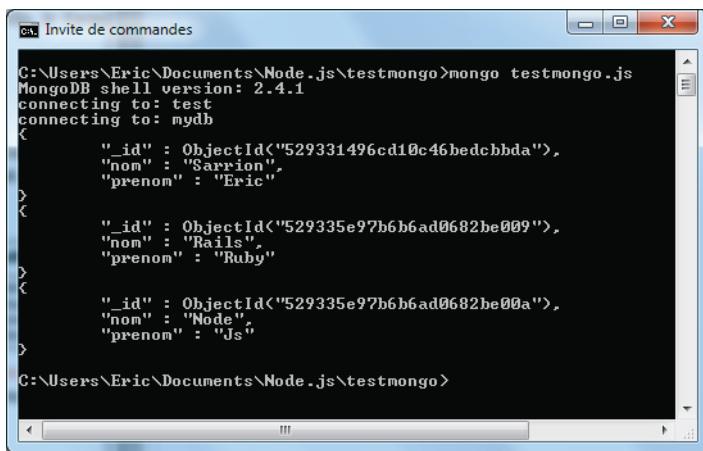
Rechercher tous les clients de la collection clients

```
db = connect("mydb");

var clients = db.clients.find();
clients.forEach(function(client) {
  printjson(client);
});
```

La méthode `find()` renvoie un tableau de tous les clients de la collection. Ce tableau est ensuite parcouru par la méthode `forEach()`, qui affiche chacun des clients au format JSON.

Figure 19–6
Affichage de tous les documents de la collection clients



```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("529331496cd10c46bedchbda"),
    "nom" : "Sarrion",
    "prenom" : "Eric"
}
{
    "_id" : ObjectId("529335e97b6b6ad0682be009"),
    "nom" : "Rails",
    "prenom" : "Ruby"
}
{
    "_id" : ObjectId("529335e97b6b6ad0682be00a"),
    "nom" : "Node",
    "prenom" : "Js"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Nous retrouvons chacun des clients précédemment insérés. En plus des champs `nom` et `prenom`, MongoDB a ajouté pour chacun des documents une clé nommée `_id`, qui sert à affecter une clé unique à chaque document de la collection. Ceci permet de distinguer deux documents qui posséderaient les mêmes informations. La clé `_id` est similaire à une clé primaire dans une base de données relationnelle.

Spécifier une condition AND

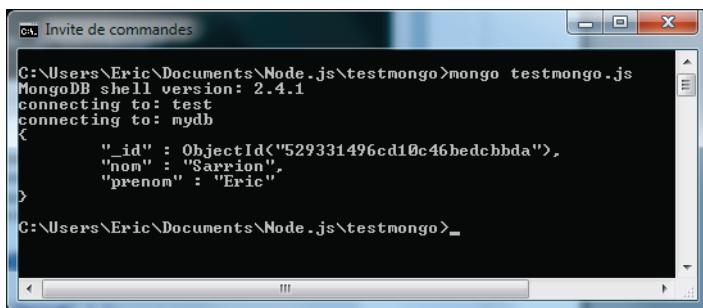
On peut rechercher les documents qui correspondent à des critères particuliers dans une collection. Pour cela, on indique dans l'objet `query` de la méthode `find(query, projection)`, les conditions que doivent satisfaire les documents recherchés.

Rechercher tous les clients ayant le nom "Sarrion"

```
db = connect("mydb");

var clients = db.clients.find( { nom : "Sarrion" } );
clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19–7
Recherche des documents ayant le nom "Sarrion"



```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("529331496cd10c46bedchbda"),
    "nom" : "Sarrion",
    "prenom" : "Eric"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

L'enregistrement ayant le nom "Sarrion" est retourné. Une seule condition `{ nom : "Sarrion" }` est ici exprimée. Pour spécifier plusieurs conditions reliées par AND, il suffit d'indiquer ces valeurs dans l'objet `query`.

Rechercher tous les clients ayant le nom "Sarrion" et le prénom "Eric"

```
db = connect("mydb");

var clients = db.clients.find( { nom : "Sarrion", prenom : "Eric" } );
clients.forEach(function(client) {
    printjson(client);
});
```

Le même document que précédemment sera ici retourné. En revanche, si vous spécifiez le même nom mais un autre prénom, aucun enregistrement ne sera retourné.

Une autre forme d'écriture est permise, en utilisant le mot-clé `$and`. Le programme précédent peut alors s'écrire :

Utiliser le mot-clé `$and` pour rechercher tous les clients ayant le nom "Sarrion" et le prénom "Eric"

```
db = connect("mydb");

var clients = db.clients.find({
    $and : [{ nom : "Sarrion" }, { prenom : "Eric" }]
});
clients.forEach(function(client) {
    printjson(client);
});
```

Les conditions à relier par AND sont exprimées dans un tableau, chaque élément du tableau étant un objet exprimant une condition. Nous verrons par la suite que l'on peut également utiliser le mot-clé `$or` pour exprimer des conditions OR.

Utiliser `$gt`, `$lt` ou `$in` dans une condition

Les conditions précédemment utilisées étaient des strictes égalités. On peut aussi utiliser d'autres types de conditions, par exemple :

- supérieur à (avec `$gt`) ;
- inférieur à (avec `$lt`) ;
- contenu dans un ensemble (avec `$in`).

Le tableau 19-3 liste les clés permettant des comparaisons.

Tableau 19–3 Clés spécifiant des comparaisons

Clé	Signification
\$gt	Supérieur à. Par exemple, { \$gt : 5 }
\$lt	Inférieur à. Par exemple, { \$lt : 5 }
\$in	Inclus dans le tableau de valeurs. Par exemple, { \$in : [1, 2, 3] }
\$gte	Supérieur ou égal à. Par exemple, { \$gte : 5 }
\$lte	Inférieur ou égal à. Par exemple, { \$lte : 5 }
\$ne	Non égal à. Par exemple, { \$ne : 5 }
\$nin	Non inclus dans le tableau de valeurs. Par exemple, { \$nin : [1, 2, 3] }

Le comparateur d'égalité n'existe pas. Par exemple, pour tester que le nom vaut "Sarrion", il suffit d'écrire { nom : "Sarrion" }.

Utilisons ces clés dans les conditions. Pour cela, recherchons tous les documents dont le nom est supérieur à "J" et inférieur à "S". Seuls correspondent "Node Js" et "Rails Ruby". Les clés utilisées sont \$gt et \$lt.

Rechercher tous les documents dont le nom est supérieur à "J" et inférieur à "S"

```
db = connect("mydb");

var clients = db.clients.find( { nom : { $gt : "J", $lt : "S" } } );
clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19–8

Affichage de tous les documents dont le nom est supérieur à "J" et inférieur à "S".

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
  "_id" : ObjectId("529335e97b6b6ad0682be009"),
  "nom" : "Rails",
  "prenom" : "Ruby"
}
{
  "_id" : ObjectId("529335e97b6b6ad0682be00a"),
  "nom" : "Node",
  "prenom" : "Js"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Recherchons maintenant tous les clients dont le nom est "Rails" ou "Sarrion". On utilise pour cela la clé `$in`.

Utiliser la clé `$in` pour rechercher les clients "Sarrion" ou "Rails"

```
db = connect("mydb");

var clients = db.clients.find( { nom : { $in : ["Sarrion", "Rails"] } } );
clients.forEach(function(client) {
  printjson(client);
});
```

Figure 19–9

Affichage de tous les documents dont le nom est "Sarrion" ou "Rails".

The screenshot shows the mongo shell window titled "Invite de commandes". The command run is "mongo testmongo.js". The output shows two documents found in the "clients" collection:

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("529331496cd10c46bedcbbda"),
    "nom" : "Sarrion",
    "prenom" : "Eric"
}
{
    "_id" : ObjectId("529335e97b6b6ad0682be009"),
    "nom" : "Rails",
    "prenom" : "Ruby"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Spécifier une condition OR

Les conditions précédentes étaient des conditions `AND`. On peut également exprimer des conditions `OR`. On utilise pour cela la clé `$or`.

Recherchons par exemple tous les clients dont le nom est "Sarrion" ou le prénom "Ruby".

Utiliser la clé `$or` pour rechercher les clients dont le nom est "Sarrion" ou le prénom "Ruby"

```
db = connect("mydb");

var clients = db.clients.find( { $or : [{ nom : "Sarrion" },
                                         { prenom : "Ruby" }] } );
clients.forEach(function(client) {
  printjson(client);
});
```

La valeur de la clé `$or` est un tableau spécifiant les conditions.

Figure 19-10

Affichage de tous les documents dont le nom est "Sarrion" ou le prénom est "Ruby".

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("529331496cd10c46bedcbda"),
    "nom" : "Sarrion",
    "prenom" : "Eric"
}
{
    "_id" : ObjectId("529335e97b6b6ad0682be009"),
    "nom" : "Rails",
    "prenom" : "Ruby"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Utiliser les conditions AND et OR simultanément

L'objet `query` permet de définir les conditions `AND` et `OR` simultanément. Par défaut, les conditions exprimées sont des conditions `AND`, sauf si la condition `OR` est explicitement indiquée au moyen de la clé `$or`.

Rechercher les clients dont le nom est "Sarrion" et le prénom est "Eric" ou "Ruby"

```
db = connect("mydb");

var clients = db.clients.find({ nom : "Sarrion",
                               $or : [{ prenom : "Eric" }, { prenom : "Ruby" }] });

clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19-11

Affichage de tous les documents dont le nom est "Sarrion" et le prénom est "Eric" ou "Ruby".

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("529331496cd10c46bedcbda"),
    "nom" : "Sarrion",
    "prenom" : "Eric"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Seul le client "Sarrion Eric" répond à la recherche.

Rechercher selon l'existence ou le type d'un champ dans un document

Un autre type de recherche est permis par MongoDB. Elle consiste à rechercher :

- les documents qui possèdent un champ particulier (par exemple, tous les documents qui ont un champ `nom`, quelle que soit sa valeur) ;
- les documents dont un champ particulier est d'un type donné (par exemple, tous les documents dont le champ `nom` est une chaîne de caractères).

Examinons ces deux types de recherches.

Rechercher l'existence d'un champ dans un document avec \$exists

Il est possible, grâce à la clé `$exists`, d'indiquer que l'on recherche les documents qui possèdent un champ donné, mais également l'inverse, c'est-à-dire rechercher tous les documents qui ne possèdent pas ce champ. La valeur du champ dans le document est quelconque, du moment qu'elle est indiquée :

- les valeurs `null`, `" "`, etc., sont des valeurs possibles pour le champ ;
- la valeur `undefined` sera affichée en valeur `null` par le shell MongoDB, bien que la valeur mémorisée en interne ne soit pas `null`.

Dans l'exemple qui suit, nous ajoutons à notre collection `clients` les quatre documents indiqués, pour lesquels la valeur du champ `nom` est `" "`, `null`, `undefined` ou non indiquée. Ces documents permettront de tester la recherche en utilisant la clé `$exists`.

Ajouter quatre clients dont le nom n'est pas spécifié

```
db = connect("mydb");

var client1 = {
    nom : "",
    prenom : "Eric (nom vide)"
};

var client2 = {
    nom : null,
    prenom : "Eric (nom null)"
};

var client3 = {
    nom : undefined,
    prenom : "Eric (nom undefined)"
};
```

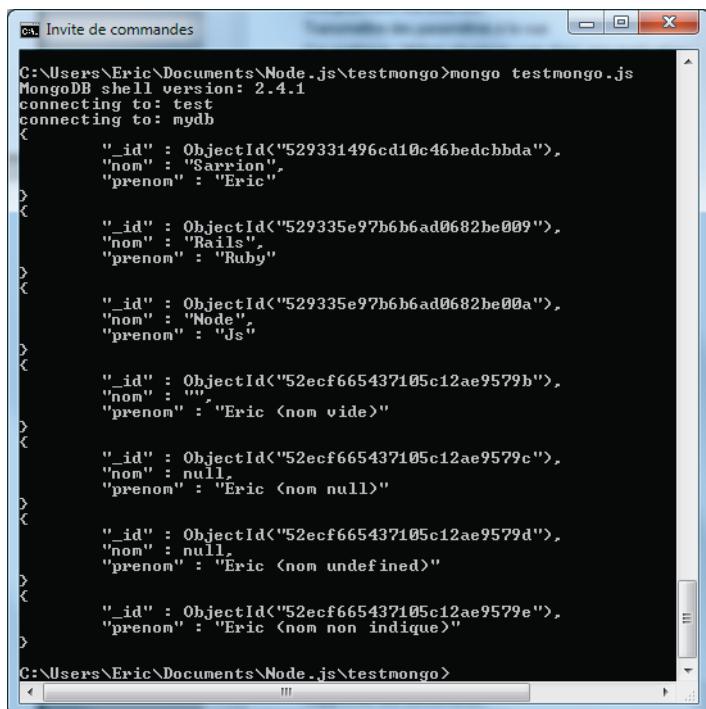
```
var client4 = {
    prenom : "Eric (nom non indiqué)"
};

// insertion des clients dans la collection
db.clients.insert(client1);
db.clients.insert(client2);
db.clients.insert(client3);
db.clients.insert(client4);

// affichage des clients dans la collection
var clients = db.clients.find();
clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19–12

Ajout de nouveaux documents dans la collection clients



The screenshot shows the MongoDB shell version 2.4.1 connected to the 'test' database. It displays the insertion of five documents into the 'clients' collection. The documents are as follows:

- Object ID: 529331496cd10c46bedchbda, nom: Sargon, prenom: Eric
- Object ID: 529335e97b6b6ad0682be009, nom: Rails, prenom: Ruby
- Object ID: 529335e97b6b6ad0682be00a, nom: Node, prenom: Js
- Object ID: 52ecf665437105c12ae9579b, nom: , prenom: Eric <nom vide>
- Object ID: 52ecf665437105c12ae9579c, nom: null, prenom: Eric <nom null>
- Object ID: 52ecf665437105c12ae9579d, nom: null, prenom: Eric <nom undefined>
- Object ID: 52ecf665437105c12ae9579e, nom: null, prenom: Eric <nom non indiqué>

Recherchons maintenant tous les documents pour lesquels le champ `nom` a été indiqué. Tous les documents sauf le dernier doivent correspondre.

Rechercher tous les documents possédant le champ nom

```
db = connect("mydb");

var clients = db.clients.find( { nom : { $exists : true } });
clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19–13

Affichage de tous les documents possédant le champ nom

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
  "_id" : ObjectId("529331496cd10c46bedcbda"),
  "nom" : "SaxriOn",
  "prenom" : "Eric"
}
{
  "_id" : ObjectId("529335e97b6b6ad0682be009"),
  "nom" : "Rails",
  "prenom" : "Ruby"
}
{
  "_id" : ObjectId("529335e97b6b6ad0682be00a"),
  "nom" : "Node",
  "prenom" : "Js"
}
{
  "_id" : ObjectId("52ecf665437105c12ae9579b"),
  "nom" : "",
  "prenom" : "Eric <nom vide>"
}
{
  "_id" : ObjectId("52ecf665437105c12ae9579c"),
  "nom" : null,
  "prenom" : "Eric <nom null>"
}
{
  "_id" : ObjectId("52ecf665437105c12ae9579d"),
  "nom" : null,
  "prenom" : "Eric <nom undefined>"}

C:\Users\Eric\Documents\Node.js\testmongo>
```

À l'inverse, recherchons tous les documents qui ne possèdent pas le champ `nom`. Seul le dernier document doit satisfaire la recherche.

Rechercher tous les documents qui ne possèdent pas le champ nom

```
db = connect("mydb");

var clients = db.clients.find( { nom : { $exists : false } });
clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19–14

Affichage de tous les documents ne possédant pas le champ nom

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("52ecf665437105c12ae9579e"),
    "nom" : null
}
{
    "_id" : ObjectId("52ecf665437105c12ae9579e"),
    "nom" : "Eric <nom non indique>"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Recherchons maintenant tous les documents ayant le champ `nom` positionné à la valeur `null`.

Rechercher tous les documents ayant le champ nom positionné à null

```
db = connect("mydb");

var clients = db.clients.find( { nom : null } );
clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19–15

Affichage de tous les documents possédant le champ nom positionné à null

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("52ecf665437105c12ae9579c"),
    "nom" : null,
    "prenom" : "Eric <nom null>"
}
{
    "_id" : ObjectId("52ecf665437105c12ae9579e"),
    "nom" : "Eric <nom non indique>"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

On voit que les documents trouvés ont le champ `nom` positionné à `null`, mais on trouve également ceux dont le champ `nom` n'a pas été spécifié. De plus, le document dont le champ `nom` est positionné à `undefined` n'est pas trouvé par la recherche (même si l'affichage du champ `nom` est `null`).

Rechercher selon un type de champ avec \$type

Il est possible de rechercher les documents correspondants à un type donné (String, booléen, date, etc.), en utilisant la clé `$type`. Le type peut être de l'un de ceux définis dans le tableau 19-4. La valeur correspondante sera celle indiquée avec la clé `$type`.

Tableau 19-4 Types des champs dans un document

Type	Valeur de la clé \$type
Double	1
String	2
Object	3
Array	4
Binaire	5
Undefined	6 (déprécié, ne plus utiliser)
Object id	7
Boolean	8
Date	9
Null	10
Timestamp	17

Par exemple, si l'on souhaite rechercher tous les documents qui ont le champ `nom` écrit sous forme de chaîne de caractères, on écrira `{ nom : { $type : 2 } }`, sachant que la valeur 2 est associée aux champs de type `String`.

Figure 19-16

Affichage des documents ayant le champ nom sous forme de chaîne de caractères

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("529331496cd10c46bedcbhda"),
    "nom" : "Saxion",
    "prenom" : "Eric"
}

{
    "_id" : ObjectId("529335e97b6b6ad0682be009"),
    "nom" : "Rails",
    "prenom" : "Ruby"
}

{
    "_id" : ObjectId("529335e97b6b6ad0682be00a"),
    "nom" : "Node",
    "prenom" : "Js"
}

{
    "_id" : ObjectId("52ecf665437105c12ae9579h"),
    "nom" : "",
    "prenom" : "Eric <nom vide>"
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

Rechercher tous les documents dont le nom est sous forme de chaîne de caractères

```
db = connect("mydb");

var clients = db.clients.find( { nom : { $type : 2 } } );
clients.forEach(function(client) {
    printjson(client);
});
```

Les documents ayant le nom positionné à `null`, `undefined`, voire non indiqué, ne sont pas récupérés par le critère de recherche utilisé ici. Pour récupérer les documents ayant également le champ `nom` positionné à `null`, on pourrait écrire :

Récupérer les documents ayant le champ nom à null ou sous forme de chaîne de caractères

```
db = connect("mydb");

var clients = db.clients.find({
    $or : [ { nom : { $type : 2 } }, { nom : { $type : 10 } } ]
});
clients.forEach(function(client) {
    printjson(client);
});
```

La clé `$or` permet de spécifier un tableau d'objets définissant chacun une condition. Les valeurs 2 et 10 pour la clé `$type` désignent respectivement les types `String` et les valeurs `null`.

Figure 19-17

Affichage des documents ayant le champ nom sous forme de chaîne de caractères ou valant null

```
Invité de commandes
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("529331496cd10c46bedchbda"),
    "nom" : "Sarrión",
    "prenom" : "Eric"
}
{
    "_id" : ObjectId("529335e97b6b6ad0682be009"),
    "nom" : "Rails",
    "prenom" : "Ruby"
}
{
    "_id" : ObjectId("529335e97b6b6ad0682be00a"),
    "nom" : "Node",
    "prenom" : "Js"
}
{
    "_id" : ObjectId("52ecf665437105c12ae9579b"),
    "nom" : null,
    "prenom" : "Eric <nom vide>"
}
{
    "_id" : ObjectId("52ecf665437105c12ae9579c"),
    "nom" : null,
    "prenom" : "Eric <nom null>"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Le document ayant le champ `nom` positionné à `null` est maintenant trouvé, en plus des autres ayant le type `String`.

Rechercher à l'aide d'une expression JavaScript avec `$where`

Lorsque les conditions de recherche ne peuvent s'exprimer uniquement à l'aide des mots-clés explicités précédemment, on peut utiliser le mot-clé `$where` dont la valeur est une expression JavaScript.

Recherchons, par exemple, tous les clients dont le nom est "Sarrion" ou "Node", sans tenir compte des majuscules. Une expression avec `$or` serait possible mais un peu compliquée à écrire pour tenir compte de tous les cas possibles.

Utiliser `$where` pour exprimer des conditions en JavaScript

```
db = connect("mydb");

var clients = db.clients.find({
  $where : "this.nom.match(/^sarrion|node$/i)"
});

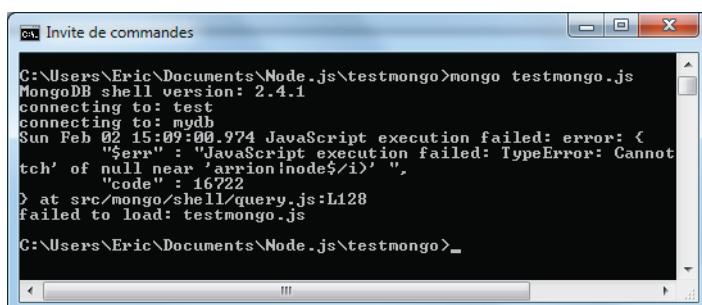
clients.forEach(function(client) {
  printjson(client);
});
```

L'accès à un document est effectué au moyen du mot-clé `this`. L'instruction `this.nom` permet donc d'accéder au nom du client, sur lequel on utilise la méthode `match()` contenant une expression régulière associée au test. Si le retour de l'instruction JavaScript est `true`, le document correspondant fait partie du résultat de la recherche, sinon il en est écarté.

Regardons si cela fonctionne. Deux documents devraient correspondre au résultat attendu.

Figure 19-18

Erreur lors de la recherche des documents dont le nom est "sarrion" ou "node".



Une erreur se produit. En effet, certains documents de la collection n'ont pas le champ `nom`, et ceux qui l'ont, n'ont pas forcément celui-ci valant une chaîne de caractères. Ainsi, l'utilisation de la méthode `match()` sur autre chose qu'une chaîne de caractères produit une erreur.

En effet, l'utilisation de la clause `$where` implique que la totalité des documents de la collection soit analysée par MongoDB, y compris ceux qui ne seront pas récupérés en tant que résultat. On doit donc être prudent dans l'écriture du code JavaScript associé à la clause `$where`.

Modifions le programme pour éviter cette erreur. Il suffit de vérifier que le document correspondant possède un champ `nom` ayant le type `String`. On effectue ceci au moyen de la clé `$type`.

Tester si le champ nom est de type String

```
db = connect("mydb");

var clients = db.clients.find({
  nom : { $type : 2 },
  $where : "this.nom.match(/^sarrion|node$/i)"
});

clients.forEach(function(client) {
  printjson(client);
});
```

L'ajout de conditions en supplément de la clause `$where` permet de faire un tri préalable des documents qui seront transmis à la clause `$where` pour analyse. Ces conditions peuvent être écrites avant ou après la clause `$where`, celle-ci étant toujours exécutée en dernier.

Figure 19–19

Plus aucune erreur lors de la recherche des documents dont le nom est "sarrion" ou "node".

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
  "_id" : ObjectId("529331496cd10c46bedchbda"),
  "nom" : "Sarrion",
  "prenom" : "Eric"
}
{
  "_id" : ObjectId("529335e97b6b6ad0682be00a"),
  "nom" : "Node",
  "prenom" : "Js"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Les deux documents attendus sont trouvés dans la recherche.

Rechercher dans des sous-documents

Un document peut contenir d'autres documents, appelés « sous-documents ». Il suffit pour cela d'indiquer, sous forme JSON, une valeur qui est elle-même un objet JSON.

Insérons trois nouveaux documents dans la collection `clients`, comportant la mention de l'adresse du client sous la forme d'un objet `adresse` décrit sous la forme `{ rue, cp, ville }`, chacun de ces trois champs ayant pour valeur une chaîne de caractères.

Insertion de trois nouveaux documents dans la collection clients, comportant l'adresse du client

```
db = connect("mydb");

var client1 = {
    nom : "Dupont",
    prenom : "Valentin",
    adresse : {
        rue : "17 rue Villeneuve",
        cp : "69000",
        ville : "Lyon"
    }
};

db.clients.insert(client1);

var client2 = {
    nom : "Durand",
    prenom : "Georges",
    adresse : {
        rue : "7 av du centre",
        cp : "69000",
        ville : "Lyon"
    }
};

db.clients.insert(client2);

var client3 = {
    nom : "Duchemin",
    prenom : "Valentin",
    adresse : {
        rue : "7 place du Tertre",
        cp : "75018",
        ville : "Paris"
    }
};
```

```
db.clients.insert(client3);

var clients = db.clients.find();

clients.forEach(function(client) {
  printjson(client);
});
```

Figure 19–20

Insertion de nouveaux documents (et sous-documents) dans la collection clients

```
db> db.clients.insert(client3);

db> var clients = db.clients.find();

db> clients.forEach(function(client) {
  printjson(client);
});

{
  "_id" : ObjectId("52ecf665437105c12ae9579c"),
  "nom" : null,
  "prenom" : "Eric <nom null>"

{
  "_id" : ObjectId("52ecf665437105c12ae9579d"),
  "nom" : null,
  "prenom" : "Eric <nom undefined>"

{
  "_id" : ObjectId("52ecf665437105c12ae9579e"),
  "prenom" : "Eric <nom non indiqué>"

{
  "_id" : ObjectId("52ee6159da4f2753442403de"),
  "nom" : "Dupont",
  "prenom" : "Valentin",
  "adresse" : {
    "rue" : "17 rue Villeneuve",
    "cp" : "69000",
    "ville" : "Lyon"
  }

{
  "_id" : ObjectId("52ee6159da4f2753442403df"),
  "nom" : "Durand",
  "prenom" : "Georges",
  "adresse" : {
    "rue" : "7 av du centre",
    "cp" : "69000",
    "ville" : "Lyon"
  }

{
  "_id" : ObjectId("52ee6159da4f2753442403e0"),
  "nom" : "Duchemin",
  "prenom" : "Valentin",
  "adresse" : {
    "rue" : "7 place du Tertre",
    "cp" : "75018",
    "ville" : "Paris"
  }
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Les trois derniers clients insérés sont visibles en bas de la fenêtre, ils utilisent le champ `adresse` défini comme dans le code précédent.

On souhaite rechercher les documents dont la ville est "Lyon". Le champ `ville` fait partie du champ `adresse`, il sera donc accédé par `adresse.ville`. Toutefois, ce champ étant un nom composé, il devra être mentionné dans les attributs par "`adresse.ville`" et non pas seulement `adresse.ville`, ce qui provoquerait une erreur de syntaxe JavaScript.

Rechercher les clients qui sont de "Lyon"

```
db = connect("mydb");

var clients = db.clients.find({ "adresse.ville" : "Lyon" });

clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19–21

Affichage des clients dont la ville est "Lyon".

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
  "_id" : ObjectId("52ee6159da4f2753442403de"),
  "nom" : "Dupont",
  "prenom" : "Valentin",
  "adresse" : {
    "rue" : "17 rue Villeneuve",
    "cp" : "69000",
    "ville" : "Lyon"
  }
}
{
  "_id" : ObjectId("52ee6159da4f2753442403df"),
  "nom" : "Durand",
  "prenom" : "Georges",
  "adresse" : {
    "rue" : "7 av du centre",
    "cp" : "69000",
    "ville" : "Lyon"
  }
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Il suffit donc de mentionner l'attribut de façon complète, en séparant chaque nom du suivant par un point, et en l'entourant des guillemets nécessaires. Une fois ceci mis en place, on peut utiliser ce nouvel attribut de la même manière que les autres.

Par exemple, si l'on souhaite rechercher les clients sur "Lyon" dont le nom est "Dupont", on écrira alors :

Rechercher les clients sur "Lyon" dont le nom est "Dupont"

```
db = connect("mydb");

var clients = db.clients.find({ nom : "Dupont", "adresse.ville" : "Lyon" });

clients.forEach(function(client) {
    printjson(client);
});
```

L'attribut `nom` valant "Dupont" est spécifié de manière traditionnelle, suivi de l'attribut "adresse.ville".

Figure 19–22

Affichage des clients
dont le nom est "Dupont"
et dont la ville est "Lyon".

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("52ee6159da4f2753442403de"),
    "nom" : "Dupont",
    "prenom" : "Valentin",
    "adresse" : [
        {
            "rue" : "17 rue Villeneuve",
            "cp" : "69000",
            "ville" : "Lyon"
        },
        {
            "rue" : "3 bd Daumesnil",
            "cp" : "75012",
            "ville" : "Paris"
        }
    ]
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

Rechercher dans des tableaux

Un tableau peut être associé à la valeur d'un champ. Considérons qu'un client peut avoir plusieurs adresses, et non une seule comme précédemment. On indiquera donc dans le champ `adresse` un tableau d'objets et non plus seulement un objet.

Insérer le client Martin qui possède deux adresses

```
db = connect("mydb");

var client1 = {
    nom : "Martin",
    prenom : "Michel",
    adresse : [
        {
            rue : "17 rue Villeneuve",
            cp : "69000",
            ville : "Lyon"
        },
        {
            rue : "3 bd Daumesnil",
            cp : "75012",
            ville : "Paris"
        }
    ];
};
```

```
db.clients.insert(client1);

var clients = db.clients.find();

clients.forEach(function(client) {
  printjson(client);
});
```

Figure 19–23

Insertion d'un client ayant deux adresses

```
"prenom" : "Eric <nom non indiqué>"  
>  
<  
  "_id" : ObjectId("52ee6159da4f2753442403de"),  
  "nom" : "Dupont",  
  "prenom" : "Valentin",  
  "adresse" : [  
    {"rue" : "17 rue Villeneuve",  
     "cp" : "69000",  
     "ville" : "Lyon"}  
>  
<  
  "_id" : ObjectId("52ee6159da4f2753442403df"),  
  "nom" : "Duxand",  
  "prenom" : "Georges",  
  "adresse" : [  
    {"rue" : "7 av du centre",  
     "cp" : "69000",  
     "ville" : "Lyon"}  
>  
<  
  "_id" : ObjectId("52ee6159da4f2753442403e0"),  
  "nom" : "Duchemin",  
  "prenom" : "Valentin",  
  "adresse" : [  
    {"rue" : "7 place du Tertre",  
     "cp" : "75018",  
     "ville" : "Paris"}  
>  
<  
  "_id" : ObjectId("52ee7158b1ece7e0cf70fd0h"),  
  "nom" : "Martin",  
  "prenom" : "Michel",  
  "adresse" : [  
    {"rue" : "17 rue Villeneuve",  
     "cp" : "69000",  
     "ville" : "Lyon"},  
    {"rue" : "3 bd Daumesnil",  
     "cp" : "75012",  
     "ville" : "Paris"}  
>  
>  
C:\Users\Eric\Documents\Node.js\testmongo>
```

Nous voyons ce client en dernier dans la liste des clients déjà inscrits dans la collection.

Recherchons tous les clients situés sur "Lyon". Trois clients ont une adresse à "Lyon", le troisième ayant également une adresse à "Paris".

Rechercher tous les clients situés à "Lyon"

```
db = connect("mydb");

var clients = db.clients.find({ "adresse.ville" : "Lyon" });

clients.forEach(function(client) {
    printjson(client);
});
```

La même syntaxe que précédemment s'applique, que la valeur soit un tableau ou pas.

Figure 19–24

Recherche des documents dont la ville est "Lyon".

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("52ee6159da4f2753442403de"),
    "nom" : "Dupont",
    "prenom" : "Valentin",
    "adresse" : [
        {
            "rue" : "17 rue Villeneuve",
            "cp" : "69000",
            "ville" : "Lyon"
        }
    ]
}
{
    "_id" : ObjectId("52ee6159da4f2753442403df"),
    "nom" : "Durand",
    "prenom" : "Georges",
    "adresse" : [
        {
            "rue" : "7 av du centre",
            "cp" : "69000",
            "ville" : "Lyon"
        }
    ]
}
{
    "_id" : ObjectId("52ee7158b1e7e0cf70fd0b"),
    "nom" : "Martin",
    "prenom" : "Michel",
    "adresse" : [
        {
            "rue" : "17 rue Villeneuve",
            "cp" : "69000",
            "ville" : "Lyon"
        },
        {
            "rue" : "3 bd Daumesnil",
            "cp" : "75012",
            "ville" : "Paris"
        }
    ]
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Le client ayant une adresse à "Lyon" et "Paris" est trouvé. La recherche ne tient pas compte ici de l'ordre dans lequel les éléments ont été placés dans le tableau. Pour en

tenir compte, il suffit d'indiquer l'index de l'élément dans le nom du champ lors de la recherche. Ainsi, pour indiquer que l'on souhaite trouver les clients ayant un tableau d'adresses dont la première indiquée est "Lyon", on écrira :

Rechercher tous les clients ayant un tableau d'adresses et dont la première adresse est "Lyon"

```
db = connect("mydb");

var clients = db.clients.find({ "adresse.0.ville" : "Lyon" });

clients.forEach(function(client) {
    printjson(client);
});
```

Figure 19–25

Affichage des documents dont la ville est "Lyon" dans un tableau en première position.

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
  "_id" : ObjectId("52ee7158b1ece7e0cf70fd0b"),
  "nom" : "Martin",
  "prenom" : "Michel",
  "adresse" : [
    {
      "rue" : "17 rue Villeneuve",
      "cp" : "69000",
      "ville" : "Lyon"
    },
    {
      "rue" : "3 bd Daumesnil",
      "cp" : "75012",
      "ville" : "Paris"
    }
  ]
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

Seul le dernier client inséré, ayant plusieurs adresses dont la première indiquée est "Lyon", est trouvé lors de cette recherche. Pour trouver les autres clients situés sur "Lyon", il aurait fallu uniformiser l'insertion des documents en indiquant un tableau d'adresses pour chacun d'eux, même si ce tableau ne contenait qu'un seul élément.

Trier les documents lors d'une recherche

Les documents retournés lors d'une recherche sont triés par défaut selon leur ordre de création. Il peut être intéressant de les trier selon d'autres critères, par exemple alphabétiquement selon le nom.

Trier les documents par ordre croissant des noms

```
db = connect("mydb");

var clients = db.clients.find().sort({ nom : 1 });

clients.forEach(function(client) {
    printjson(client);
});
```

La valeur 1 indiquée pour le nom signifie de trier selon l'ordre croissant. Une valeur de -1 trierait selon un ordre décroissant.

La figure 19-26 montre l'affichage obtenu après un tri croissant sur les noms.

Figure 19-26

Documents triés selon les noms croissants



The screenshot shows the MongoDB shell interface with the title bar "Invite de commandes". The command line at the top reads "C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js". The main pane displays the results of a database query:

```

C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
  "_id" : ObjectId("52ecf665437105c12ae9579d"),
  "nom" : null,
  "prenom" : "Eric <nom undefined>"

  "_id" : ObjectId("52ecf665437105c12ae9579e"),
  "prenom" : "Eric <nom non indique>"

  "_id" : ObjectId("52ecf665437105c12ae9579c"),
  "nom" : null,
  "prenom" : "Eric <nom null>"

  "_id" : ObjectId("52ecf665437105c12ae9579b"),
  "nom" : "",
  "prenom" : "Eric <nom vide>"

  "_id" : ObjectId("52ee6159da4f2753442403e0"),
  "nom" : "Duchemin",
  "prenom" : "Valentin",
  "adresse" : {
    "rue" : "7 place du Tertre",
    "cp" : "75018",
    "ville" : "Paris"
  }

  "_id" : ObjectId("52ee6159da4f2753442403de"),
  "nom" : "Dupont",
  "prenom" : "Valentin",
  "adresse" : {
    "rue" : "17 rue Villeneuve",
    "cp" : "69000",
    "ville" : "Lyon"
  }

  "_id" : ObjectId("52ee6159da4f2753442403df"),
  "nom" : "Durand",
  "prenom" : "Georges",
  "adresse" : {
    "rue" : "7 av du centre",
    "cp" : "69000",
    "ville" : "Lyon"
  }

  "_id" : ObjectId("52ee7158b1ece7e0cf70fd0b"),
  "nom" : "Martin"
}

```

Pour trier selon un ordre décroissant des noms, il suffit d'indiquer la valeur -1 au lieu de 1.

Trier les documents par ordre décroissant des noms

```
db = connect("mydb");

var clients = db.clients.find().sort({ nom : -1 });

clients.forEach(function(client) {
  printjson(client);
});
```

Figure 19-27
Documents triés selon
les noms décroissants

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
  "_id" : ObjectId("529331496cd10c46bedcbhda"),
  "nom" : "Sarrion",
  "prenom" : "Eric"
}
{
  "_id" : ObjectId("529335e97b6b6ad0682be009"),
  "nom" : "Rails",
  "prenom" : "Ruby"
}
{
  "_id" : ObjectId("529335e97b6b6ad0682be00a"),
  "nom" : "Node",
  "prenom" : "Js"
}
{
  "_id" : ObjectId("52ee7158b1ece7e0cf70fd0h"),
  "nom" : "Martin",
  "prenom" : "Michel",
  "adresse" : [
    {
      "rue" : "17 rue Villeneuve",
      "cp" : "69000",
      "ville" : "Lyon"
    },
    {
      "rue" : "3 bd Daumesnil",
      "cp" : "75012",
      "ville" : "Paris"
    }
  ]
}
{
  "_id" : ObjectId("52ee6159da4f2753442403df"),
  "nom" : "Dupond",
  "prenom" : "Georges",
  "adresse" : [
    {
      "rue" : "7 av du centre",
      "cp" : "69000",
      "ville" : "Lyon"
    }
  ]
}
{
  "_id" : ObjectId("52ee6159da4f2753442403de"),
  "nom" : "Dupont",
  "prenom" : "Valentin",
  "adresse" : [
    {
      "rue" : "17 rue Villeneuve",
      "cp" : "69000",
      "ville" : "Lyon"
    }
  ]
}
```

Pour trier selon plusieurs critères, par exemple le nom croissant, puis le prénom décroissant, il suffit d'indiquer les attributs les uns à la suite des autres.

Trier selon les noms croissants, puis les prénoms décroissants (dans le cas où deux noms sont identiques)

```
db = connect("mydb");

var clients = db.clients.find().sort({ nom : 1, prenom : -1 });

clients.forEach(function(client) {
  printjson(client);
});
```

Indiquer les champs à retourner lors d'une recherche

Par défaut, tous les champs inscrits dans un document sont retournés lors d'une recherche. On peut indiquer à la méthode `find(query, projection)` les champs à récupérer au moyen du paramètre `projection`.

On indique dans l'objet `projection` les noms des champs que l'on souhaite inclure ou exclure, en leur associant la valeur 1 pour les inclure ou 0 pour les exclure. Les champs indiqués doivent être tous inclus ou tous exclus (c'est-à-dire uniquement des valeurs 0 ou uniquement des valeurs 1), à l'exception du champ `_id` qui peut être inclus ou exclu sans tenir compte des autres champs (par défaut, il est inclus s'il n'est pas mentionné).

La méthode `find(query, projection)` recherche tout d'abord tous les documents qui correspondent à l'objet `query` spécifié, puis elle conserve ou exclut dans ces documents les champs associés à l'objet `projection`. Le nombre de documents résultants n'est pas modifié par l'objet `projection`, seuls les champs retournés le sont.

Par exemple, pour afficher tous les documents en ne conservant que les champs `nom` et `_id`, on écrira :

Afficher tous les documents avec le nom et le champ `_id`

```
db = connect("mydb");

var clients = db.clients.find(null, { nom : 1 });

clients.forEach(function(client) {
  printjson(client);
});
```

Le paramètre `query`, valant `null`, permet de rechercher tous les documents sans spécifier de conditions. On peut aussi indiquer `undefined` ou un objet vide `{ }`.

Figure 19–28

Documents dont seuls les noms et l'identifiant sont retournés.

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mldb
{
  "_id": ObjectId("529331496cd10c46bedchbda"),
  "nom": "Sarrion"
}
{
  "_id": ObjectId("529335e97b6b6ad0682be009"),
  "nom": "Rails"
}
{
  "_id": ObjectId("529335e97b6b6ad0682be00a"),
  "nom": "Node"
}
{
  "_id": ObjectId("52ecf665437105c12ae9579b"),
  "nom": ""
}
{
  "_id": ObjectId("52ecf665437105c12ae9579c"),
  "nom": null
}
{
  "_id": ObjectId("52ecf665437105c12ae9579d"),
  "nom": null
}
{
  "_id": ObjectId("52ee6159da4f2753442403de"),
  "nom": "Dupont"
}
{
  "_id": ObjectId("52ee6159da4f2753442403df"),
  "nom": "Durand"
}
{
  "_id": ObjectId("52ee6159da4f2753442403e0"),
  "nom": "Duchemin"
}
{
  "_id": ObjectId("52ee7158blece7e0cf70f0b"),
  "nom": "Martin"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Tous les documents sont affichés, y compris ceux qui ne possèdent pas le champ `nom`. Mais parmi les documents affichés, seul le champ `nom` est retourné, même pour ceux qui possèdent d'autres champs.

Le champ `_id` est retourné dans tous les cas. Pour ne pas récupérer ce champ, il suffit de l'exclure de la recherche.

Afficher tous les documents avec le nom uniquement

```
db = connect("mydb");

var clients = db.clients.find({ }, { _id : 0, nom : 1 });

clients.forEach(function(client) {
  printjson(client);
});
```

Figure 19–29

Documents dont seuls les noms sont retournés.

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mldb
{
  "nom": "Sarrion"
}
{
  "nom": "Rails"
}
{
  "nom": "Node"
}
{
  "nom": ""
}
{
  "nom": null
}
{
  "nom": null
}
{
  "nom": "Dupont"
}
{
  "nom": "Durand"
}
{
  "nom": "Duchemin"
}
{
  "nom": "Martin"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Compter le nombre de documents trouvés lors d'une recherche

Une recherche au moyen de la méthode `find()` permet de récupérer un tableau de documents. Toutefois, ce tableau n'est pas un objet de classe `Array`, mais plutôt un objet appelé curseur (`cursor` en anglais). Sur cet objet curseur, MongoDB a défini des méthodes parmi lesquelles la méthode `forEach()` permettant de parcourir chacun des éléments du curseur.

Ainsi, les méthodes de la classe `Array` ne fonctionneront généralement pas sur les objets curseur. Pour récupérer le nombre total d'éléments dans le curseur, la propriété `length` associée aux tableaux est inopérante et on utilisera alors la méthode `count()`, définie par MongoDB et qui permet de connaître le nombre d'éléments rentrés par une recherche.

Tableau 19–5 Compter le nombre de documents retournés lors d'une recherche

Méthode	Signification
<code>cursor.count()</code>	Retourne le nombre de documents associés au curseur indiqué. Celui-ci a été préalablement créé avec la commande <code>db.nomCollection.find()</code> .
<code>cursor.length()</code>	Synonyme de <code>cursor.count()</code> . Ne pas confondre avec la propriété <code>length</code> sur les tableaux (ici, c'est la méthode <code>length()</code> sur les curseurs).

Utiliser la méthode `cursor.count()` pour compter le nombre de documents retournés par la recherche

```
db = connect("mydb");

var clients = db.clients.find();
print ("Nombre de documents dans la collection clients : " + clients.count());
```

L'objet `clients` retourné par la méthode `find()` est un curseur sur lequel s'applique la méthode `count()`.

Figure 19–30
Affichage du nombre de documents dans la collection clients

```
C:\ Invité de commandes
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
Nombre de documents dans la collection clients : 11
C:\Users\Eric\Documents\Node.js\testmongo>
```

Rechercher le premier document qui satisfait une recherche

Plutôt que de rechercher tous les documents qui satisfont une recherche, on peut rechercher uniquement le premier (dans l'ordre où ils sont écrits dans la collection). On utilise pour cela la méthode `findOne()` sur la collection, qui retourne un seul document au lieu d'un tableau comme précédemment. Si aucun document n'est trouvé, la méthode retourne `null`.

Tableau 19–6 Méthode de recherche du premier document

Méthode	Signification
<code>db.nomCollection.findOne(query)</code>	Recherche le premier document correspondant aux critères indiqués dans l'objet <code>query</code> . Si aucun document n'est trouvé, retourne <code>null</code> .

Rechercher le premier document dont le champ nom est "Sarrion"

```
db = connect("mydb");

var client = db.clients.findOne( { nom : "Sarrion" } );
printjson(client);
```

Figure 19–31

Afficher le premier client dont le nom est "Sarrion".

```
C:\Users\Eric\Documents\Node.js\testmongo>mongo testmongo.js
MongoDB shell version: 2.4.1
connecting to: test
connecting to: mydb
{
    "_id" : ObjectId("529331496cd10c46bedcbbda"),
    "nom" : "Sarrion",
    "prenom" : "Eric"
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

Mettre à jour des documents

Nous avons vu comment insérer des documents et comment les récupérer lors d'une recherche. Voyons maintenant comment les mettre à jour. Deux méthodes existent pour cela, elles sont présentées dans le tableau 19–7.

Tableau 19–7 Méthodes de mise à jour des documents

Méthode	Signification
<code>db.nomCollection.save(document)</code>	Sauvegarde le document dans la collection, en effectuant une insertion ou une mise à jour. Si le document à sauvegarder contient le champ <code>_id</code> renseigné à une valeur existante, le document est mis à jour (il remplace le précédent ayant le même <code>_id</code>), sinon le document est inséré dans la collection en tant que nouveau document (MongoDB affecte éventuellement une valeur au champ <code>_id</code> s'il n'en possède pas encore).
<code>db.nomCollection.update(query, update, options)</code>	Met à jour un document ou un ensemble de documents. Le paramètre <code>query</code> spécifie le ou les documents à mettre à jour. Le paramètre <code>update</code> indique la mise à jour à effectuer. Le paramètre <code>options</code> possède les deux clés suivantes : - <code>multi : false</code> , valeur par défaut) ou plusieurs (<code>{ multi : true }</code>). - <code>upsert</code> indique si un nouveau document est à créer si aucun document ne correspond au paramètre <code>query</code> (<code>{ upsert : true }</code>) ou non (<code>{ upsert : false }</code> , valeur par défaut).

Examinons chacune de ces méthodes, en commençant par la méthode `save()`.

Utiliser la méthode `save(document)`

Utilisons la méthode `save(document)` pour mettre à jour un document préalablement inséré dans la collection `clients`. Un de nos documents ne possède pas de champ `nom` : mettons à jour celui-ci avec le nom "Hollande".

Ajouter "Hollande" au document n'ayant pas de nom indiqué

```
db = connect("mydb");

var clients = db.clients.find( { nom : { $exists : false } });
if (clients.count()) {
  clients[0].nom = "Hollande";
  db.clients.save(clients[0]);
}

var clients = db.clients.find();
clients.forEach(function(client) {
  printjson(client);
});
```

Nous recherchons d'abord les clients dont le nom n'a pas été indiqué (si le nom est une chaîne vide ou `null`, il est considéré comme étant indiqué). Pour le premier client trouvé (ici, un seul), on ajoute la propriété `nom` en la positionnant à "`Hollande`" comme demandé, puis on sauvegarde ce document dans la collection. Le champ `_id` n'ayant pas été modifié, ce document remplace l'ancien.

Figure 19-32
Modification du nom
d'un client

```

[{"_id": ObjectId("52ee7158biece7e0cf70fd0h"), "nom": "Martin", "prenom": "Michel", "adresse": [{"rue": "17 rue Villeneuve", "cp": "69000", "ville": "Lyon"}, {"rue": "3 bd Daumesnil", "cp": "75012", "ville": "Paris"}]}, {"_id": ObjectId("52ecf665437105c12ae9579e"), "prenom": "Eric <nom non indiqué>", "nom": "Hollande"}, {"_id": ObjectId("529331496cd10c46bedcbdbda"), "nom": "Sarrion", "prenom": "Eric"}, {"_id": ObjectId("529335e97b6b6ad0682be009"), "nom": "Rails", "prenom": "Ruby"}]
C:\Users\Eric\Documents\Node.js\testmongo>

```

Le champ `nom` a été créé pour le client demandé, en conservant la même valeur du champ `_id`.

La méthode `save()` peut également servir à créer de nouveaux documents. Il suffit pour cela que le champ `_id` ne soit pas affecté (ou affecté à une valeur non encore insérée dans la collection). Par exemple, créons le client de nom "`Pays`" et de prénom "`Bas`".

Créer un client en utilisant la méthode `save()`

```

db = connect("mydb");

var c = { nom : "Pays" , prenom : "Bas" };
db.clients.save(c);

var clients = db.clients.find();

clients.forEach(function(client) {
  printjson(client);
});

```

Figure 19–33

Création du client "Pays Bas"

The screenshot shows the MongoDB shell interface with the title bar 'Invite de commandes'. The command line displays the insertion of a new document into the 'clients' collection:

```

    "_id" : ObjectId("52ee7158b1ece7e0cf70fd0b"),
    "nom" : "Martin",
    "prenom" : "Michel",
    "adresse" : [
        {
            "rue" : "17 rue Villeneuve",
            "cp" : "69000",
            "ville" : "Lyon"
        },
        {
            "rue" : "3 bd Daumesnil",
            "cp" : "75012",
            "ville" : "Paris"
        }
    ],
    "_id" : ObjectId("52ecf665437105c12ae9579e"),
    "prenom" : "Eric <nom non indiqué>",
    "nom" : "Hollande"
},
{
    "_id" : ObjectId("529331496cd10c46bedchbda"),
    "nom" : "Sarrion",
    "prenom" : "Eric"
},
{
    "_id" : ObjectId("529335e97b6b6ad0682be009"),
    "nom" : "Rails",
    "prenom" : "Ruby"
},
{
    "_id" : ObjectId("52f65f98a4e1669bbbb0695e"),
    "nom" : "Pays",
    "prenom" : "Bas"
}

```

The command concludes with the path 'C:\Users\Eric\Documents\Node.js\testmongo>'.

Une fois inséré, MongoDB affecte un nouvel `_id` à ce document.

Enfin, montrons comment supprimer une propriété dans un document. On souhaite supprimer le nom "[Hollande](#)" pour le client auquel on l'a précédemment ajouté.

Supprimer le nom pour le client "Hollande"

```

db = connect("mydb");

var clients = db.clients.find( { nom : "Hollande" } );
if (clients.count()) {
  delete clients[0].nom;
  db.clients.save(clients[0]);
}

var clients = db.clients.find();

clients.forEach(function(client) {
  printjson(client);
});

```

On utilise l'opérateur JavaScript `delete` qui permet de supprimer une propriété dans un objet. Le document doit ensuite être sauvegardé par `save()` dans la collection `clients`.

Figure 19–34
Suppression d'un champ
dans un document

```

C:\ Invite de commandes
{
  "_id" : ObjectId("52ee7158b1ece7e0cf70fd0b"),
  "nom" : "Martin",
  "prenom" : "Michel",
  "adresse" : [
    {
      "rue" : "17 rue Villeneuve",
      "cp" : "69000",
      "ville" : "Lyon"
    },
    {
      "rue" : "3 bd Daumesnil",
      "cp" : "75012",
      "ville" : "Paris"
    }
  ]
}
{
  "_id" : ObjectId("52ecf665437105c12ae9579e"),
  "prenom" : "Eric (nom non indiqué)"
}
{
  "_id" : ObjectId("529331496cd10c46bedcbda"),
  "nom" : "Sarrión",
  "prenom" : "Eric"
}
{
  "_id" : ObjectId("529335e97b6b6ad0682be009"),
  "nom" : "Rails",
  "prenom" : "Ruby"
}
{
  "_id" : ObjectId("52f65f98a4e1669bbbb0695e"),
  "nom" : "Pays",
  "prenom" : "Bas"
}

C:\Users\Eric\Documents\Node.js\testmongo> ...

```

Utiliser la méthode update(query, update, options)

La méthode `update(query, update, options)` sera utile lorsqu'on ne dispose pas en mémoire des documents à mettre à jour dans la base de données (sinon utiliser plutôt la méthode `save(document)` précédente).

- Le paramètre `query` permet de sélectionner les documents à mettre à jour dans la base de données, et correspond à celui utilisé dans la méthode `find(query, projection)` étudiée précédemment. On indique sous forme JSON les conditions que doivent respecter le ou les éléments à mettre à jour. Par défaut, un seul document sera mis à jour (même si plusieurs documents satisfont les conditions de recherche), sauf si l'option `multi` est positionnée (paramètre `options`).
- Le paramètre `update` permet d'indiquer la mise à jour à effectuer. On peut remplacer tout ou partie des documents trouvés, selon la façon dont le paramètre `update` est écrit.
- Le paramètre `options` comprend deux clés possibles : `multi` et `upsert`. La clé `multi` (si positionnée à `true`, par défaut `false`) permet d'indiquer de mettre à jour tous les documents retournés par la recherche, au lieu du premier seulement. La clé `upsert` (si positionnée à `true`, par défaut `false`) permet de créer un nouveau document dans le cas où la recherche n'a trouvé aucun document à mettre à jour.

Dans les sections suivantes, nous examinons diverses utilisations possibles de ces paramètres dans la méthode `update()`.

Mettre à jour la totalité d'un document

Cette première forme d'utilisation de la méthode `update(query, update, options)` permet de remplacer un document existant par un nouveau document possédant le même `_id`. Il suffit d'indiquer dans le paramètre `update` le nouveau document. Tous les champs de l'ancien document, excepté `_id`, seront remplacés par les champs du nouveau document.

Remplacer le document n'ayant pas de nom par { nom : "Jobs", prenom : "Steve" }

```
db = connect("mydb");

db.clients.update(
  nom : { $exists : false }
), {
  nom : "Jobs", prenom : "Steve"
});

var clients = db.clients.find();

clients.forEach(function(client) {
  printjson(client);
});
```

Figure 19–35

Mise à jour des documents n'ayant pas de nom

```
cat Invité de commandes
{
  "_id": ObjectId("52ee7158b1ece7e0cf70fd0b"),
  "nom": "Martin",
  "prenom": "Michel",
  "adresse": [
    {
      "rue": "17 rue Villeneuve",
      "cp": "69000",
      "ville": "Lyon"
    },
    {
      "rue": "3 bd Daumesnil",
      "cp": "75012",
      "ville": "Paris"
    }
  ]
},
{
  "_id": ObjectId("52ecf665437105c12ae9579e"),
  "nom": "Jobs",
  "prenom": "Steve"
},
{
  "_id": ObjectId("529331496cd10c46bedchbda"),
  "nom": "Saxxon",
  "prenom": "Eric"
},
{
  "_id": ObjectId("529335e97b6b6ad0682be009"),
  "nom": "Rails",
  "prenom": "Ruby"
},
{
  "_id": ObjectId("52f65f98a4e1669bbbb0695e"),
  "nom": "Pays",
  "prenom": "Bas"
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

Le document dont le nom n'était pas indiqué a été remplacé par le nouveau `{ nom : "Jobs", prenom : "Steve" }`, en conservant le même `_id`.

Mettre à jour partiellement un document

Plutôt que de remplacer la totalité d'un document, on peut ne remplacer qu'une partie de celui-ci, par exemple seulement le champ `nom` ou le champ `prenom`. On utilise pour cela des clés spécifiques dans le paramètre `update`.

Tableau 19–8 Clés du paramètre `update`

Clé	Signification
<code>\$set</code>	Positionne les champs aux valeurs indiquées. Par exemple, <code>{ \$set : { nom : "Sarrion", prenom : "Eric" } }</code> . Si un champ n'existe pas dans le document, il est créé.
<code>\$inc</code>	Incrémente la valeur du champ, de l'incrément indiqué. Si l'incrément est négatif, le champ est décrémenté. Plusieurs champs peuvent être spécifiés avec des incréments différents pour chacun. Par exemple, <code>{ \$inc : { age : 1, max : -1 } }</code> .
<code>\$unset</code>	Supprime les champs indiqués dans les documents. Par exemple, <code>{ \$unset : { nom : "", prenom : "" } }</code> . La valeur associée à chaque champ est sans importance ici (on indique une chaîne vide mais on pourrait mettre n'importe quelle valeur à la place), car elle ne sert que pour formater l'objet au format JSON.
<code>\$rename</code>	Renomme un ou plusieurs champs dans un document. Par exemple, <code>{ \$rename : { nom : "firstname", prenom : "lastname" } }</code> . Le champ <code>nom</code> s'appellera <code>firstname</code> et le champ <code>prenom</code> deviendra <code>lastname</code> .

Utilisons ces clés afin de mettre à jour les documents de la collection `clients`.

Ajout des champs `age` et `datemaj` dans un document

```
db = connect("mydb");

db.clients.update(
  nom : "Jobs"
}, {
  $set : { age : 55, datemaj : new Date() }
});

var clients = db.clients.find();
clients.forEach(function(client) {
  printjson(client);
});
```

Le premier document ayant le `nom` positionné à "Jobs" (en fait, il en existe un seul pour l'instant) aura les champs `age` et `datemaj` positionnés aux valeurs indiquées. Ces champs sont créés s'ils n'existent pas (ce qui est le cas ici).

Figure 19–36

Ajout des champs `age` et `datemaj` pour le nom "Jobs"

```

"prenom" : "Michel",
"adresse" : [
    {
        "rue" : "17 rue Villeneuve",
        "cp" : "69000",
        "ville" : "Lyon"
    },
    {
        "rue" : "3 bd Daumesnil",
        "cp" : "75012",
        "ville" : "Paris"
    }
],
"_id" : ObjectId("529331496cd10c46bedchbda"),
"nom" : "Sarrion",
"prenom" : "Eric"

"_id" : ObjectId("529335e97b6b6ad0682be009"),
"nom" : "Rails",
"prenom" : "Ruby"

"_id" : ObjectId("52f65f98a4e1669bbb0695e"),
"nom" : "Pays",
"prenom" : "Bas"

"_id" : ObjectId("52ecf665437105c12ae9579e"),
"age" : 55,
"datemaj" : ISODate("2014-02-11T10:05:01.892Z"),
"nom" : "Jobs",
"prenom" : "Steve"

```

C:\Users\Eric\Documents\Node.js\testmongo>

Incrémenter le champ `age` de 1

```

db = connect("mydb");

db.clients.update( {
    nom : "Jobs"
}, {
    $inc : { age : 1 }
});

var clients = db.clients.find();
clients.forEach(function(client) {
    printjson(client);
});

```

Figure 19–37

Incrémenter le champ age de 1
(55 devient 56).

```

[{"prenom": "Michel",
  "adresse": [
    {
      "rue": "17 rue Villeneuve",
      "cp": "69000",
      "ville": "Lyon"
    },
    {
      "rue": "3 bd Daumesnil",
      "cp": "75012",
      "ville": "Paris"
    }
  ],
  "_id": ObjectId("529331496cd10c46bedcbda"),
  "nom": "Sarkion",
  "prenom": "Eric"
},
{
  "_id": ObjectId("529335e97b6b6ad0682be009"),
  "nom": "Rails",
  "prenom": "Ruby"
},
{
  "_id": ObjectId("52f65f98a4e1669bbbb0695e"),
  "nom": "Pays",
  "prenom": "Bas"
},
{
  "_id": ObjectId("52ecf665437105c12ae9579e"),
  "age": 56,
  "datemaj": ISODate("2014-02-11T10:05:01.892Z"),
  "nom": "Jobs",
  "prenom": "Steve"
}
C:\Users\Eric\Documents\Node.js\testmongo>
  
```

L'âge de "Steve Jobs" est passé de 55 à 56 ans.

Mettre à jour plusieurs documents simultanément

Par défaut, la méthode `update()` met à jour le premier document trouvé correspondant aux critères de recherche. Pour mettre à jour tous les documents au lieu d'un seul, il suffit d'indiquer l'option `multi` positionnée à `true` (au lieu de `false`, par défaut).

Par exemple, on souhaite modifier le libellé des champs `nom` et `prenom`, en les remplaçant par leurs équivalents anglais `firstname` et `lastname`. Cette modification doit être effectuée sur tous les documents de la collection `clients`.

Remplacer les libellés nom et prenom par firstname et lastname

```

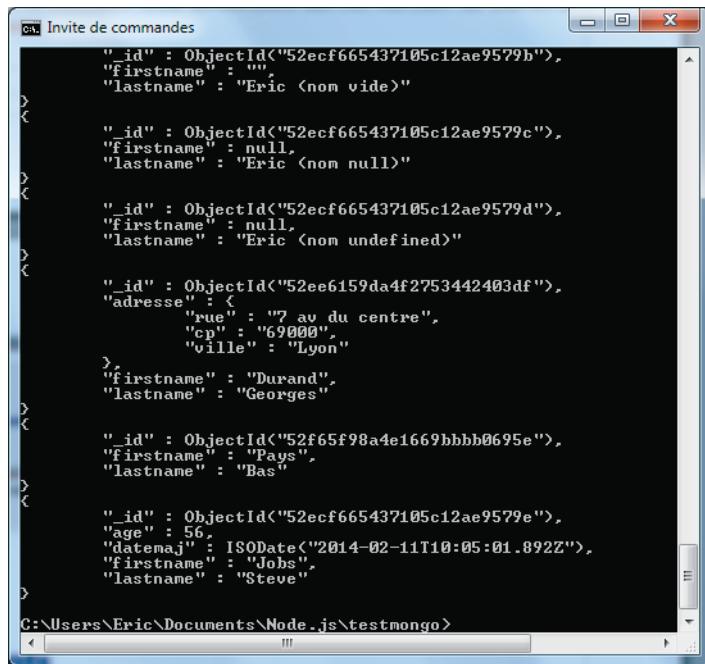
db = connect("mydb");

db.clients.update( { }, {
  $rename : { nom : "firstname", prenom : "lastname" }
}, { multi : true });

var clients = db.clients.find();
clients.forEach(function(client) {
  printjson(client);
});
  
```

Figure 19–38

Remplacer nom et prenom par
firstname et lastname



```
mongo
{
  "_id" : ObjectId("52ecf665437105c12ae9579b"),
  "firstname" : "",
  "lastname" : "Eric <nom vide>"
}

{
  "_id" : ObjectId("52ecf665437105c12ae9579c"),
  "firstname" : null,
  "lastname" : "Eric <nom null>"
}

{
  "_id" : ObjectId("52ecf665437105c12ae9579d"),
  "firstname" : null,
  "lastname" : "Eric <nom undefined>"
}

{
  "_id" : ObjectId("52ee6159da4f2753442403df"),
  "adresse" : {
    "rue" : "7 av du centre",
    "cp" : "69000",
    "ville" : "Lyon"
  },
  "firstname" : "Durand",
  "lastname" : "Georges"
}

{
  "_id" : ObjectId("52f65f98a4e1669bbbb0695e"),
  "firstname" : "Pays",
  "lastname" : "Bas"
}

{
  "_id" : ObjectId("52ecf665437105c12ae9579e"),
  "age" : 56,
  "datemaj" : ISODate("2014-02-11T10:05:01.892Z"),
  "firstname" : "Jobs",
  "lastname" : "Steve"
}
C:\Users\Eric\Documents\Node.js\testmongo>
```

Tous les documents ont maintenant leurs champs `nom` et `prenom` changés en `firstname` et `lastname`.

Supprimer des documents

La dernière action que l'on peut effectuer sur un document sera de le supprimer. Pour cela, on utilise la méthode `remove (query, justOne)` qui permet de supprimer les documents spécifiés dans l'objet `query`.

Tableau 19–9 Méthode de suppression des documents

Méthode	Signification
<code>db.nomCollection.remove(query, justOne)</code>	Supprime un document ou un ensemble de documents, correspondant aux critères définis dans l'objet <code>query</code> . Si <code>justOne</code> vaut <code>true</code> , on supprime le premier document qui satisfait la condition, sinon (valeur par défaut) on supprime tous les documents qui correspondent à cette condition.

Afin de montrer l'utilisation de `remove()`, insérons plusieurs clients ayant le nom "Rails" dans la collection `clients`. Il s'agit de "Rails Eric" et "Rails Michel", sachant que "Rails Ruby" est déjà présent dans la collection.

Insertion des clients "Rails Eric" et "Rails Michel"

```
db = connect("mydb");

db.clients.insert( { nom : "Rails", prenom : "Eric" });
db.clients.insert( { nom : "Rails", prenom : "Michel" });

var clients = db.clients.find().sort({ nom : 1 });
clients.forEach(function(client) {
  printjson(client);
});
```

Les noms des clients sont triés et affichés par ordre alphabétique afin de les retrouver plus facilement.

Figure 19-39

Insertion de deux nouveaux clients dans la collection clients

```
cat Invite de commandes
>
< 
  "rue" : "3 bd Daumesnil",
  "cp" : "75012",
  "ville" : "Paris"
>
< 
  "nom" : "Martin",
  "prenom" : "Michel"
>
< 
  "_id" : ObjectId("529335e97b6b6ad0682be00a"),
  "nom" : "Node",
  "prenom" : "Js"
>
< 
  "_id" : ObjectId("52f65f98a4e1669bbbb0695e"),
  "nom" : "Pays",
  "prenom" : "Bas"
>
< 
  "_id" : ObjectId("529335e97b6b6ad0682be009"),
  "nom" : "Rails",
  "prenom" : "Ruby"
>
< 
  "_id" : ObjectId("53062b7d353180a6dbfd4678"),
  "nom" : "Rails",
  "prenom" : "Eric"
>
< 
  "_id" : ObjectId("53062b7d353180a6dbfd4679"),
  "nom" : "Rails",
  "prenom" : "Michel"
>
< 
  "_id" : ObjectId("529331496cd10c46bedcbda"),
  "nom" : "Sarrión",
  "prenom" : "Eric"
>

C:\Users\Eric\Documents\Node.js\testmongo>
```

On retrouve ainsi les trois clients ayant le nom "Rails". Supprimons le premier client ayant ce nom, au moyen de l'instruction `remove()`.

Supprimer le premier client ayant le nom "Rails"

```
db = connect("mydb");

db.clients.remove( { nom : "Rails" }, true );

var clients = db.clients.find().sort({ nom : 1 });
clients.forEach(function(client) {
  printjson(client);
});
```

Figure 19–40

Le premier client dont le nom est "Rails" a été supprimé.

```
Invité de commandes
"adresse" : [
    {
        "rue" : "17 rue Villeneuve",
        "cp" : "69000",
        "ville" : "Lyon"
    },
    {
        "rue" : "3 bd Daumesnil",
        "cp" : "75012",
        "ville" : "Paris"
    }
],
"nom" : "Martin",
"prenom" : "Michel"

{
    "_id" : ObjectId("529335e97b6b6ad0682be00a"),
    "nom" : "Node",
    "prenom" : "Js"
}

{
    "_id" : ObjectId("52f65f98a4e1669bbbb0695e"),
    "nom" : "Pays",
    "prenom" : "Bas"
}

{
    "_id" : ObjectId("53062b7d353180a6dbfd4678"),
    "nom" : "Rails",
    "prenom" : "Eric"
}

{
    "_id" : ObjectId("53062b7d353180a6dbfd4679"),
    "nom" : "Rails",
    "prenom" : "Michel"
}

{
    "_id" : ObjectId("529331496cd10c46bedccbda"),
    "nom" : "Sarrión",
    "prenom" : "Eric"
}

C:\Users\Eric\Documents\Node.js\testmongo>
```

Le client ayant le nom "Rails" et le prénom "Ruby" a été supprimé. Les autres clients ayant le nom "Rails" ont été conservés car on a positionné le paramètre `justOne` à `true`. Positionnons ce paramètre à `false` (valeur par défaut) pour supprimer tous les clients ayant ce nom.

Supprimer tous les clients dont le nom est "Rails"

```
db = connect("mydb");
```

```
db.clients.remove( { nom : "Rails" } ); // justOne vaut false par défaut

var clients = db.clients.find().sort({ nom : 1 });
clients.forEach(function(client) {
  printjson(client);
});
```

Figure 19–41

Tous les clients dont le nom est "Rails" ont été supprimés.

```
C:\Users\Eric\Documents\Node.js\testmongo> mongo
MongoDB shell version: 2.6.1
connecting to: test
> db.test.insert({nom: "Steve", prenom: "Steve", age: 56, dateNaissance: ISODate("2014-02-11T00:00:00Z"), adresse: [{"rue": "17 rue Villeneuve", "ville": "Lyon", "cp": "69000"}, {"rue": "3 bd Daumesnil", "ville": "Paris", "cp": "75012"}], nom: "Martin", prenom: "Michel"})
{
  "_id": ObjectId("52ecf665437105c12ae9579e"),
  "age": 56,
  "dateaj": ISODate("2014-02-11T00:00:00Z"),
  "nom": "Jobs",
  "prenom": "Steve"
}

{
  "_id": ObjectId("52ee7158b1e7e0cf70fd0b"),
  "adresse": [
    {
      "rue": "17 rue Villeneuve",
      "ville": "Lyon",
      "cp": "69000"
    },
    {
      "rue": "3 bd Daumesnil",
      "ville": "Paris",
      "cp": "75012"
    }
  ],
  "nom": "Martin",
  "prenom": "Michel"
}

{
  "_id": ObjectId("529335e97b6b6ad0682be00a"),
  "nom": "Node",
  "prenom": "Js"
}

{
  "_id": ObjectId("52f65f98a4e1669bbbb0695e"),
  "nom": "Pay",
  "prenom": "Bas"
}

{
  "_id": ObjectId("529331496cd10c46bedcbdb"),
  "nom": "Sarpon",
  "prenom": "Eric"
}
```

Tous les clients de nom "Rails" ont été supprimés.

Actions globales sur une collection

Dans les pages précédentes, nous avons étudié les possibilités offertes sur les documents contenus dans une collection, en particulier les quatre opérations de base : lecture, écriture, modification et suppression de documents.

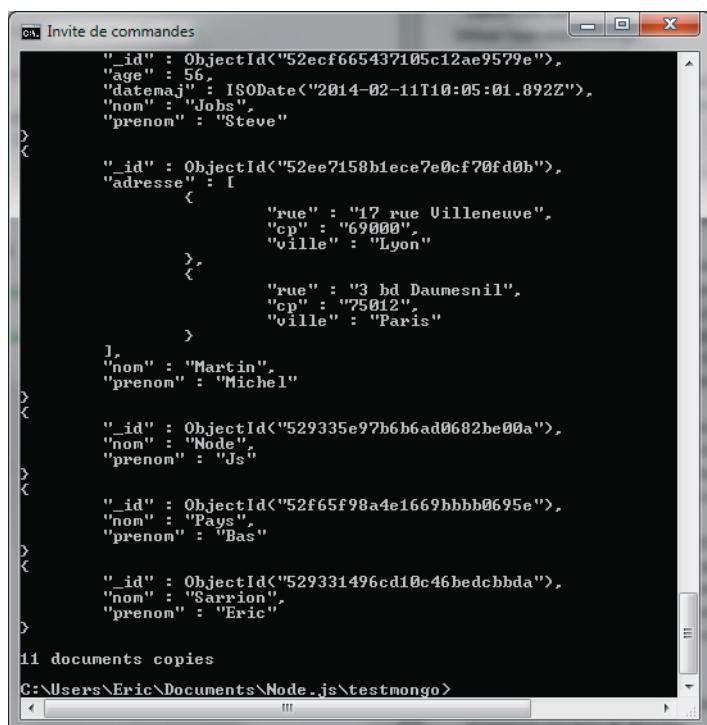
Il existe des opérations plus générales au niveau de la collection. Par exemple, copier une collection dans une autre, renommer une collection voire la supprimer.

Tableau 19–10 Méthodes globales sur les collections

Méthode	Signification
<code>db.nomCollection.copyTo(newCollection)</code>	Recopie les éléments de <code>nomCollection</code> dans la collection <code>newCollection</code> (chaîne de caractères). La nouvelle collection devient identique à la collection d'origine (les valeurs des champs <code>_id</code> sont conservées). Si <code>newCollection</code> existe avant la copie, ses documents sont remplacés par ceux de <code>nomCollection</code> . Retourne le nombre de documents copiés.
<code>db.nomCollection.drop()</code>	Supprime la collection <code>nomCollection</code> .
<code>db.nomCollection.renameCollection(newCollection)</code>	Renomme la collection dans le nouveau nom <code>newCollection</code> . Le nouveau nom ne doit pas déjà exister.

Figure 19–42

Copie d'une collection dans une autre



The screenshot shows the MongoDB shell interface with the title "Invite de commandes". The command entered is:

```
db.jobs.copyTo("users")

```

The output displays the copied documents from the "jobs" collection to the "users" collection:

```

{
  "_id": ObjectId("52ecf665437105c12ae9579e"),
  "age": 56,
  "datemaj": ISODate("2014-02-11T10:05:01.892Z"),
  "nom": "Jobs",
  "prenom": "Steve"
}

{
  "_id": ObjectId("52ee7158biece7e0cf70fd0b"),
  "adresse": [
    {
      "rue": "17 rue Villeneuve",
      "cp": "69000",
      "ville": "Lyon"
    },
    {
      "rue": "3 bd Daumesnil",
      "cp": "75012",
      "ville": "Paris"
    }
  ],
  "nom": "Martin",
  "prenom": "Michel"
}

{
  "_id": ObjectId("529335e97b6b6ad0682be00a"),
  "nom": "Node",
  "prenom": "Js"
}

{
  "_id": ObjectId("52f65f98a4e1669bbbb0695e"),
  "nom": "Pays",
  "prenom": "Bas"
}

{
  "_id": ObjectId("529331496cd10c46bedchbda"),
  "nom": "Saxion",
  "prenom": "Eric"
}

11 documents copies

```

The command prompt at the bottom is "C:\Users\Eric\Documents\Node.js\testmongo>".

Utiliser copyTo() pour dupliquer la collection clients dans clients_save

```
db = connect("mydb");

var nb = db.clients.copyTo("clients_save");

var clients = db.clients_save.find().sort({ nom : 1 });
clients.forEach(function(client) {
    printjson(client);
});

print();
print(nb + " documents copies");
```

Actions globales sur une base de données

On peut agir au niveau global d'une base de données. Par exemple, pour obtenir la liste des collections, dupliquer une base de données complète, voire la supprimer.

Les méthodes associées aux bases de données sont préfixées directement de l'objet `db`, sans indication de la collection.

Tableau 19-11 Méthodes globales sur les bases de données

Méthode	Signification
<code>db.getCollectionNames()</code>	Retourne un tableau contenant les noms des collections présentes dans la base de données active.
<code>db.getName()</code>	Retourne le nom de la base de données active.
<code>db.copyDatabase(origin, destination, hostname)</code>	Copie la base de données <code>origin</code> dans la base de données <code>destination</code> . Le paramètre <code>hostname</code> est optionnel et sert à indiquer l'adresse IP de la base de données <code>origin</code> , si elle n'est pas située sur le serveur qui exécute l'instruction. La base de données <code>destination</code> sera située sur le serveur actuel qui exécute l'instruction.
<code>db.dropDatabase()</code>	Supprime la base de données active.
<code>connect(nomDatabase)</code>	Active la base de données indiquée, sur laquelle s'appliqueront les instructions préfixées par l'objet <code>db</code> .

20

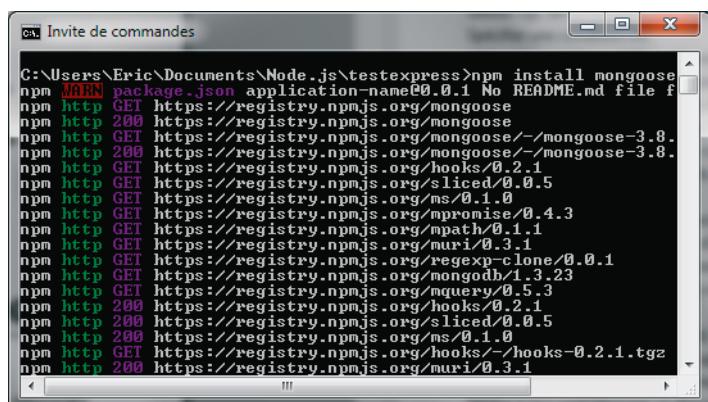
Introduction au module Mongoose

La base de données MongoDB, installée dans le chapitre précédent, peut être utilisée avec plusieurs types de serveurs, et en particulier avec Node.js. Différents modules Node fournissent une interface avec MongoDB. Nous étudions dans cette partie le module Mongoose qui offre un accès simple mais performant à MongoDB.

Installer le module Mongoose

Mongoose étant un module Node, on l'installe au moyen de la commande `npm`, en tapant `npm install mongoose`.

Figure 20-1
Installation du module
Mongoose



The screenshot shows a Windows Command Prompt window titled "Invite de commandes". The command entered is "C:\Users\Eric\Documents\Node.js\testexpress>npm install mongoose". The output shows the installation process, including requests for various dependencies from the npm registry. The output is as follows:

```
C:\Users\Eric\Documents\Node.js\testexpress>npm install mongoose
npm WARN package.json application-name@0.0.1 No README.md file found
npm http GET https://registry.npmjs.org/mongoose
npm http 200 https://registry.npmjs.org/mongoose
npm http GET https://registry.npmjs.org/mongoose/-/mongoose-3.8.
npm http 200 https://registry.npmjs.org/mongoose/-/mongoose-3.8.
npm http GET https://registry.npmjs.org/hooks/0.2.1
npm http GET https://registry.npmjs.org/sliced/0.0.5
npm http GET https://registry.npmjs.org/ms/0.1.0
npm http GET https://registry.npmjs.org/mpromise/0.4.3
npm http GET https://registry.npmjs.org/mpath/0.1.1
npm http GET https://registry.npmjs.org/muri/0.3.1
npm http GET https://registry.npmjs.org/regexp-clone/0.0.1
npm http GET https://registry.npmjs.org/mongodb/1.3.23
npm http GET https://registry.npmjs.org/mquery/0.5.3
npm http 200 https://registry.npmjs.org/hooks/0.2.1
npm http 200 https://registry.npmjs.org/sliced/0.0.5
npm http 200 https://registry.npmjs.org/ms/0.1.0
npm http GET https://registry.npmjs.org/hooks/-/hooks-0.2.1.tgz
npm http 200 https://registry.npmjs.org/muri/0.3.1
```

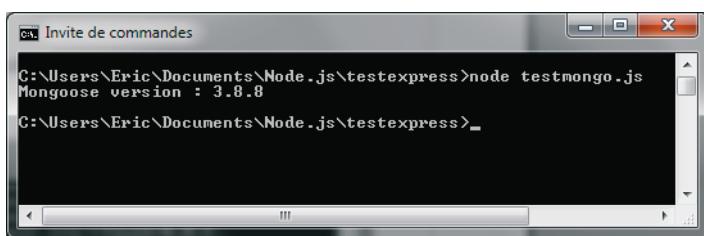
Vérifions que Mongoose est installé et accessible. Affichons le numéro de la version utilisée.

Afficher le numéro de version de Mongoose

```
var mongoose = require("mongoose");
console.log("Mongoose version : " + mongoose.version);
```

Nous insérons le module Mongoose au moyen de l'instruction `require("mongoose")`, puis nous accédons au numéro de version inscrit dans la propriété `version` du module.

Figure 20–2
Affichage de la version
du module Mongoose



Établir une connexion à la base de données avec Mongoose

Une connexion à MongoDB est nécessaire pour effectuer l'une des quatre opérations CRUD (créer, lire, mettre à jour et supprimer) dans celle-ci. On obtient une connexion à une base de données MongoDB en utilisant la méthode `mongoose.connect(url)` du module `mongoose`.

Tableau 20–1 Méthode connect() sur la base de données

Méthode	Signification
<code>mongoose.connect(url, options)</code>	Crée une connexion à la base de données représentée par l'URL. Cette dernière est de la forme <code>mongodb://nomDomaine/nomDatabase</code> , par exemple <code>mongodb://localhost/mydb</code> pour accéder à la base de données <code>mydb</code> sur le serveur <code>localhost</code> . La base de données peut ne pas exister sur le serveur (elle sera créée lors de la création du premier document). Les options sont un objet optionnel, ayant les propriétés suivantes : - <code>options.user</code> : nom de l'utilisateur qui accède à la base de données ; - <code>options.pass</code> : mot de passe de l'utilisateur ; - <code>options.server.socketOptions.keepAlive</code> : positionné à 1, il permet de garder une connexion active à la base de données sans déconnexion intempestive.

Par exemple, établissons une connexion à la base de données `mydb` que nous avons utilisée dans le chapitre précédent. Nous affichons également le contenu de l'objet retourné par la méthode `connect()`.

Établir une connexion à la base de données mydb

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

console.log(db);
```

Figure 20–3

Connexion à la base de données avec Mongoose

L'objet retourné par la méthode `connect()` correspond au module `mongoose` lui-même. En effet, si nous affichons celui-ci dans la console, nous voyons les mêmes informations apparaître. Ceci permet de chaîner l'appel à d'autres méthodes du module, à la suite de celle-ci.

Il est possible de gérer des événements lors de la connexion à la base de données. L'objet `mongoose` associé au module contient une propriété `connection` sur laquelle on peut attacher la gestion des événements `error` et `open`.

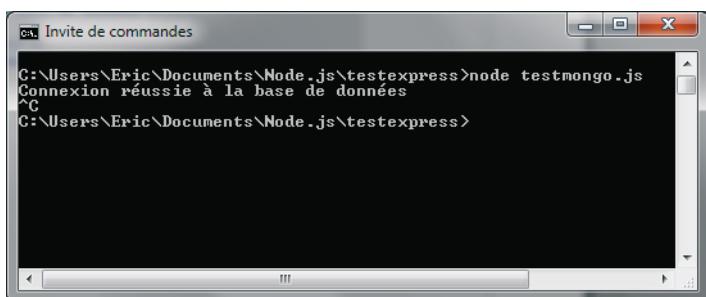
Gérer les événements error et open sur la base de données

```
var mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/mydb");
```

```
mongoose.connection.on("error", function() {
  console.log("Erreur de connexion à la base de données")
});
mongoose.connection.on("open", function() {
  console.log("Connexion réussie à la base de données");
});
```

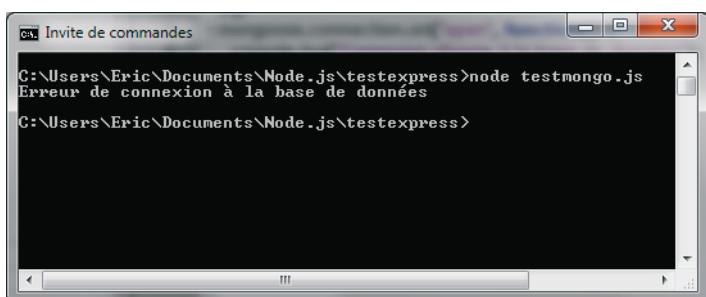
Si la connexion est établie, l'événement `open` est déclenché.

Figure 20–4
Connexion réussie
à la base de données



Dans le cas où l'URL n'existe pas, si on indique un nom de serveur inexistant (par exemple, `localhost1` au lieu de `localhost`), l'événement `error` est déclenché.

Figure 20–5
Échec de la connexion
à la base de données



Utiliser les schémas et les modèles

On sait qu'un document MongoDB peut être décrit sous la forme d'un objet JSON quelconque. De plus, chaque document inséré dans une collection est indépendant des autres documents de la même collection, et peut ainsi avoir une structure différente des autres documents de cette collection.

Toutefois, dans la pratique, il est rare que les documents d'une même collection aient une structure radicalement différente les uns des autres. En effet, le fait qu'ils soient tous regroupés au sein d'une même collection signifie qu'ils partagent une structure commune. Par exemple, les documents de la collection `clients` utilisée dans le chapitre précédent, avaient tous en commun le nom, le prénom et l'adresse d'un client. Ceci n'empêche pas que des champs soient renseignés dans certains documents, et non renseignés dans d'autres. Ainsi, le champ `adresse` n'était pas renseigné dans tous les documents, mais uniquement dans certains d'entre eux.

Définir un schéma

Constatant cela, les créateurs de Mongoose ont proposé que l'on définisse la structure des documents d'une collection en indiquant les champs qui pourront y figurer. Cette définition s'appelle un « schéma » et on peut créer un schéma particulier au moyen de la classe `mongoose.Schema(definition, options)` définie dans le module `mongoose`.

Tableau 20–2 Définir un schéma

Méthode	Signification
<code>mongoose.Schema(definition, options)</code>	Classe permettant de définir un schéma. L'objet <code>definition</code> permet de définir les champs qui seront présents dans les documents associés à ce schéma. L'objet <code>options</code> (optionnel) permet de modifier le comportement par défaut du schéma créé : - <code>options.collection</code> : définit le nom de la collection associée à ce schéma. Par défaut, le nom est celui du modèle associé (voir ci-dessus) mis au pluriel. - <code>options.strict</code> : si <code>true</code> (valeur par défaut), indique de ne pas sauvegarder dans la base les champs non inscrits dans les schémas. Ceci permet que les documents sauvegardés respectent le schéma associé.

Créons le schéma associé aux clients. Il comportera les champs `nom`, `prenom` et `adresse`, qui sont tous des chaînes de caractères.

Créer le schéma associé aux clients

```
var mongoose = require("mongoose");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});
```

Le type de chaque champ est ici défini au moyen de la classe `String`. D'autres classes existent, elles sont listées dans le tableau 20-3.

Tableau 20-3 Classes permettant de définir le type des champs dans un schéma

Classe	Signification
<code>String</code>	Définit une chaîne de caractères.
<code>Number</code>	Définit un champ numérique.
<code>Boolean</code>	Définit un booléen.
<code>Array</code>	Définit un tableau.
<code>Buffer</code>	Définit un buffer d'octets.
<code>Date</code>	Définit une date.
<code>mongoose.Schema.Types.ObjectId</code>	Définit un identifiant (tel que <code>_id</code>).
<code>mongoose.Schema.Types.Mixed</code>	Définit un type quelconque. La valeur pourra être de n'importe quel type.

Les deux dernières classes sont internes à Mongoose, elles doivent donc être utilisées en indiquant le chemin complet dans le module `mongoose`. Les autres classes sont internes à Node et sont donc directement accessibles.

Une fois le schéma défini, il faut l'associer à une collection. Ceci est effectué grâce au modèle.

Définir un modèle

Un modèle sera la représentation sous forme de classe d'une collection de documents. Mongoose permet d'associer un (voire plusieurs) modèle à un schéma. On utilise pour cela la méthode `mongoose.model(modelName, schema)` qui retourne une classe JavaScript qui permettra de créer des documents associés à ce modèle.

Tableau 20-4 Définir un modèle

Méthode	Signification
<code>mongoose.model(modelName, schema)</code>	Crée un modèle associé au schéma indiqué. Le modèle est une classe JavaScript qui servira à créer des documents se basant sur le schéma. La classe associée au modèle est retournée par la méthode. La collection associée à ce modèle portera le nom <code>modelName</code> (transformé en minuscules et mis au pluriel). Par exemple, si <code>modelName</code> est "Client", le nom de la collection dans MongoDB sera "clients". Le nom de la collection peut être modifié lors de la création du schéma, en utilisant l'option <code>collection</code> (voir section précédente).

Créons le modèle `Client` associé au schéma `clientSchema` créé précédemment.

Créer le modèle Client à partir du schéma clientSchema

```
var mongoose = require("mongoose");

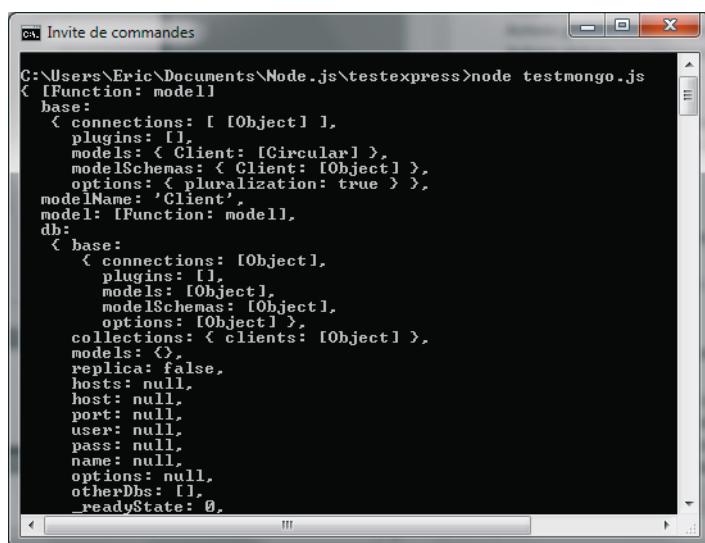
var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});

var Client = mongoose.model("Client", clientSchema);

console.log(Client);
```

Il est de coutume de nommer la classe renvoyée par `mongoose.model()` avec le même nom que celui du modèle indiqué en paramètre. Toutefois, le nom indiqué en paramètre sert uniquement à identifier le nom de la collection qui sera utilisée par MongoDB (en transformant ce nom en minuscules et terminé par la lettre "s").

Figure 20–6
Caractéristiques du modèle Client



```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
< Function: model>
  base:
    < Object> {
      connections: [Object],
      plugins: [],
      models: { Client: [Circular] },
      modelSchemas: { Client: [Object] },
      options: { pluralization: true },
      modelName: 'Client',
      model: [Function: model],
      db:
        < Object> {
          base:
            < Object> {
              connections: [Object],
              plugins: [],
              models: [Object],
              modelSchemas: [Object],
              options: [Object],
              collections: { clients: [Object] },
              models: {},
              replica: false,
              hosts: null,
              host: null,
              port: null,
              user: null,
              pass: null,
              name: null,
              options: null,
              otherDbs: [],
              _readyState: 0
            }
        }
```

L'objet `Client` renvoyé par `mongoose.model()` est une fonction, qui sera utilisée ici en tant que classe JavaScript. C'est pour cela qu'il est de tradition d'indiquer un nom de variable commençant par une lettre majuscule, permettant ainsi de le faire comprendre dans le code de notre programme.

On voit dans les propriétés affichées que le nom de la collection utilisée est `clients`, ce qui montre le mécanisme de transformation entre le nom du modèle transmis en paramètre (ici, "`Client`") et le nom de la collection (ici, "`clients`").

Il reste maintenant à utiliser ces connaissances afin d'interagir avec la base de données MongoDB pour créer des documents. C'est l'objet du prochain chapitre.

21

Créer des documents

Dans ce chapitre, nous allons voir comment créer des documents à l'aide du module `mongoose` précédemment installé. Nous utiliserons les schémas et les modèles décrits dans le chapitre précédent.

Deux moyens sont possibles pour créer de nouveaux documents :

- en utilisant la méthode d'instance `save()` ;
- en utilisant la méthode de classe `create()`.

Créer un document en utilisant la méthode d'instance `save()`

Insérer un document dans la collection

La première façon de créer un document dans la base de données consiste à utiliser la méthode `save()` sur l'objet document créé en mémoire. Un document associé à un client sera créé en mémoire en utilisant l'opérateur `new` sur la classe du modèle associé. Créons ainsi un document associé à un client, puis sauvegardons-le dans la base de données au moyen de l'instruction `save()`.

Créer un client dans la base de données

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});

var Client = mongoose.model("Client", clientSchema);
var c = new Client({ nom : "Obama", prenom : "Barack" });
c.adresse = "Washington";
c.save();
```

Une connexion à la base de données utilisée (ici, `mydb`) est obligatoire, sinon rien ne se passe...

L'instruction `c = new Client()` permet de créer un document `c` en mémoire associé à un client. Les champs associés à ce client sont indiqués dans un objet en premier paramètre de la classe du modèle (ici, `{ nom : "Obama", prenom : "Barack" }`). L'objet `c` associé à ce document peut être modifié, comme ici en ajoutant le champ `adresse` dans le document. Puis nous sauvegardons le document à l'aide de la méthode d'instance `c.save()`, utilisée sur le document lui-même.

Le document est sauvegardé dans la collection `clients` de la base de données `mydb` de MongoDB. Le nom de la collection est obtenu à partir du nom du modèle, en le transformant en minuscules et mis au pluriel. Toutefois, si le nom de la collection attribué par Mongoose ne vous satisfait pas, il suffit d'indiquer l'option `collection` lors de la définition du schéma (voir la section « Définir un schéma » dans le chapitre précédent).

Récupérer la liste des documents de la collection

Écrivons maintenant le programme permettant de visualiser les clients insérés dans la base de données. On utilise pour cela la méthode de classe `find()` définie sur le modèle lui-même (une description complète des possibilités de recherche avec Mongoose sera détaillée dans le chapitre suivant).

Afficher les clients de la collection clients

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");
```

```

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});

var Client = mongoose.model("Client", clientSchema);
Client.find(function (err, clients) {
  if (err) return console.error(err);
  console.log(clients);
});

```

La méthode `find()` s'utilise directement sur le modèle, ici la classe `Client`. La méthode `find()` est appelée méthode de classe, à la différence de la méthode `save()` qui est une méthode d'instance.

Figure 21–1
Affichage de tous les documents de la collection clients

```

cat Invite de commandes - node testmongo.js
< _id: 52ee7158b1ece7e0cf70fd0b,
  adresse: [object Object],[object Object],
  nom: 'Martin',
  prenom: 'Michel' >
< _id: 529331496cd10c46bedcbdbda, nom: 'Sarrion', prenom: 'Eric' >
< _id: 529335e97b6b6ad0682be00a, nom: 'Node', prenom: 'Js' >
< _id: 52ecf665437105c12ae9579b,
  nom: '',
  prenom: 'Eric <nom vide>' >
< _id: 52ecf665437105c12ae9579c,
  nom: null,
  prenom: 'Eric <nom null>' >
< _id: 52ecf665437105c12ae9579d,
  prenom: 'Eric <nom undefined>' >
< _id: 52ee6159da4f2753442403df,
  adresse: [object Object],
  nom: 'Durand',
  prenom: 'Georges' >
< _id: 52f65f98a4e16693bbb0695e, nom: 'Pays', prenom: 'Bas' >
< _id: 52ecf665437105c12ae9579e,
  age: 56,
  datenaj: Tue Feb 11 2014 11:05:01 GMT+0100 (Paris, Madrid),
  nom: 'Jobs',
  prenom: 'Steve' >
< _id: 531c9ac8d7d90bb41c6a1ce5,
  adresse: 'Washington',
  nom: 'Obama',
  prenom: 'Barack',
  __v: 0 > > 1

```

On constate l'ajout d'un nouveau document dans la collection. Ce document possède en plus le champ `_v` positionné à 0, qui est un champ géré par Mongoose afin de donner un numéro de version à chaque document inséré à partir de l'API Mongoose.

Insérer un document, puis récupérer la liste des documents de la collection

Dans l'exemple précédent, nous avons procédé en deux étapes.

- Nous avons tout d'abord créé le document dans la collection `clients`.

- Puis nous avons récupéré la liste des clients, incluant le nouveau client.

Ces deux étapes correspondent à deux programmes différents. Regroupons ces morceaux de programmes pour n'en faire qu'un seul.

Insérer, puis afficher la liste des clients

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});

var Client = mongoose.model("Client", clientSchema);
var c = new Client({ nom : "Obama2", prenom : "Barack2" });
c.adresse = "Washington";
c.save();

Client.find(function (err, clients) {
  if (err) return console.error(err);
  console.log(clients);
});
```

Nous avons simplement mis à la suite, l'insertion du nouveau client et l'appel à la fonction de recherche. Exécutons le programme correspondant.

Figure 21–2

Insertion d'un nouveau document, mais il n'apparaît pas.

```
ca Invité de commandes - node testmongo.js
< _id: 52ee7158b1ece7e0cf70fd0b,
  adresse: 'Object Object1, Object Object1',
  nom: 'Martin',
  prenom: 'Miche1' >
< _id: 529331496cd10c46bedcbbda, nom: 'Sarrion', prenom: 'Eric' >
< _id: 529335e97b6b6ad0602be00a, nom: 'Node', prenom: 'Js' >
< _id: 52ecf665437105c12ae9579b,
  nom: '',
  prenom: 'Eric <nom vide>' >
< _id: 52ecf665437105c12ae9579c,
  nom: null,
  prenom: 'Eric <nom null>' >
< _id: 52ecf665437105c12ae9579d,
  prenom: 'Eric <nom undefined>' >
< _id: 52ee6159da4f2753442403df,
  adresse: 'Object Object1',
  nom: 'Dupand',
  prenom: 'Georges' >
< _id: 52f65f98a4e1669bbbb0695e, nom: 'Pays', prenom: 'Bas' >
< _id: 52ecf665437105c12ae9579e,
  age: 56,
  datemaj: Tue Feb 11 2014 11:05:01 GMT+0100 (Paris, Madrid),
  nom: 'Jobs',
  prenom: 'Steve' >
< adresse: 'Washington',
  nom: 'Obama',
  prenom: 'Barack',
  _id: 531c9ac8d7d90bb41c6a1ce5,
  __v: 0 > ]
```

C'est le même affichage que le précédent, qui semble ne pas tenir compte de l'ajout de notre nouveau client ("Obama2"). En réalité, le client a bien été sauvegardé dans la collection `clients`, mais l'appel de la méthode `find()` est effectué trop tôt, avant que le client n'ait fini d'être sauvegardé dans la base de données. Il faudrait donc effectuer l'appel de la méthode `find()` uniquement lorsque le client a fini d'être sauvegardé par la méthode `save()`.

Rappelez-vous que Node délègue le maximum de tâches aux autres processus du système, afin de ne pas bloquer la boucle principale de traitement du programme. Si Node devait attendre que le document soit écrit dans la base de données pour continuer la suite d'instructions, il y aurait un blocage inutile du programme. Entre le moment où l'on demande la sauvegarde du document dans la base de données et le moment où c'est effectif, d'autres requêtes d'utilisateurs auront ainsi pu être traitées par Node.

Pour cela, la méthode `save(callback)` possède une fonction de callback en paramètre qui est appelée par Mongoose uniquement lorsque le document a été écrit dans la base de données. La forme générale de la méthode `save()` est donc la suivante.

Tableau 21–1 Méthode d'instance `save(callback)`

Méthode	Signification
<code>doc.save(callback)</code>	Crée le document dans la collection associée au modèle. La fonction de callback est appelée à la fin de la sauvegarde du document, quel que soit le résultat. Elle est de la forme <code>function(err, doc)</code> dans laquelle <code>err</code> contient l'erreur ou <code>null</code> si pas d'erreur.

On peut alors écrire, dans la fonction de callback de la méthode `save()`, le traitement lié à la recherche des documents et les afficher.

Afficher les documents dans la fonction de callback de la méthode `save()`

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});

var Client = mongoose.model("Client", clientSchema);
var c = new Client({ nom : "Obama3", prenom : "Barack3" });
c.adresse = "Washington";
c.save(function() {
```

```

Client.find(function (err, clients) {
  if (err) return console.error(err);
  console.log(clients);
});
});

```

Figure 21–3

Le document inséré est maintenant visible.

```

nom: null,
prenom: 'Eric <nom null>',
_id: 52ecf665437105c12ae9579d,
prenom: 'Eric <nom undefined>',
_id: 52ee6159da4f2753442403df,
adresse: 'Object Object',
nom: 'Durand',
prenom: 'Georges',
_id: 52f665f98a4e1669bb0695e, nom: 'Pays', prenom: 'Bas',
_id: 52ecf665437105c12ae9579e,
age: 56,
datemaj: Tue Feb 11 2014 11:05:01 GMT+0100 (Paris, Madrid),
nom: 'Jobs',
prenom: 'Steve',
adresse: 'Washington',
nom: 'Obama',
prenom: 'Barack',
_id: 531c9ac8d7d90bb41c6a1ce5,
_v: 0,
adresse: 'Washington',
nom: 'Obama2',
prenom: 'Barack2',
_id: 531c9f107bb1d6b8295f1417,
_v: 0,
adresse: 'Washington',
nom: 'Obama3',
prenom: 'Barack3',
_id: 531ca2af807cdf442b5a117c,
_v: 0
}

```

Le dernier client inséré (ici, "Obama3") est maintenant directement visible après son insertion.

Créer un document en utilisant la méthode de classe `create`

Dans la section précédente, nous avons vu comment sauvegarder un document au moyen de la méthode d'instance `save()`. Une méthode d'instance est appelée sur un objet, à la différence d'une méthode de classe qui est directement appelée sur la classe (bien qu'en JavaScript, la classe soit également un objet de classe `Function`).

Mongoose permet de sauvegarder un nouveau document en utilisant la méthode de classe `create(doc, callback)`. Le paramètre `doc` correspond au document en mémoire que l'on désire sauvegarder, tandis que la fonction de callback est appelée à la fin de la sauvegarde du document.

Tableau 21–2 Méthode de classe create(doc, callback)

Méthode	Signification
<code>ModelName.create(doc, callback)</code>	Crée le document dans la collection associée au modèle de nom <code>ModelName</code> . La fonction de callback est appelée à la fin de la sauvegarde du document, quel que soit le résultat. Elle est de la forme <code>function(err, doc)</code> dans laquelle <code>err</code> contient l'erreur ou <code>null</code> si pas d'erreur, tandis que <code>doc</code> est le document sauvégarde si succès.

Utiliser la méthode create() pour insérer des documents

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});

var Client = mongoose.model("Client", clientSchema);
Client.create({ nom : "Obama4", prenom : "Barack4", adresse : "Washington" },
  function() {
    Client.find(function (err, clients) {
      if (err) return console.error(err);
      console.log(clients);
    });
  }
);
```

Nous utilisons la fonction de callback pour lister les documents de la collection.

Figure 21–4

Création d'un nouveau document par Client.create()

```
adresse: '[object Object]',
nom: 'Durand',
prenom: 'Georges' ,
_id: 52f65f98a4e1669bbb0695e, nom: 'Pays', prenom: 'Bas' ,
_id: 52ecf665437105c12ae9579e,
age: 56,
datenaj: Tue Feb 11 2014 11:05:01 GMT+0100 (Paris, Madrid),
nom: 'Jobs',
prenom: 'Steve' ,
adresse: 'Washington',
nom: 'Obama',
prenom: 'Barack',
_id: 531c9ac8d7d90bb41c6a1ce5,
__v: 0 },
adresse: 'Washington',
nom: 'Obama2',
prenom: 'Barack2',
_id: 531c9fb07bb1d6b8295f1417,
__v: 0 },
adresse: 'Washington',
nom: 'Obama3',
prenom: 'Barack3',
_id: 531ca2af807cdf442b5a117c,
__v: 0 },
nom: 'Obama4',
prenom: 'Barack4',
adresse: 'Washington',
_id: 531cab5c6d9c32742b9ef366,
__v: 0 }
```

Le document "Obama4" est visible directement après son insertion, grâce à la fonction de callback indiquée dans la méthode `create()`.

Créer des sous-documents

Un sous-document est un document à l'intérieur d'un autre document. Mongoose permet d'en créer, à condition que le sous-document soit dans un tableau du document parent.

L'intérêt de créer des sous-documents est de pouvoir y accéder directement comme pour un document classique. En effet, Mongoose leur affecte un attribut `_id` qui les identifie de façon unique.

Considérons qu'un client soit associé à un `nom`, `prénom` et `adresse`, celle-ci étant composée des trois champs `rue`, `ville` et `cp` (code postal). Le champ `adresse` sera considéré dans cet exemple comme un sous-document du document `client`. Il aura un schéma associé décrit comme ceci.

Utiliser un sous-document dans la collection clients

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var adresseSchema = mongoose.Schema({
  rue : String,
  ville : String,
  cp : String
});

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : [adresseSchema]
});

var Client = mongoose.model("Client", clientSchema);
Client.create({
  nom : "Eastwood",
  prenom : "Clint",
  adresse : [{ rue : "rue de la mer", ville : "Santa Monica" }] }, function() {
  Client.find(function (err, clients) {
    if (err) return console.error(err);
    console.log(clients);
  });
});
```

Le document associé à l'adresse est décrit dans un schéma `adresseSchema`, auquel fait référence le document parent à travers `clientSchema`. Remarquez que le schéma `adresseSchema` doit se trouver dans un tableau du schéma parent, sinon cela ne fonctionne pas.

Figure 21–5

Créer un document contenant un sous-document (possédant le champ `_id`)

```

[{"nom": "Obama", "prenom": "Barack", "adresse": [{"rue": "1600 Pennsylvania Avenue NW", "ville": "Washington"}]}, {"nom": "Obama2", "prenom": "Barack2", "adresse": [{"rue": "1600 Pennsylvania Avenue NW", "ville": "Washington"}]}, {"nom": "Obama3", "prenom": "Barack3", "adresse": [{"rue": "1600 Pennsylvania Avenue NW", "ville": "Washington"}]}, {"nom": "Obama4", "prenom": "Barack4", "adresse": [{"rue": "1600 Pennsylvania Avenue NW", "ville": "Washington"}]}]
  
```

Le sous-document comporte un champ `_id`, montrant ainsi qu'il s'agit bien d'un sous-document.

Une autre alternative serait de ne pas créer un sous-document au sens strict, mais plutôt une sous-structure dans le document principal. On écrirait alors :

Créer une sous-structure dans un document

```

var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});
  
```

```

var Client = mongoose.model("Client", clientSchema);
Client.create({
  nom : "Eastwood2",
  prenom : "Clint2",
  adresse : {
    rue : "rue de la mer",
    ville : "Santa Monica"
  } }, function() {
  Client.find(function (err, clients) {
    if (err) return console.error(err);
    console.log(clients);
  });
});

```

Le sous-document n'est plus décrit par un schéma, il est maintenant décrit dans le document principal.

Figure 21-6

Créer un sous-document qui est une sous-structure (pas de champ `_id`).

```

4. Invite de commandes - node testmongo.js
{
  adresse: 'Washington',
  nom: 'Obama2',
  prenom: 'Barack2',
  _id: 531c9fb07bb1d6b8295f1417,
  __v: 0,
  adresse: 'Washington',
  nom: 'Obama3',
  prenom: 'Barack3',
  _id: 531ca2af807cdf442b5a117c,
  __v: 0,
  adresse: 'Washington',
  nom: 'Obama4',
  prenom: 'Barack4',
  _id: 531cab5c6d9c32742b9ef366,
  __v: 0,
  adresse: 'Washington',
  nom: 'Eastwood',
  prenom: 'Clint',
  _id: 531d94a5386699202ca38897,
  __v: 0,
  adresse:
    [
      {
        _id: 531d94a5386699202ca38898,
        ville: 'Santa Monica',
        rue: 'rue de la mer'
      }
    ],
  nom: 'Eastwood2',
  prenom: 'Clint2',
  _id: 531da064ec79e4842dc3834f,
  __v: 0,
  adresse: { rue: 'rue de la mer', ville: 'Santa Monica' }
}

```

Le sous-document n'étant plus décrit par un schéma, il ne possède plus de champ `_id` automatiquement attribué par Mongoose à un document.

22

Rechercher des documents

Une fois des documents insérés dans une collection, il est naturel de vouloir les rechercher pour les utiliser. Mongoose utilise le modèle précédemment créé afin de permettre des recherches de documents au sein de la base de données. Il utilise également la syntaxe que nous avions décrite dans le chapitre précédent pour exprimer les conditions de la recherche.

Deux méthodes sont disponibles pour effectuer une recherche : `find()` et `findOne()`, disponibles sur le modèle. Elles s'utilisent avec ou sans fonction de callback en paramètre, comme on va le voir dans les lignes qui suivent.

Utiliser la méthode `find(conditions, callback)`

Cette forme de la méthode `find()` est similaire à celle que nous avons utilisée dans le chapitre précédent. Les conditions de recherche sont exprimées dans un objet `conditions` au format JSON, tandis que la fonction de callback est appelée lorsque la recherche est terminée. Cette méthode est une méthode de classe qui s'utilise sur le modèle lui-même (par exemple, `Client`).

Tableau 22–1 Méthode de classe find(conditions, callback)

Méthode	Signification
<code>ModelName.find(conditions, callback)</code>	Effectue une recherche de document dans la collection associée au modèle utilisé. Les conditions sont indiquées sous forme d'objet JSON et sont similaires à celles que nous avions écrites dans le chapitre 19, « Introduction à MongoDB » (elles sont explicitées à la suite), tandis que la fonction de callback (de la forme <code>function(err, docs)</code>) est appelée lorsque la recherche est terminée, avec succès (<code>err</code> vaut <code>null</code>) ou non. Le paramètre <code>docs</code> de la fonction de callback représente un tableau (éventuellement vide) des documents trouvés.

C'est la présence de la fonction de callback qui indique à Mongoose de déclencher la recherche dans la base de données. Si la fonction de callback n'est pas indiquée dans les paramètres, la recherche devra être déclenchée au moyen de la méthode `exec()` sur l'objet `mongoose.Query` retourné par la méthode `find()` (voir section suivante).

Exemples de recherche

Suite aux différentes insertions de documents effectuées dans les chapitres précédents, écrivons quelques fonctions de recherche dans la collection `clients`.

Rechercher tous les documents ayant un champ adresse contenant un champ ville dont la valeur est "Santa Monica"

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.find({ "adresse.ville" : "Santa Monica"}, function (err, clients) {
  if (err) return console.error(err);
  console.log(clients);
});
```

La condition est exprimée sous forme d'objet JSON, telle que nous l'avons écrite au chapitre 19.

Figure 22–1

Affichage de tous les clients situés à "Santa Monica"

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
[ { nom: 'Eastwood',
  prenom: 'Clint',
  _id: 531d94a5386699202ca38897,
  adresse:
    [ { _id: 531d94a5386699202ca38898,
      ville: 'Santa Monica',
      rue: 'rue de la mer' } ],
  __v: 0 },
  { nom: 'Eastwood2',
  prenom: 'Clint2',
  _id: 531da064ec79e4842dc3834f,
  adresse: { rue: 'rue de la mer', ville: 'Santa Monica' },
  __v: 0 } ]
```

On retrouve les deux enregistrements précédemment insérés. Modifions légèrement le schéma afin d'indiquer le champ `adresse` sous forme de `String` au lieu d'un sous-document.

Utiliser un champ `adresse` sous forme de `String`

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});

var Client = mongoose.model("Client", clientSchema);

Client.find({ "adresse.ville" : "Santa Monica"}, function (err, clients) {
  if (err) return console.error(err);
  console.log(clients);
});
```

Figure 22–2

Recherche de documents sans indiquer les sous-documents dans le schéma

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
[ { nom: 'Eastwood',
  prenom: 'Clint',
  _id: 531d94a5386699202ca38897,
  adresse: '[object Object]',
  __v: 0 },
  { nom: 'Eastwood2',
  prenom: 'Clint2',
  _id: 531da064ec79e4842dc3834f,
  adresse: '[object Object]',
  __v: 0 } ]
```

Les deux mêmes enregistrements sont trouvés, mais comme le schéma n'indique pas la décomposition du champ `adresse` sous forme d'un objet `{ rue, ville, cp }`, le détail du sous-document n'est pas affiché.

Tous les clients dont le champ `adresse` contient un champ `ville` existant

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.find({ "adresse.ville" : { $exists : true} }, function (err, clients) {
  if (err) return console.error(err);
  console.log(clients);
});
```

Figure 22–3

Afficher tous les clients ayant le champ `adresse` contenant le champ `ville`

```
ca Invité de commandes - node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
[ { _id: '52ee6159da4f2753442403de',
  nom: 'Dupont',
  prenom: 'Valentin',
  adresse: { rue: '17 rue Villeneuve', cp: '69000', ville: 'Lyon' },
  _id: '52ee6159da4f2753442403e0',
  nom: 'Duchemin',
  prenom: 'Valentin',
  adresse: { rue: '7 place du Tertre', cp: '75018', ville: 'Paris' },
  _id: '52ee7158b1e7e0cf70fd0b',
  nom: 'Martin',
  prenom: 'Michel',
  adresse:
    [ { ville: 'Lyon', cp: '69000', rue: '17 rue Villeneuve' },
      { ville: 'Paris', cp: '75012', rue: '3 bd Daumesnil' } ],
  _id: '52ee6159da4f2753442403df',
  nom: 'Durand',
  prenom: 'Georges',
  adresse: { rue: '7 av du centre', cp: '69000', ville: 'Lyon' },
  nom: 'Eastwood',
  prenom: 'Clint',
  _id: '531d94a5386699202ca38897',
  _v: 0,
  adresse:
    [ { _id: '531d94a5386699202ca38898',
        ville: 'Santa Monica',
        rue: 'rue de la mer' } ],
  nom: 'Eastwood2',
  prenom: 'Clint2',
  _id: '531da064ec79e4842dc3834f',
  _v: 0,
  adresse: { rue: 'rue de la mer', ville: 'Santa Monica' } } ]
```

Tous les clients dont le nom est "Sarrion" ou le prénom est "Js"

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.find({ $or : [{ nom : "Sarrion" } , { prenom : "Js" }] }, function (err, clients) {
  if (err) return console.error(err);
  console.log(clients);
});
```

Figure 22–4

Afficher tous les clients dont le nom est "Sarrion" ou le prénom est "Js".

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
[ { _id: '529331496cd10c46bedchbda',
  nom: 'Sarrion',
  prenom: 'Eric',
  adresse: {} },
  { _id: '529335e97b6b6ad0682be00a',
  nom: 'Node',
  prenom: 'Js',
  adresse: {} } ]
```

Écriture des conditions de recherche

Les exemples précédents ont montré que l'utilitaire `mongo` de MongoDB (utilisé au chapitre 19) permet d'exprimer les conditions de recherche sous la même forme que celle utilisée dans Mongoose. Afin de faciliter leur écriture, nous listons dans le tableau 22–2 quelques-unes des formes les plus utilisées dans l'expression des conditions.

Tableau 22–2 Quelques conditions de recherche

Condition	Signification
{ }	Tous les documents de la collection. On peut aussi écrire <code>find()</code> qui est équivalent à <code>find({})</code> .
{ nom : "Sarrion" }	Tous les documents dont le nom vaut "Sarrion".

Tableau 22–2 Quelques conditions de recherche (suite)

Condition	Signification
{ nom : "Sarrion", prenom : "Eric" }	Tous les documents dont le nom vaut "Sarrion" et le prénom vaut "Eric".
{ \$or : [{ nom : "Sarrion" } , { prenom : "Js" }] }	Tous les documents dont le nom vaut "Sarrion" ou le prénom vaut "Js".
{ nom : /obama/i }	Tous les documents dont le nom contient la chaîne "obama" sans tenir compte de la casse (expression régulière).
{ adresse : { \$exists : true} }	Tous les documents dont le champ <code>adresse</code> existe, quel que soit son type (<code>String</code> , <code>Object</code> , etc.).
{ adresse : { \$exists : true, \$type : 2 } }	Tous les documents dont le champ <code>adresse</code> existe, et qui soit du type <code>String</code> .
{ "adresse.ville" : "Santa Monica" }	Tous les documents contenant le champ <code>adresse</code> ayant lui-même un champ <code>ville</code> dont la valeur est "Santa Monica".
{ "adresse.0.ville" : "Santa Monica" }	Tous les documents contenant le champ <code>adresse</code> constitué en tableau, dont le premier élément contient le champ <code>ville</code> ayant la valeur "Santa Monica".
{ nom : { \$type : 2 }, \$where : "this.nom.match(/^sarrion node\$/i)" }	Tous les documents dont le nom est une chaîne de caractères (type = 2) et dont le nom commence par "sarrion" ou se termine par "node", quelle que soit la casse. Il faut indiquer que le nom est une chaîne de caractères sinon on peut avoir une erreur avec des noms qui ne sont pas sous cette forme.
{ nom : { \$gt : "J", \$lt : "S" } }	Tous les documents dont le nom est supérieur à "J" et inférieur à "S".
{ nom : { \$in : ["Sarrion", "Node", "Obama"] } }	Tous les documents dont les noms sont "Sarrion", "Node" ou "Obama".

Utiliser la méthode `find(conditions)`

La méthode `find()` peut également s'écrire sans fonction de callback en second paramètre. Le retour de l'appel de la méthode `find()` est dans ce cas un objet de classe `mongoose.Query`, sur lequel on peut appeler de nouvelles méthodes qui serviront à effectuer une recherche plus complexe.

Pour avoir une idée des méthodes utilisables sur la classe `mongoose.Query`, il suffit de les afficher dans la console du serveur.

Afficher les méthodes de la classe mongoose.Query

```
var mongoose = require("mongoose");  
  
console.log(mongoose.Query);
```

Figure 22–5

Affichage de toutes les méthodes de la classe mongoose.Query



```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js  
< [Function: Query]  
base:  
  toConstructor: [Function: toConstructor],  
  setOptions: [Function],  
  collection: [Function: collection],  
  '$where': [Function],  
  where: [Function],  
  equals: [Function: equals],  
  or: [Function: or],  
  nor: [Function: nor],  
  and: [Function: and],  
  gt: [Function],  
  gte: [Function],  
  lt: [Function],  
  lte: [Function],  
  ne: [Function],  
  in: [Function],  
  nin: [Function],  
  all: [Function],  
  regex: [Function],  
  size: [Function],  
  maxDistance: [Function],  
  mod: [Function],  
  exists: [Function],  
  elemMatch: [Function],  
  within: [Function: within],  
  box: [Function],  
  polygon: [Function],  
  circle: [Function],  
  near: [Function: near],  
  intersects: [Function: intersects],  
  geometry: [Function: geometry],  
  select: [Function: select],  
  slice: [Function],  
  sort: [Function],  
  limit: [Function],  
  skip: [Function],  
  maxScan: [Function],  
  batchSize: [Function],  
  comment: [Function],  
  snapshot: [Function],  
  hint: [Function],  
  slaveOk: [Function],  
  read: [Function],  
  tailable: [Function],  
  merge: [Function],  
  find: [Function],  
  findOne: [Function],  
  count: [Function],  
  distinct: [Function],  
  update: [Function: update],  
  remove: [Function],  
  findOneAndUpdate: [Function],  
  findOneAndRemove: [Function],  
  _findAndModify: [Function],  
  exec: [Function: exec],  
  selected: [Function: selected],  
  selectedInclusively: [Function: selectedInclusively],  
  selectedExclusively: [Function: selectedExclusively],  
  _mergeUpdate: [Function],  
  _optionsForExec: [Function],  
  _fieldsForExec: [Function],  
  _updateForExec: [Function],  
  _ensurePath: [Function],  
  _validate: [Function] },  
  useNewUrlParser: true }
```

Parmi ces méthodes, la méthode `exec(callback)` permet d'indiquer une fonction de callback qui sera appelée à la fin du traitement, comme si elle avait été indiquée directement en paramètre de la méthode `find(conditions, callback)`. Entre l'appel de la méthode `find()` et celui de la méthode `exec()`, d'autres méthodes peuvent être utilisées, qui sont celles de la liste affichée dans la figure 22–5. Toutes ces autres méthodes permettront de modifier les résultats de la recherche.

Utilisons la méthode `exec()` afin de traiter les résultats de la méthode `find()`.

Utiliser `find()`, puis `exec()`

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.find().exec(function(err, clients) {
  console.log(clients);
});
```

Figure 22–6

Afficher les clients par la méthode `exec()`

```
datemaj: Tue Feb 11 2014 11:05:01 GMT+0100 (Paris, Madrid),
nom: 'Jobs',
prenom: 'Steve',
adresse: <>,
< nom: 'Obama',
prenom: 'Barack',
_id: 531c9ac8d7d90bb41c6a1ce5,
_v: 0,
adresse: 'Washington' >,
< nom: 'Obama2',
prenom: 'Barack2',
_id: 531c9fb07bb1d6b8295f1417,
_v: 0,
adresse: 'Washington' >,
< nom: 'Obama3',
prenom: 'Barack3',
_id: 531ca2af807cdf442b5a117c,
_v: 0,
adresse: 'Washington' >,
< nom: 'Obama4',
prenom: 'Barack4',
_id: 531cab5c6d9c32742b9ef366,
_v: 0,
adresse: 'Washington' >,
< nom: 'Eastwood',
prenom: 'Clint',
_id: 531d94a5386699202ca38897,
_v: 0,
adresse:
[ < _id: 531d94a5386699202ca38898,
ville: 'Santa Monica',
rue: 'rue de la mer' > ] >,
< nom: 'Eastwood2',
prenom: 'Clint2',
_id: 531da064ec79e4842dc3834f,
_v: 0,
adresse: < rue: 'rue de la mer', ville: 'Santa Monica' > > ]
```

Nous pouvons altérer les résultats de la recherche en appelant de nouvelles méthodes entre les méthodes `find()` et `exec()`. Par exemple, conservons uniquement les documents dont le nom est supérieur à "N", triés dans l'ordre des noms croissants.

Trier les résultats par noms croissants, en ne conservant que ceux supérieurs à "N"

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.find().where("nom").gt("N").sort("nom").exec(function(err, clients) {
  console.log(clients);
});
```

Nous avons ici l'enchaînement des méthodes `find()`, `where()`, `gt()`, `sort()`, puis `exec()`, dans cet ordre. Pour plus de lisibilité, on peut aussi écrire le code précédent de la façon suivante.

Réécriture du code précédent de façon plus lisible

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);
```

```

Client.find()
  .where("nom").gt("N")
  .sort("nom")
  .exec(function(err, clients) {
    console.log(clients);
  });
  
```

Figure 22-7

Recherche de clients triés par noms croissants

```

C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
[ { _id: 529335e97b6b6ad0682be00a,
  nom: 'Node',
  prenom: 'Js',
  adresse: {} },
{ nom: 'Obama',
  prenom: 'Barack',
  _id: 531c9ac8d7d90bb41c6a1ce5,
  __v: 0,
  adresse: 'Washington' },
{ nom: 'Obama2',
  prenom: 'Barack2',
  _id: 531c9f007bb1d6b8295f1417,
  __v: 0,
  adresse: 'Washington' },
{ nom: 'Obama3',
  prenom: 'Barack3',
  _id: 531ca2af807cdf442b5a117c,
  __v: 0,
  adresse: 'Washington' },
{ nom: 'Obama4',
  prenom: 'Barack4',
  _id: 531cab5c6d9c32742b9ef366,
  __v: 0,
  adresse: 'Washington' },
{ _id: 52f65f98a4e1669bbb0695e,
  nom: 'Pays',
  prenom: 'Bas',
  adresse: {} },
{ _id: 529331496cd10c46bedcbda,
  nom: 'Saxion',
  prenom: 'Eric',
  adresse: {} } ]
  
```

On voit que les requêtes complexes peuvent s'écrire de façon plus simple grâce aux méthodes définies dans la classe `mongoose.Query`.

La méthode `where()` est particulière. Elle permet de choisir les champs du document sur lesquels des tests sont ensuite effectués. Plusieurs appels à `where()` peuvent être réalisés les uns après les autres, portant sur des champs différents ou non.

Plusieurs appels à where()

```

var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.find()                                // tous les clients
  .where("nom").gt("N").ne("Node")           // dont le nom est > "N" et != "Node"
  .where("prenom").nin(["Barack", "Barack2"]) // et dont le prénom est ni Barack
                                                // ni Barack2
  .sort("nom")                                // triés par noms croissants
  .exec(function(err, clients) {
    console.log(clients);
  });
}

```

Plusieurs méthodes peuvent s'enchaîner à la suite de la méthode `where()`. Par exemple, `gt("N")`, puis `ne("Node")` signifiant de conserver ce qui est supérieur à "N" et différent de "Node". On voit qu'on a ici un moyen très intuitif d'écrire les conditions de la recherche.

Figure 22–8

Recherche de clients triés par noms croissants

```

C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
[ { nom: 'Obama3',
  prenom: 'Barack3',
  _id: 531ca2af807cd442b5a117c,
  __v: 0,
  adresse: 'Washington' },
{ nom: 'Obama4',
  prenom: 'Barack4',
  _id: 531cab5c6d9c32742b9ef366,
  __v: 0,
  adresse: 'Washington' },
{ _id: 52f65f98a4e1669bb0b0695e,
  nom: 'Pays',
  prenom: 'Bas',
  adresse: {} },
{ _id: 529331496cd10c46bedchbda,
  nom: 'Sarrion',
  prenom: 'Eric',
  adresse: {} } ]

```

Méthodes utilisables dans la classe mongoose.Query

Dans la section précédente, nous avons listé les méthodes définies sur la classe `mongoose.Query`, et nous avons vu l'utilisation de certaines d'entre elles. Les méthodes de cette classe peuvent être chaînées les unes avec les autres jusqu'à ce que la recherche soit déclenchée au moyen de la méthode `exec()`. Nous donnons dans le tableau 22-3 une liste des principales méthodes utilisables.

Elles sont classées en deux catégories :

- celles qui s'utilisent après une méthode `where(attribut)`, laquelle aura spécifié l'attribut sur lequel on effectue des comparaisons ;
- celles qui sont indépendantes de l'utilisation d'une clause `where()` et ne sont donc pas rattachées à un attribut en particulier.

Le tableau 22-3 présente les méthodes qui s'utilisent indépendamment de la méthode `where(attribut)`.

Tableau 22-3 Méthodes de la classe `mongoose.Query` indépendantes de la méthode `where(attribut)`

Méthode	Signification
<code>exec(callback)</code>	S'utilise à la fin de l'appel des méthodes pour déclencher la recherche dans la base de données. La fonction de callback (optionnelle) est de la forme <code>function (err, docs)</code> dans laquelle <code>docs</code> est le tableau des documents retournés par la recherche. Cette méthode permet, de façon plus générale, de déclencher un objet <code>mongoose.Query</code> (retourné par la méthode <code>find()</code> ou similaire).
<code>sort(attribut)</code>	Trie les résultats de la recherche dans l'ordre des attributs indiqués. Par exemple : - <code>sort("nom")</code> trie de façon ascendante sur le nom ; - <code>sort("-nom")</code> trie de façon descendante sur le nom ; - <code>sort("nom prenom")</code> trie de façon ascendante sur le nom, puis sur le prénom.
<code>select(attribut)</code>	Indique les attributs que l'on désire récupérer. Le champ <code>_id</code> est toujours récupéré, sauf s'il est explicitement exclu. Par exemple : - <code>select("nom")</code> sélectionne l'attribut <code>nom</code> et <code>_id</code> ; - <code>select("nom prenom")</code> sélectionne l'attribut <code>nom</code> , <code>prenom</code> et <code>_id</code> ; - <code>select("nom prenom _id")</code> sélectionne l'attribut <code>nom</code> , <code>prenom</code> (en excluant <code>_id</code>) ; - <code>select("-_id")</code> sélectionne tous les attributs sauf <code>_id</code> .
<code>\$where(jsexpr)</code>	Exécute l'expression booléenne JavaScript pour chaque document retourné par la recherche en ne conservant que les documents qui satisfont l'expression. Le mot-clé <code>this</code> référence chacun des documents. Par exemple : <code>\$where("this.nom == 'Node'")</code> indique de ne conserver que les documents dont le nom est "Node".

Le tableau 22-4 liste quant à lui les méthodes qui sont précédées de la méthode `where(attribut)`. Chacune de ces méthodes sert à récupérer un sous-ensemble de documents qui satisfont la condition.

Tableau 22-4 Méthodes de la classe `mongoose.Query` utilisées avec la méthode `where(attribut)`

Méthode	Signification
<code>equals(value)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a la valeur indiquée en paramètre.
<code>gt(value)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a une valeur supérieure à celle indiquée en paramètre.
<code>gte(value)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a une valeur supérieure ou égale à celle indiquée en paramètre.
<code>lt(value)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a une valeur inférieure à celle indiquée en paramètre.
<code>lte(value)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a une valeur inférieure ou égale à celle indiquée en paramètre.
<code>ne(value)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a une valeur différente de celle indiquée en paramètre.
<code>in(array)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a une valeur correspondant à l'une de celles indiquées dans le tableau en paramètre.
<code>nin(array)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a une valeur différente de celles indiquées dans le tableau en paramètre.
<code>regex(rerexp)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> a une valeur qui correspond à l'expression régulière indiquée en paramètre.
<code>exists(boolean)</code>	Teste si le champ indiqué dans la méthode <code>where()</code> existe dans le document (si <code>boolean</code> vaut <code>true</code> , valeur par défaut) ou non (si <code>boolean</code> vaut <code>false</code>).

Utiliser la méthode `findOne()`

La méthode `findOne()` est similaire à la méthode `find()` vue précédemment, sauf qu'elle recherche un seul document au lieu de plusieurs comme la méthode `find()`. Le résultat transmis à la méthode `exec(err, result)` sera donc un seul document (ou `null` si aucun résultat) au lieu d'un tableau (qui pouvait être vide si pas de résultat).

La méthode `findOne()` peut s'écrire, comme la méthode `find()`, avec la fonction de callback insérée en paramètre de la méthode, ou à l'extérieur à l'aide de la méthode `exec()`.

Tableau 22–5 Formes de la méthode findOne()

Méthode	Signification
<code>ModelName.findOne(conditions, callback)</code>	Recherche le premier document qui correspond aux conditions de recherche et appelle la fonction de callback de la forme <code>function(err, doc)</code> dans laquelle <code>doc</code> est le document trouvé ou <code>null</code> .
<code>ModelName.findOne(conditions)</code>	Même principe que la méthode précédente, mais la fonction de callback sera ajoutée à l'aide de la méthode <code>exec(callback)</code> .

Utiliser la méthode findById()

Lorsque la condition porte sur l'identifiant (champ `_id`), Mongoose permet d'utiliser la méthode `findById()` utilisable également sur le modèle.

La méthode `findById()` est similaire à la méthode `find()` vue précédemment, sauf qu'elle recherche un seul document au lieu de plusieurs comme la méthode `find()`. Le résultat transmis à la méthode `exec(err, result)` sera donc un seul document (ou `null` si aucun résultat) au lieu d'un tableau (qui pouvait être vide si pas de résultat).

La méthode `findById()` peut s'écrire, comme la méthode `find()`, avec la fonction de callback insérée en paramètre de la méthode, ou à l'extérieur à l'aide de la méthode `exec()`.

Tableau 22–6 Formes de la méthode findById()

Méthode	Signification
<code>ModelName.findById(conditions, callback)</code>	Recherche le premier document qui correspond à cet identifiant et appelle la fonction de callback de la forme <code>function(err, doc)</code> dans laquelle <code>doc</code> est le document trouvé ou <code>null</code> .
<code>ModelName.findById(conditions)</code>	Même principe que la méthode précédente, mais la fonction de callback est ajoutée à l'aide de la méthode <code>exec(callback)</code> .

Utiliser la méthode count()

La méthode `count()` s'utilise sur le modèle (par exemple, `Client`) ou sur l'objet `mongoose.Query` (retourné par la méthode `find()`). Elle permet de compter les documents qui satisfont les conditions de recherche. Elle s'utilise sous l'une des deux formes listées dans le tableau 22–7.

Tableau 22–7 Formes de la méthode findOne()

Méthode	Signification
count(conditions, callback)	Compte le nombre de documents qui satisfont les conditions indiquées en paramètres. La fonction de callback est de la forme <code>function (err, count)</code> dans laquelle <code>count</code> est le nombre de documents trouvés.
count(callback)	Compte le nombre de documents qui satisfont les conditions exprimées dans les méthodes précédemment utilisées. La fonction de callback est de la forme <code>function (err, count)</code> dans laquelle <code>count</code> est le nombre de documents trouvés.

Voici un exemple d'utilisation de chacune de ces deux méthodes.

Utiliser count(conditions, callback)

Dans l'utilisation de cette forme de la méthode, le nombre de documents à compter provient des conditions exprimées en paramètres.

Compter le nombre de documents dont le nom contient "Obama"

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.count({ nom : /Obama/ }, function(err, count) {
  console.log("Nombre de clients dont le nom contient Obama : " + count);
});
```

La condition est exprimée en premier paramètre de la méthode `count()`, tandis que la fonction de callback est appelée pour indiquer le résultat.

Figure 22–9

Affichage du nombre de clients dont le nom contient "Obama".

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Nombre de clients dont le nom contient Obama : 4
```

Si une recherche est effectuée avant l'appel de la méthode `count()`, les conditions de cette recherche sont ignorées par la méthode `count()` qui privilégie celles indiquées en paramètres de la méthode.

Effectuons le même comptage que le précédent, mais en appelant la méthode `find()` avant la méthode `count()`. Nous verrons que seules les conditions exprimées dans `count()` sont prises en compte.

Utiliser `find()` avant `count()`

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.find({ nom : "Sarrion" }).count({ nom : /Obama/ }, function(err, count) {
  console.log("Nombre de clients dont le nom contient Obama : " + count);
});
```

Figure 22–10

Utilisation de `find()`, puis `count()` : seules les conditions exprimées dans `count()` sont prises en compte.

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Nombre de clients dont le nom contient Obama : 4
```

Utiliser count(callback)

Dans le cas où la méthode `count()` est appelée sans indiquer les conditions, ce sont celles exprimées dans la méthode `find()` utilisée précédemment qui sont prises en compte. Le même comptage que celui que nous venons de réaliser peut alors s'écrire :

Compter le nombre de documents dont le nom contient "Obama"

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

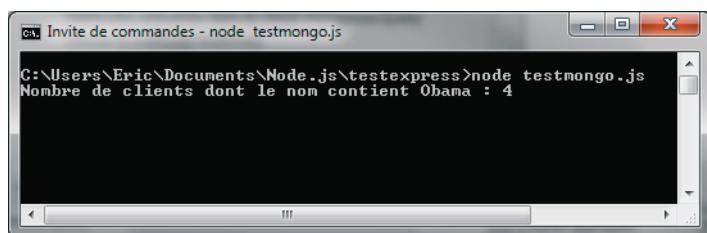
var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});

var Client = mongoose.model("Client", clientSchema);

Client.find({ nom : /Obama/ }).count(function(err, count) {
  console.log("Nombre de clients dont le nom contient Obama : " + count);
});
```

Figure 22-11

Utilisation de `find()`,
puis `count()`



En fait, la méthode `count()` tient compte de l'objet `mongoose.Query` sur lequel elle est appelée. On pourrait donc aussi écrire, de manière plus complexe :

Utiliser where() et regex() avant count()

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
```

```
adresse : {
    rue : String,
    ville : String,
    cp : String
}
});

var Client = mongoose.model("Client", clientSchema);

Client.find().where("nom").regex(/Obama/).count(function(err, count) {
    console.log("Nombre de clients dont le nom contient Obama : " + count);
});
```

23

Modifier des documents

Après avoir vu comment créer et rechercher des documents, voyons maintenant comment modifier les documents déjà présents dans la base de données. Plusieurs manières sont possibles.

- En utilisant la méthode de classe `update()` définie sur le modèle.
- En utilisant la méthode d'instance `save()`, qui permet de sauvegarder un document modifié dans la collection.
- En utilisant les méthodes de classe `findOneAndUpdate()` et `findByIdAndUpdate()` définies sur le modèle, qui permettent de mettre à jour un seul document (ce que fait par défaut la méthode `update()`). La différence avec la méthode `update()` est que l'enregistrement mis à jour est disponible en tant que paramètre dans la fonction de callback, alors qu'il n'est pas fourni avec la méthode `update()`.

Les différentes formes de ces méthodes sont étudiées dans les sections suivantes.

Utiliser la méthode de classe `update()`

La méthode `update(conditions, update, options, callback)` est une méthode de classe, elle s'utilise donc sur le modèle (par exemple, `Client`). Elle permet de mettre à jour un (valeur par défaut) ou plusieurs documents de la collection, qui satisfont les conditions de recherche indiquées. La fonction de callback est appelée à la fin de la mise à jour et indique le nombre de documents mis à jour (0 si aucun ne correspond aux conditions).

Tableau 23–1 Méthode de classe update(conditions, update, options, callback)

Méthode	Signification
<code>ModelName.update (conditions, update, options, callback)</code>	<p>Effectue une mise à jour d'un ou plusieurs documents de la collection associée au modèle.</p> <p>Le paramètre <code>conditions</code> correspond aux conditions de recherche des documents à mettre à jour.</p> <p>Le paramètre <code>update</code> indique les champs à mettre à jour dans les documents, avec leurs valeurs. Attention : seuls les champs indiqués dans le schéma seront mis à jour. Les autres champs seront ignorés.</p> <p>Le paramètre <code>options</code> (facultatif) permet d'indiquer des options :</p> <ul style="list-style-type: none"> - <code>options.upsert</code> : si <code>true</code>, indique de créer un nouveau document si aucun document ne peut être mis à jour car aucun document ne correspond aux conditions de recherche. Par défaut, cette option vaut <code>false</code> (pas de création de nouveau document si aucun ne correspond). - <code>options.multi</code> : si <code>true</code>, indique de mettre à jour tous les documents qui satisfont la recherche, sinon mise à jour du premier document trouvé uniquement. Par défaut, cette option vaut <code>false</code> (mise à jour du premier document trouvé uniquement). <p>Le paramètre <code>callback</code> est une fonction de callback appelée lorsque la mise à jour a été effectuée (avec ou sans erreur). Elle est de la forme <code>function(err, numberAffected)</code> dans laquelle <code>numberAffected</code> est le nombre de documents mis à jour.</p>

Par défaut, un seul document satisfaisant les conditions de recherche est mis à jour : le premier qui est trouvé. Il est possible de modifier cela à l'aide de l'option `multi` positionnée à `true` (valeur `false` par défaut). Les sections suivantes sont consacrées à la mise à jour d'un ou de plusieurs documents.

Mise à jour d'un seul document correspondant aux critères de recherche

On souhaite modifier un seul document correspondant à notre critère de recherche. Le fonctionnement par défaut de la méthode `update()` convient.

Ajouter l'âge dans le premier document contenant le nom Obama

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  }
});
```

```

    },
    age : Number
});

var Client = mongoose.model("Client", clientSchema);

Client.update({ nom : /Obama/ }, { age : 52 }, function(err, numberAffected) {
  console.log("Nombre de documents mis à jour : " + numberAffected);
  Client.find({nom : /Obama/}, function(err, clients) {
    console.log(clients);
  });
});

```

On recherche tous les clients dont le nom contient "Obama", tout en ne conservant que le premier (car l'option `multi` est positionnée à `false` par défaut), puis on positionne le champ `age` à la valeur 52. Il faut pour cela que le schéma contienne ce champ (ce qui n'était pas le cas auparavant, d'où l'ajout du champ `age` dans le schéma). La fonction de callback indique le nombre de documents affectés, puis on affiche dans celle-ci la nouvelle liste des clients qui contiennent le nom "Obama".

Figure 23-1
Ajout du champ "age"
dans un document

```

C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>Nombre de documents mis à jour : 1
[ {
  _v: 0,
  _id: 531c9ac8d7d90bb41c6aice5,
  age: 52,
  nom: 'Obama',
  prenom: 'Barack',
  adresse: 'Washington' },
  {
  _v: 0,
  _id: 531c9fb07bb1d6b8295f1417,
  nom: 'Obama2',
  prenom: 'Barack2',
  adresse: 'Washington' },
  {
  _v: 0,
  _id: 531ca2af807cdf442b5a117c,
  nom: 'Obama3',
  prenom: 'Barack3',
  adresse: 'Washington' },
  {
  _v: 0,
  _id: 531cab5c6d9c32742b9ef366,
  nom: 'Obama4',
  prenom: 'Barack4',
  adresse: 'Washington' } ]

```

Seul le client dont le nom est "Obama" (le premier trouvé dans la collection) est modifié.

Mise à jour de plusieurs documents correspondant aux critères de recherche

Pour mettre à jour plusieurs documents dans la collection, vous aurez besoin de l'option `multi` qui doit être positionnée à `true`. Utilisons cette option pour mettre à jour tous les documents contenant le nom "Obama".

Ajouter le champ age à tous les documents contenant le nom "Obama"

```

var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

Client.update({nom : /Obama/}, { age : 52 }, { multi : true }, function(err,
numberAffected) {
  console.log("Nombre de documents mis à jour : " + numberAffected);
  Client.find({nom : /Obama/}, function(err, clients) {
    console.log(clients);
  });
});

```

Figure 23-2

Ajout du champ "age" à tous les documents contenant le nom "Obama"

```

C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Nombre de documents mis à jour : 4
[ { _v: 0,
  _id: 531c9ac8d7d90bb41c6a1ce5,
  age: 52,
  nom: 'Obama',
  prenom: 'Barack',
  adresse: 'Washington' },
  { _v: 0,
  _id: 531c9fb07bb1d6b8295f1417,
  age: 52,
  nom: 'Obama2',
  prenom: 'Barack2',
  adresse: 'Washington' },
  { _v: 0,
  _id: 531ca2af807cdf442b5a117c,
  age: 52,
  nom: 'Obama3',
  prenom: 'Barack3',
  adresse: 'Washington' },
  { _v: 0,
  _id: 531cab5c6d9c32742b9ef366,
  age: 52,
  nom: 'Obama4',
  prenom: 'Barack4',
  adresse: 'Washington' } ]

```

Les documents mis à jour sont au nombre de quatre, incluant le précédent qui était déjà à jour.

Suppression de champs dans les documents

La clé `$unset` dans le paramètre `update` permet de supprimer les champs indiqués dans un document. Par exemple, supprimons le champ `age` précédemment ajouté.

Supprimer le champ `age` dans tous les documents contenant le nom "Obama"

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

Client.update({nom : /Obama/}, { $unset : { age : 0 } }, { multi : true },
function(err, numberAffected) {
  console.log("Nombre de documents mis à jour : " + numberAffected);
  Client.find({nom : /Obama/}, function(err, clients) {
    console.log(clients);
  });
});
```

Le champ `age` est ici positionné à 0 bien que sa valeur soit non significative, du fait que le champ est supprimé des documents concernés.

Figure 23–3

Le champ "age"
est maintenant supprimé
de tous les documents.

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Nombre de documents mis à jour : 4
[ { _v: 0,
  _id: '531c9ac8d7d90bb41c6a1ce5',
  nom: 'Obama',
  prenom: 'Barack',
  adresse: 'Washington' },
{ _v: 0,
  _id: '531c9fb07bb1d6b8295f1417',
  nom: 'Obama2',
  prenom: 'Barack2',
  adresse: 'Washington' },
{ _v: 0,
  _id: '531ca2af807cdf442b5a117c',
  nom: 'Obama3',
  prenom: 'Barack3',
  adresse: 'Washington' },
{ _v: 0,
  _id: '531cab5c6d9c32742b9ef366',
  nom: 'Obama4',
  prenom: 'Barack4',
  adresse: 'Washington' } ]
```

Le champ `age` a été supprimé de tous les documents contenant le nom "Obama".

Utiliser la méthode `save()`

La méthode `save()` que nous avions déjà utilisée pour créer de nouveaux documents peut également être utilisée pour sauvegarder un document modifié. La méthode `save()` permet de modifier un seul document à la fois, contrairement à la méthode `update()` qui permet de modifier tous les documents qui satisfont la condition (en positionnant l'option `multi` à `true`).

Tableau 23–2 Méthode d'instance `save(callback)`

Méthode	Signification
<code>doc.save(callback)</code>	Modifie le document dans la collection associée au modèle. La fonction de callback est appelée à la fin de la sauvegarde du document, quel que soit le résultat. Elle est de la forme <code>function(err, doc)</code> dans laquelle <code>err</code> contient l'erreur ou <code>null</code> si pas d'erreur, tandis que <code>doc</code> est le document modifié.

Utilisons la méthode `save()` pour modifier le nom "Obama" en "Obama1" et le prénom "Barack" en "Barack1". Pour cela, il faut tout d'abord récupérer le document en mémoire, puis le modifier et enfin le sauvegarder.

Modifier le nom et le prénom de Barack Obama

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

Client.findOne({ nom : "Obama" }, function(err, c) {
  if (err) return;
```

```

if (!c) {
  console.log("Aucun client de nom \"Obama\"");
  return;
}
c.nom = "Obamal";
c.prenom = "Barack1";
c.save(function(err, client) {
  console.log(client + " modifié !");
  console.log("\nListe des clients contenant le nom \"Obama\"");
  Client.find({ nom : /Obama/ }, function(err, clients) {
    console.log(clients);
  });
});
}
);

```

Tous les traitements sont effectués dans les fonctions de callback de chacune des méthodes. La fonction de callback associée à la méthode `save()` permet de savoir que le document a été sauvegardé. On peut alors afficher la liste des clients de la collection.

Figure 23–4

Le document "Barack Obama" est devenu "Barack1 Obama1".

```

c:\ Invité de commandes - node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
{
  _v: 0,
  _id: 531c9ac8d7d90bb41c6a1ce5,
  nom: 'Obamal',
  prenom: 'Barack1',
  adresse: 'Washington' } modifié !
Liste des clients contenant le nom "Obama"
[ { _v: 0,
  _id: 531c9ac8d7d90bb41c6a1ce5,
  nom: 'Obamal',
  prenom: 'Barack1',
  adresse: 'Washington' },
{ _v: 0,
  _id: 531c9fb07bb1d6b8295f1417,
  nom: 'Obama2',
  prenom: 'Barack2',
  adresse: 'Washington' },
{ _v: 0,
  _id: 531ca2af807cdf442b5a117c,
  nom: 'Obama3',
  prenom: 'Barack3',
  adresse: 'Washington' },
{ _v: 0,
  _id: 531cab5c6d9c32742b9ef366,
  nom: 'Obama4',
  prenom: 'Barack4',
  adresse: 'Washington' } ]

```

L'avantage de la méthode `save()` est qu'elle donne accès au document modifié (dans la fonction de callback), contrairement à la méthode `update()`.

Utiliser la méthode findOneAndUpdate()

L'exemple précédent montre que pour modifier un document au moyen de la méthode `save()`, il faut d'abord le rechercher au moyen de la méthode `findOne()`, puis le modifier dans la fonction de callback associée. Pour accéder au document modifié, on doit donc utiliser deux fonctions de callback comme le montre le précédent exemple.

La méthode de classe `findOneAndUpdate(conditions, update, options, callback)` permet de réduire le nombre de fonctions de callback à utiliser pour arriver au même résultat. On utilisera alors une seule fonction de callback (celle indiquée en paramètres de la méthode).

Tableau 23–3 Méthode de classe `findOneAndUpdate(conditions, update, options, callback)`

Méthode	Signification
<code>ModelName.findOneAndUpdate(conditions, update, options, callback)</code>	<p>Effectue une mise à jour d'un document de la collection associée au modèle.</p> <p>Le paramètre <code>conditions</code> correspond aux conditions de recherche du document à mettre à jour.</p> <p>Le paramètre <code>update</code> indique les champs à mettre à jour dans le document, avec leurs valeurs. Attention : seuls les champs indiqués dans le schéma seront mis à jour. Les autres champs sont ignorés.</p> <p>Le paramètre <code>options</code> (facultatif) permet d'indiquer des options :</p> <ul style="list-style-type: none"> - <code>options.upsert</code> : si <code>true</code>, indique de créer un nouveau document si aucun document ne peut être mis à jour car aucun document ne correspond aux conditions de recherche. Par défaut, cette option vaut <code>false</code> (pas de création de nouveau document si aucun ne correspond). - <code>options.sort</code> : si le paramètre <code>conditions</code> retourne plusieurs documents, l'option <code>sort</code> permet de les trier afin d'en sélectionner un seul (sinon un des documents est choisi aléatoirement). Par exemple, pour trier selon les noms croissants, on indiquera <code>{ sort : { nom : 1 } }</code>. Spécifiez <code>-1</code> pour trier dans l'ordre décroissant. <p>Le paramètre <code>callback</code> est une fonction de callback appelée lorsque la mise à jour a été effectuée (avec ou sans erreur). Elle est de la forme <code>function(err, doc)</code> dans laquelle <code>doc</code> est le document mis à jour.</p>

Prenons le même exemple que le précédent, en remplaçant "`Barack1 Obama1`" par "`Barack Obama`".

Utiliser la méthode `findOneAndUpdate()`

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

Client.findOneAndUpdate({ nom : /Obama/ }, { nom : "Obama", prenom : "Barack" },
{ sort : { nom : 1 } }, function(err, client) {
  if (err) return;
  if (!client) {
    console.log("Aucun client contenant le nom \"Obama\"");
    return;
  }
  console.log(client + " modifié !");
  console.log("\nListe des clients contenant le nom \"Obama\"");
  Client.find({ nom : /Obama/ }, function(err, clients) {
    console.log(clients);
  });
});
```

La condition `{ nom : /Obama/ }` concerne quatre documents, le premier étant "Barack1 Obama1" dans l'ordre alphabétique. Pour être certain de modifier celui-ci, on trie les noms par ordre alphabétique croissant (grâce à l'option `sort`).

Utiliser la méthode `findByIdAndUpdate()`

Plutôt que spécifier une condition comme dans la méthode `findOneAndUpdate()`, on peut indiquer un identifiant (attribut `_id`). Celui-ci étant unique dans la collection, un seul document sera traité. Hormis le premier paramètre, cette méthode est similaire à la précédente.

Tableau 23–4 Méthode de classe findByIdAndUpdate(id, update, options, callback)

Méthode	Signification
<code>ModelName.findByIdAndUpdate (id, update, options, callback)</code>	<p>Effectue une mise à jour d'un document de la collection associée au modèle.</p> <p>Le paramètre <code>id</code> correspond à l'identifiant du document à mettre à jour.</p> <p>Le paramètre <code>update</code> indique les champs à mettre à jour dans le document, avec leurs valeurs. Attention : seuls les champs indiqués dans le schéma seront mis à jour. Les autres champs sont ignorés.</p> <p>Le paramètre <code>options</code> (facultatif) permet d'indiquer des options :</p> <ul style="list-style-type: none">- <code>options.upsert</code> : si <code>true</code>, indique de créer un nouveau document si aucun document ne peut être mis à jour car aucun document ne correspond aux conditions de recherche. Par défaut, cette option vaut <code>false</code> (pas de création de nouveau document si aucun ne correspond). <p>Le paramètre <code>callback</code> est une fonction de callback appelée lorsque la mise à jour a été effectuée (avec ou sans erreur). Elle est de la forme <code>function(err, doc)</code> dans laquelle <code>doc</code> est le document mis à jour.</p>

24

Supprimer des documents

Dans les chapitres précédents, nous avons vu comment créer, lire et mettre à jour des documents. Voyons maintenant comment les supprimer, ce qui permettra d'effectuer les quatre opérations fondamentales sur les données.

Mongoose permet de supprimer des documents au moyen de la méthode `remove()`, qui peut être utilisée en tant que méthode de classe ou méthode d'instance. Il est également possible d'utiliser les méthodes `findOneAndRemove()` et `findByIdAndRemove()` sur le modèle.

Utiliser la méthode de classe `remove()`

La méthode de classe `remove(conditions, callback)` permet de supprimer les documents de la collection qui correspondent aux conditions indiquées.

Tableau 24–1 Méthode de classe `remove(conditions, callback)`

Méthode	Signification
<code>ModelName.remove(conditions, callback)</code>	Supprime les documents associés aux conditions exprimées. Le paramètre <code>callback</code> est une fonction de callback de la forme <code>function(err, countRemoved)</code> appelée lorsque la suppression a été effectuée, dans laquelle le paramètre <code>countRemoved</code> indique le nombre de documents supprimés.

Utilisons la méthode `remove()` afin de supprimer tous les clients contenant le nom "Obama".

Supprimer tous les clients dont le nom contient "Obama"

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

Client.remove({ nom : /Obama/ }, function(err, countRemoved) {
  console.log("Nombre de clients supprimés : " + countRemoved);
  console.log("\nListe des clients contenant le nom \"Obama\"");
  Client.find({ nom : /Obama/ }, function(err, clients) {
    console.log(clients);
  });
});
```

Figure 24-1

Les clients ayant le nom contenant "Obama" ont été supprimés.

```
C:\> Invite de commandes - node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Nombre de clients supprimés : 4
Liste des clients contenant le nom "Obama"
[]
```

Les quatre clients contenant "Obama" ont été supprimés.

Utiliser la méthode d'instance remove()

La méthode `remove()` peut également être utilisée en tant que méthode d'instance sur un document particulier. Elle supprime ce document uniquement.

Tableau 24–2 Méthode d'instance remove(callback)

Méthode	Signification
<code>doc.remove(callback)</code>	Modifie le document dans la collection associée au modèle. La fonction de callback est appelée à la fin de la suppression du document, quel que soit le résultat. Elle est de la forme <code>function(err, doc)</code> dans laquelle <code>err</code> contient l'erreur ou <code>null</code> si pas d'erreur.

Utilisons la méthode `doc.remove()` pour supprimer le client ayant le nom "Node". On le récupère en mémoire au moyen de la méthode `findOne()`, puis on le supprime au moyen de `remove()`.

Supprimer le client ayant le nom "Node"

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

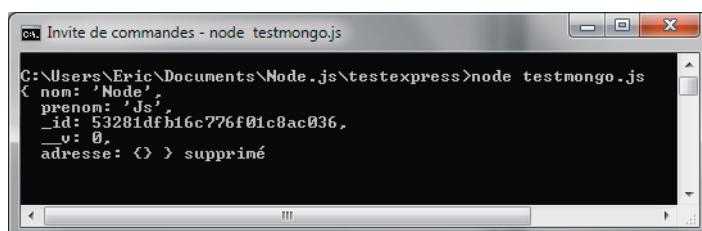
var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

Client.findOne({ nom : "Node" }, function (err, client) {
  if (!client) return;
  client.remove(function(err, c) {
    if (c) console.log(c + " supprimé");
  });
});
```

Figure 24–2

Le client dont le nom est "Node" a été supprimé.



Utiliser la méthode findOneAndRemove()

Plutôt que de supprimer un document en deux étapes et d'utiliser deux fonctions de callback comme précédemment, on peut le supprimer en une seule opération. C'est le rôle de la méthode de classe `findOneAndRemove(conditions, options, callback)`.

Tableau 24–3 Méthode de classe `findOneAndRemove(conditions, options, callback)`

Méthode	Signification
<code>ModelName.findOneAndRemove(conditions, options, callback)</code>	Supprime un document de la collection associée au modèle. Le paramètre <code>conditions</code> correspond aux conditions de recherche du document à supprimer. Le paramètre <code>options</code> (facultatif) permet d'indiquer des options : - <code>options.sort</code> : si le paramètre <code>conditions</code> retourne plusieurs documents, l'option <code>sort</code> permet de les trier afin d'en sélectionner un seul (sinon un des documents est choisi aléatoirement). Par exemple, pour trier selon les noms croissants on indiquera <code>{ sort : { nom : 1 } }</code> . Spécifier <code>-1</code> pour trier dans l'ordre décroissant. Le paramètre <code>callback</code> est une fonction de callback appelée lorsque la suppression a été effectuée (avec ou sans erreur). Elle est de la forme <code>function(err, doc)</code> dans laquelle <code>doc</code> est le document supprimé.

Utiliser la méthode `findOneAndRemove(conditions, options, callback)`

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);
```

```
Client.findOneAndRemove({ nom : "Sarrion" }, function (err, client) {
  console.log(client + " supprimé");
  console.log("Liste des clients ayant le nom \"Sarrion\"");
  Client.find({ nom : "Sarrion" }, function(err, clients) {
    console.log(clients);
  });
});
```

Figure 24–3

Le client dont le nom est "Sarrion" a été supprimé.

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
{
  _id: 529331496cd10c46bedchbda,
  nom: 'Sarrion',
  prenom: 'Eric',
  adresse: 'supprimé'
}
Liste des clients ayant le nom "Sarrion"
[]
```

Utiliser la méthode `findByIdAndRemove()`

On peut aussi utiliser la méthode `findByIdAndRemove(id, callback)` qui s'utilise sur le modèle. En précisant un identifiant (attribut `_id`) au lieu du paramètre `conditions`, on supprime le document qui possède cet identifiant.

Tableau 24–4 Méthode de classe `findByIdAndRemove(id, options, callback)`

Méthode	Signification
<code>ModelName.findByIdAndRemove(id, callback)</code>	Supprime le document ayant cet identifiant de la collection associée au modèle. Le paramètre <code>callback</code> est une fonction de callback appelée lorsque la suppression a été effectuée (avec ou sans erreur). Elle est de la forme <code>function(err, doc)</code> dans laquelle <code>doc</code> est le document supprimé.

25

Valider les données

La validation des données consiste à filtrer les documents insérés dans la base de données, en autorisant ou non l'opération d'insertion ou de modification.

Par défaut, aucune validation n'est effectuée par le système, ce qui signifie que toute opération de création ou de mise à jour dans la base de données est autorisée. Pour que le principe de validation soit actif, il faut utiliser uniquement les méthodes `save()` ou `create()` suivantes :

- la méthode `save()` permet de sauvegarder un document (déjà existant ou non) ;
- la méthode `create()` permet de créer un nouveau document.

Les autres méthodes, par exemple `findOneAndUpdate()`, ne déclenchent pas le processus de validation et donc ne tiennent pas compte des validations mises en place.

Préparation de la base de données

Afin de partir sur des bases saines, on supprime tous les documents de la collection `clients`, à l'aide du programme suivant. De nouveaux documents seront créés dans ce chapitre, permettant d'illustrer le processus de validation.

Supprimer le contenu de la collection clients

```
var mongoose = require("mongoose");  
  
var db = mongoose.connect("mongodb://localhost/mydb");
```

```

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

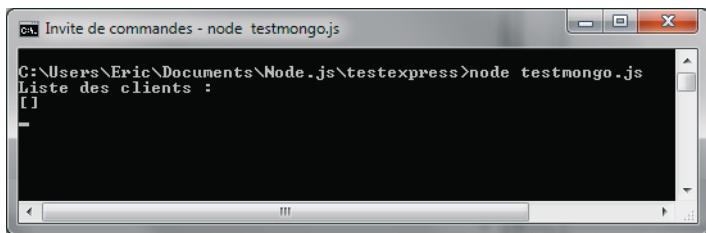
var Client = mongoose.model("Client", clientSchema);

Client.remove(function(){
  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});

```

Figure 25-1

Tous les documents de la collection clients ont été supprimés.



La collection `clients` est maintenant vide.

Valider un premier document

La mise en place de contraintes de validation s'effectue dans le code du schéma. Des clés spécifiques sont utilisées pour indiquer ces contraintes, telle que la clé `required` pour spécifier qu'un champ est obligatoire (par exemple, pour indiquer que le nom d'un client est obligatoire).

Indiquer que le nom du client est obligatoire

```

var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

```

```

var clientSchema = mongoose.Schema({
  nom : { type : String, required : true },
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "", prenom : "Eric" }); // nom vide

c.save(function(err) {
  console.log(err);
  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});

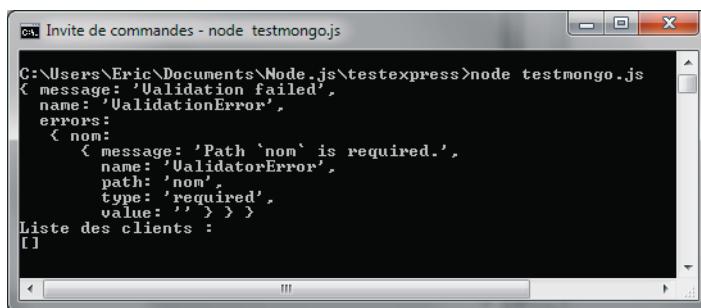
```

Dans le schéma, nous avons indiqué que le champ `nom` est de type `String`, mais également qu'il est obligatoire au moyen de la clé `required` positionnée à `true`. D'autres critères de validation existent, et on peut également en créer de nouveaux (voir la suite de ce chapitre).

Dans ce code, nous essayons de déclencher le mécanisme de validation en insérant un nom vide pour un client. Si le mécanisme de validation fonctionne, le client ne doit pas être inséré dans la collection.

Figure 25–2

La validation du client a échoué, il n'est pas inséré dans la base.



L'objet `err` est affiché, suivi de la liste des clients qui est effectivement vide, car le client n'a pas été inséré. On peut faire le test en indiquant un nom (par exemple, "Sarrion" au lieu de ""). Le client est alors inséré dans la base de données.

Afficher un message d'erreur si la validation échoue

L'exemple précédent montre que l'on peut mettre en place un mécanisme de validation. L'objet `err` contient un champ `message` qui indique ici "Path 'nom' is required". On souhaiterait afficher un texte plus francisé tel que "Le nom est obligatoire".

Afficher un message d'erreur personnalisé

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : { type : String, required : "Le nom est obligatoire" },
  prenom : String,
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "", prenom : "Eric" });

c.save(function(err) {
  console.log(err);
  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});
```

Le message d'erreur est indiqué dans la valeur de la clé `required`, à la place de la valeur `true` précédente.

Figure 25-3

Le champ "message" contient notre message personnalisé.

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
{
  message: 'Validation failed',
  name: 'ValidationError',
  errors: {
    nom: {
      message: 'Le nom est obligatoire',
      name: 'ValidatorError',
      path: 'nom',
      type: 'required',
      value: ''
    }
  }
}
Liste des clients :
[]
```

On voit maintenant que l'objet `err` contient le message d'erreur que nous avons indiqué dans le schéma. L'objet `err` contient une propriété `errors` qui indique les champs pour lesquels la validation a échoué (ici, le champ `nom`). Si d'autres validations existent sur d'autres champs, ceux-ci sont listés en tant que propriétés de l'objet `errors`.

Par exemple, indiquons que le nom et le prénom sont obligatoires, et tentons de sauvegarder un document qui ne les contient pas.

Validation sur le nom et le prénom

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : { type : String, required : "Le nom est obligatoire" },
  prenom : { type : String, required : "Le prénom est obligatoire" },
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "", prenom : "" });

c.save(function(err) {
  console.log(err);
  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});
```

Figure 25-4

Les champs "nom" et "prenom" produisent une erreur de validation.

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
{
  message: 'Validation failed',
  name: 'ValidationError',
  errors: {
    prenom: {
      message: 'Le prénom est obligatoire',
      name: 'ValidatorError',
      path: 'prenom',
      type: 'required',
      value: '' },
    nom: {
      message: 'Le nom est obligatoire',
      name: 'ValidatorError',
      path: 'nom',
      type: 'required',
      value: '' } }
}
Liste des clients :
[]
```

On voit que les champs `nom` et `prenom` sont présents dans l'objet `err.errors` du fait des deux erreurs de validation.

On souhaiterait afficher le texte des messages d'erreur plutôt que l'objet lui-même.

Afficher le texte des messages d'erreur

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : { type : String, required : "Le nom est obligatoire" },
  prenom : { type : String, required : "Le prénom est obligatoire" },
  adresse : {
    rue : String,
    ville : String,
    cp : String
  },
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

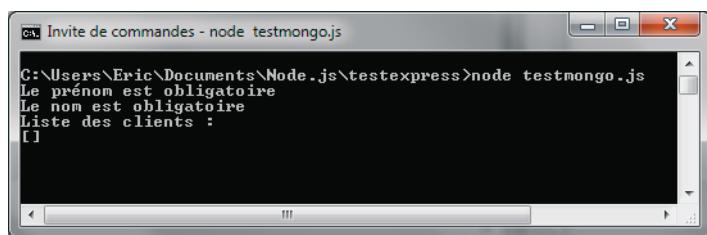
var c = new Client({ nom : "", prenom : "" });

c.save(function(err) {
  if (err) {
    for (var prop in err.errors) {
      var message = err.errors[prop].message;
      console.log(message);
    }
  }
  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});
```

On parcourt chacune des propriétés de l'objet `err.errors`, qui correspondent aux champs `nom` et `prenom`, puis on récupère le champ `message` de ces objets.

Figure 25–5

Affichage des textes associés aux erreurs de validation



Seul le texte du message d'erreur est maintenant affiché, en lieu et place de l'objet `err` précédent.

Validations par défaut de Mongoose

Nous avons utilisé la validation `required` indiquant la présence obligatoire d'un champ. D'autres critères de validation existent. Ils dépendent du type des champs sur lesquels ils sont positionnés. Par exemple, la clé `max` ne peut être positionnée que sur un champ de type `Number`, indiquant la valeur maximale que pourra prendre ce champ numérique.

Mettons en place une validation sur l'âge qui interdit d'avoir des clients de plus de 50 ans.

Interdire les clients de plus de 50 ans

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
    nom : { type : String, required : "Le nom est obligatoire" },
    prenom : { type : String, required : "Le prénom est obligatoire" },
    adresse : {
        rue : String,
        ville : String,
        cp : String
    },
    age : { type : Number, max : 50 }
});

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Sarrion", prenom : "Eric", age : 52 });

c.save(function(err) {
    if (err) {
        for (var prop in err.errors) {
```

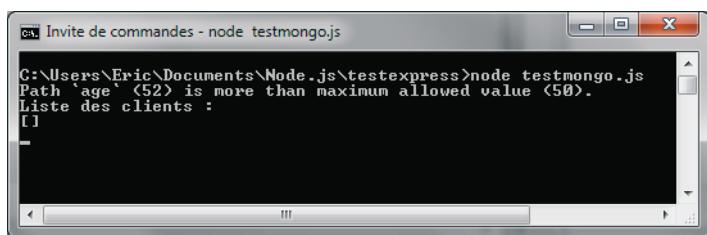
```

        var message = err.errors[prop].message;
        console.log(message);
    }
}
console.log("Liste des clients :");
Client.find(function(err, clients) {
    console.log(clients);
});
});

```

La clé `max` est positionnée sur le champ `age`, à la valeur 50 (valeur maximale que pourra prendre ce champ). Le client ajouté ayant 52 ans, il ne pourra pas être inséré dans la base de données.

Figure 25–6
Utilisation de la validation par défaut de Mongoose



Le message d'erreur par défaut de Mongoose s'est affiché. Si l'on souhaite insérer notre propre message d'erreur, il faudra l'indiquer à la place de la valeur 50. Pour cela, Mongoose permet d'affecter un tableau en tant que valeur de la clé, le tableau contenant la valeur 50 suivie du texte du message d'erreur (par exemple, `max : [50, "Pas d'âge supérieur à 50"]`).

Insérons notre message d'erreur si le champ `age` est supérieur à 50.

Indiquer notre propre message d'erreur de validation pour la clé `max`

```

var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
    nom : { type : String, required : "Le nom est obligatoire" },
    prenom : { type : String, required : "Le prénom est obligatoire" },
    adresse : {
        rue : String,
        ville : String,
        cp : String
    },
    age : { type : Number, max : [50, "Pas d'âge supérieur à 50"] }
});

```

```

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Sarrion", prenom : "Eric", age : 52 });

c.save(function(err) {
  if (err) {
    for (var prop in err.errors) {
      var message = err.errors[prop].message;
      console.log(message);
    }
  }
  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});

```

Figure 25–7

Utilisation de notre texte pour le message d'erreur

```

C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Pas d'age supérieur à 50
Liste des clients :
[]

```

Notre message d'erreur s'est affiché à la place du message d'erreur standard.

Le tableau 25-1 liste les critères de validation possibles selon le type des champs.

Tableau 25–1 Validations définies dans Mongoose

Clé de validation	Signification
required	Utilisable pour tous les types de champs. Vérifie que le champ contient une valeur. Par exemple, <code>nom : { type : String, required : true }</code> .
unique	Utilisable pour tous les types de champs. Vérifie que la valeur du champ est unique dans la collection. Par exemple, <code>nom : { type : String, unique: true }</code> .
min	Utilisable pour les champs de type <code>Number</code> . Vérifie que la valeur du champ est supérieure à la valeur minimale indiquée. Par exemple, <code>age : { type : Number, min : 50 }</code> .
max	Utilisable pour les champs de type <code>Number</code> . Vérifie que la valeur du champ est inférieure à la valeur maximale indiquée. Par exemple, <code>age : { type : Number, max : 50 }</code> .

Tableau 25–1 Validations définies dans Mongoose (suite)

Clé de validation	Signification
<code>match</code>	Utilisable pour les champs de type <code>String</code> . Vérifie que la valeur du champ correspond à l'expression régulière indiquée. Par exemple, <code>nom : { type : String, match : /Obama/ }</code> pour indiquer que le nom doit contenir la chaîne "Obama".
<code>enum</code>	Utilisable pour les champs de type <code>String</code> . Vérifie que la valeur du champ fait partie de l'énumération indiquée. Par exemple, <code>nom : { type : String, enum : ["Sarrion", "Node", "Obama"] }</code> pour indiquer que le nom doit être "Sarrion", "Node" ou "Obama".

Créer sa propre validation

Les validations par défaut proposées par Mongoose sont restreintes et ne permettent pas une grande flexibilité. Pour y remédier, Mongoose propose de créer ses propres méthodes de validation.

La fonction de validation (voir exemple ci-dessous) possède un paramètre `value` qui est la valeur du champ, et doit retourner `true` ou `false` selon que la validation réussit ou non. Le mot-clé `this` représente le document en cours de validation, ce qui permet d'avoir accès aux autres champs du document et pas seulement à celui traité par la méthode de validation.

Créons une validation sur le champ `age` permettant de s'assurer que l'âge des clients est compris entre 7 et 77 ans, sans passer par les clés `min` et `max` vues précédemment.

Vérifier que l'âge est compris entre 7 et 77 ans

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : {
    type : Number,
    required : "Le champ age est obligatoire",
    validate : function (value) {
      if (value < 7 || value > 77) return false; // échec
      else return true; // succès
    }
  }
});
```

```
var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Sarrion", prenom : "Eric", age : 80 });

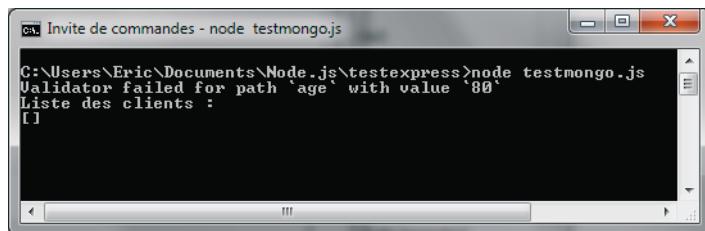
c.save(function(err) {
  if (err) {
    for (var prop in err.errors) {
      var message = err.errors[prop].message;
      console.log(message);
    }
  }

  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});
```

La fonction de validation est inscrite dans la clé `validate` associée au champ à valider.

Figure 25–8

Créer sa propre validation



L'âge du client est ici 80 ans , ce qui dépasse la limite de 77 ans autorisée. Une erreur de validation se produit.

Le message affiché correspond à celui géré par Mongoose. On peut le modifier pour produire un message spécifique. Le code du schéma est légèrement modifié pour en tenir compte.

Afficher un message d'erreur spécifique

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
```

```

prenom : String,
age : {
  type : Number,
  required : "Le champ age est obligatoire",
  validate : {
    validator : function (value) {
      if (value < 7 || value > 77) return false; // échec
      else return true; // succès
    },
    msg : "L'âge doit être compris entre 7 et 77 ans !"
  }
});
};

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Sarrion", prenom : "Eric", age : 80 });

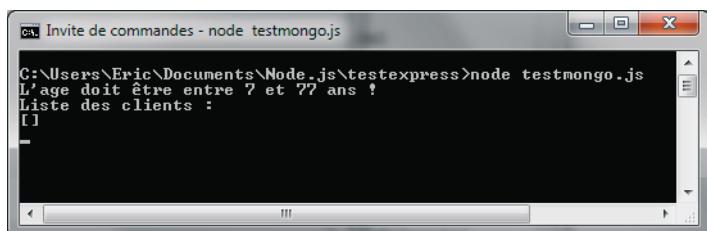
c.save(function(err) {
  if (err) {
    for (var prop in err.errors) {
      var message = err.errors[prop].message;
      console.log(message);
    }
  }

  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});

```

La clé `validate` correspond maintenant à un objet `{ validator, msg }` dans lequel la clé `validator` est la fonction de traitement, tandis que la clé `msg` correspond au texte associé au message d'erreur.

Figure 25-9
Affichage de notre texte pour le message d'erreur



Il est possible de positionner plusieurs méthodes de validation pour un champ. Il suffit de les indiquer dans la clé `validate` au moyen d'un tableau. Par exemple, si on scinde en deux la fonction de validation précédente, en traitant séparément les âges inférieurs à 7 et ceux supérieurs à 77, nous obtenons le code suivant.

Associer deux fonctions de validation sur un champ

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : {
    type : Number,
    required : "Le champ age est obligatoire",
    validate : [
      {
        validator : function (value) {
          if (value < 7) return false;
          else return true;
        },
        msg : "L'âge doit être supérieur à 7 !"
      },
      {
        validator : function (value) {
          if (value > 77) return false;
          else return true;
        },
        msg : "L'âge doit être inférieur à 77 !"
      }
    ]
  }
});

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Sarrion", prenom : "Eric", age : 80 });

c.save(function(err) {
  if (err) {
    for (var prop in err.errors) {
      var message = err.errors[prop].message;
      console.log(message);
    }
  }
});
```

```

    console.log("Liste des clients :");
    Client.find(function(err, clients) {
        console.log(clients);
    });
});
}

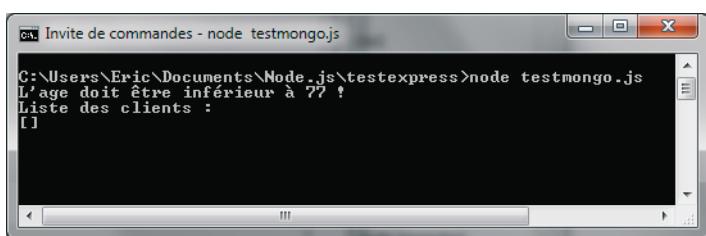
```

Selon que l'âge sera inférieur à 7 ou supérieur à 77, la fonction de validation adéquate produira son message d'erreur.

Par exemple, pour un âge de 80 ans :

Figure 25–10

La validation a produit une erreur, traitée par la seconde fonction de validation.



Validations asynchrones

Les précédentes validations étaient toutes synchrones, c'est-à-dire que le résultat de la validation (`true` ou `false`) était obtenu immédiatement dans la fonction de traitement. Supposons que cela ne soit pas le cas, par exemple si on cherche à valider le fait que le nom et le prénom d'un client soient uniques dans la collection. On connaît ce résultat uniquement après avoir interrogé la base de données, dans la fonction de callback de traitement de la méthode `findOne()`. Or, cette réponse arrive bien trop tard car la fonction de validation est déjà considérée comme terminée (traitement asynchrone).

Il faudrait donc demander à la fonction de validation de patienter pour envoyer sa réponse, afin que celle-ci soit envoyée dans la fonction de callback de la méthode `findOne()`. Pour cela, Mongoose demande que la fonction de validation possède deux paramètres.

- Le premier paramètre est `value`, qui correspond à la valeur du champ que l'on valide. C'est le même paramètre que pour une fonction de validation classique, telles que celles vues dans les sections précédentes.
- Le second paramètre est `respond`, qui correspond à une fonction de callback que la fonction de validation devra appeler pour donner sa réponse. L'appel à `respond(false)` signifie que la validation a échoué, tandis que l'appel à `respond(true)` signifie que la validation est correcte.

Écrivons la fonction de validation permettant de valider que deux clients n'ont pas des noms et prénoms identiques.

Interdire deux fois le même client (même nom et même prénom) dans la base de données

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : { type : String, validate : function(value, respond) {
    var nom = this.nom;
    var prenom = this.prenom;
    Client.findOne({ nom : nom, prenom : prenom }, function(err, client) {
      if (client) respond(false); // échec
      else respond(true); // succès
    });
  }},
  prenom : String,
  age : Number
});

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Sarrion", prenom : "Eric", age : 80 });

c.save(function(err) {
  if (err) {
    for (var prop in err.errors) {
      var message = err.errors[prop].message;
      console.log(message);
    }
  }

  console.log("Liste des clients :");
  Client.find(function(err, clients) {
    console.log(clients);
  });
});
```

L'objet `this` dans la fonction de validation correspond au document que l'on valide.

Lançons une première fois ce programme. Le client "Eric Sarrion" est inséré dans la collection car il n'est pas déjà présent.

Figure 25–11

La validation est effectuée avec succès, le client est inséré.

```
Invité de commandes - node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Liste des clients :
[ { nom: 'Sarrion',
  prenom: 'Eric',
  age: 80,
  _id: 532abc6f8ed873742bf39d15,
  __v: 0 } ]
```

Exécutons ce programme une seconde fois. Une erreur de validation apparaît.

Figure 25–12

La validation échoue, le client n'est pas inséré.

```
Invité de commandes - node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Liste des clients :
[ { nom: 'Sarrion',
  prenom: 'Eric',
  age: 80,
  _id: 532abc6f8ed873742bf39d15,
  __v: 0 } ]
^C
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Validator failed for path 'nom' with value 'Sarrion'
Liste des clients :
[ { nom: 'Sarrion',
  prenom: 'Eric',
  age: 80,
  _id: 532abc6f8ed873742bf39d15,
  __v: 0 } ]
```

Le message d'erreur est celui affiché par défaut dans Mongoose. On peut utiliser le nôtre en modifiant légèrement le code du schéma.

Afficher un message d'erreur de validation personnalisé

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : {
    type : String,
    validate : {
      validator : function(value, respond) {
        var nom = this.nom;
        var prenom = this.prenom;
        Client.findOne({ nom : nom, prenom : prenom }, function(err, client) {
          if (client) respond(false);
          else respond(true);
        });
      }
    }
});
```

```
        },
        msg : "Pas deux fois les mêmes noms et prénoms !"
    },
},
prenom : String,
age : Number
});

var Client = mongoose.model("Client", clientSchema);

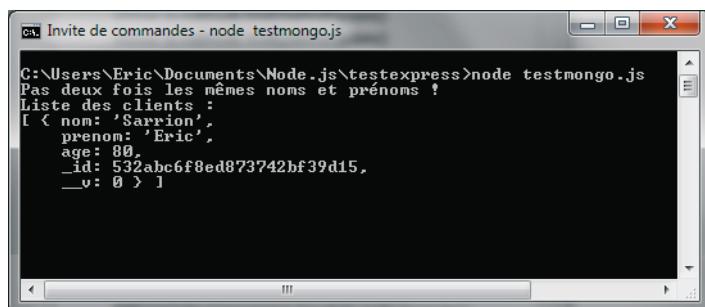
var c = new Client({ nom : "Sarrion", prenom : "Eric", age : 80 });

c.save(function(err) {
if (err) {
for (var prop in err.errors) {
var message = err.errors[prop].message;
console.log(message);
}
}
}

console.log("Liste des clients :");
Client.find(function(err, clients) {
console.log(clients);
});
});
```

La clé `validate` est transformée en objet { `validator`, `msg` } dans lequel `validator` est la fonction de traitement de la validation, et `msg` le texte du message à afficher en cas d'erreur.

Figure 25–13
Validation avec message
d'erreur spécifique



26

Utiliser le concept de population

Le concept de population est lié à l'utilisation de documents appartenant à différents modèles. Il est fréquent qu'un document utilise des références vers d'autres documents, par exemple un client est associé à des commandes qu'il effectue. On a ainsi un lien entre un client et les commandes qu'il passe, mais on a aussi un lien entre une commande et le client qui l'a passée.

Mongoose permet de gérer ces relations à travers la notion de population. Le schéma sera modifié pour tenir compte des diverses relations entre les données, et des nouvelles méthodes vont permettre de faciliter les recherches dans les différentes collections.

Indiquer les relations dans les schémas

La première chose à faire lorsqu'on utilise plusieurs collections ayant des relations entre elles est d'indiquer ces relations dans les schémas. Nous utilisons dans cet exemple les deux collections suivantes :

- la collection `clients`, sachant qu'un client peut effectuer 0 à N commandes ;
- la collection `commandes`, sachant qu'une commande est effectuée par un seul client.

Le code des schémas des collections `clients` et `commandes` est le suivant.

Schémas associés aux collections clients et commandes

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});

var commandeSchema = mongoose.Schema({
  date : Date,
  effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"}
});
```

On a ajouté les attributs `commandes` et `effectuePar` dans les schémas décrivant respectivement les collections `clients` et `commandes`.

- L'attribut `commandes` est un tableau qui contiendra les commandes du client. Chaque commande est décrite par un objet `{ type, ref }` dans lequel `type` est le type de la liaison (ici, l'identifiant de la commande sous forme `ObjectId`), et `ref` est le nom du modèle associé à la commande. Il correspond à la valeur indiquée dans le paramètre `modelName` utilisé dans la méthode `mongoose.model(modelName, schema)`.
- L'attribut `effectuePar` désigne le client qui a effectué la commande. Cet attribut est également décrit par un objet `{ type, ref }` dans lequel `type` est le type de la liaison (ici, l'identifiant du client sous forme `ObjectId`), tandis que `ref` est le nom du modèle associé au client (le même que le paramètre `modelName` utilisé dans `mongoose.model(modelName, schema)`).

Bien que le type de la liaison soit souvent utilisé sous la forme d'un identifiant `ObjectId`, Mongoose permet également de le décrire sous les formes suivantes.

Tableau 26–1 Différents types de liaisons dans les schémas

Type de la liaison	Signification
<code>mongoose.Schema.Types.ObjectId</code>	Un identifiant <code>ObjectId</code>
<code>Number</code>	Un entier
<code>String</code>	Une chaîne de caractères
<code>Buffer</code>	Un buffer d'octets

Les classes `Number`, `String` et `Buffer` sont internes à Node et n'ont donc pas besoin d'être préfixées, contrairement à la classe `ObjectId` qui est interne à Mongoose.

Ajout de documents dans les collections

Le programme précédent ne fait que décrire les schémas utilisés. Il reste maintenant à insérer des documents ayant cette structure dans les collections `clients` et `commandes`.

Création d'un client et d'une commande

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});

var commandeSchema = mongoose.Schema({
  date : Date,
  effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"},
});

var Client = mongoose.model("Client", clientSchema);
var Commande = mongoose.model("Commande", commandeSchema);

var client = new Client({ nom : "Obama", prenom : "Barack" });
client.save(function() {
  console.log(client + " sauvegardé");
  var cde = new Commande({ date : new Date(), effectuePar : client._id });
  cde.save();
  console.log("\n" + cde + " sauvegardée");
});
```

Une fois le client sauvégarde, un identifiant lui est attribué par MongoDB. Cet identifiant est utilisé pour créer l'objet de classe `Commande`, par l'initialisation de l'attribut `effectuePar`.

Figure 26–1

Un client et une commande ont été créés.

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
C:\Users\Eric> {
  __v: 0,
  nom: 'Obama',
  prenom: 'Barack',
  _id: 5336c73b5a7d5fec2a8cac8f,
  commandes: [ ] } sauvegardé
{
  date: Sat Mar 29 2014 14:14:36 GMT+0100 (Paris, Madrid),
  effectuePar: 5336c73b5a7d5fec2a8cac8f,
  _id: 5336c73c5a7d5fec2a8cac90 } sauvegardée
```

Le client est affiché en premier, puis la commande. La commande contient bien la référence au client par l'intermédiaire du champ `effectuePar`, mais le client contient un tableau `commandes` qui est vide. En effet, même si la commande a bien été sauvegardée dans la base de données, on n'a pas établi de lien entre cette commande et ce client. Il faudrait donc inclure la commande dans le tableau `commandes` du client. Cela peut être réalisé au moyen de la méthode `push()` de la classe `Array`.

Insérer la commande dans le tableau commandes du client

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});

var commandeSchema = mongoose.Schema({
  date : Date,
  effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"},
});

var Client = mongoose.model("Client", clientSchema);
var Commande = mongoose.model("Commande", commandeSchema);

var client = new Client({ nom : "Obama", prenom : "Barack" });
client.save(function() {
  var cde = new Commande({ date : new Date(), effectuePar : client._id });
  cde.save(function() {
    client.commandes.push(cde);
    client.save();
    console.log(client + " sauvégarde");
    console.log("\n" + cde + " sauvégarde");
  });
});
```

Remarquez que la commande est insérée dans le tableau, puis que le client est de nouveau sauvegardé dans la base de données. Si le client n'est pas sauvegardé une seconde fois, le tableau `commandes` sera uniquement modifié en mémoire, il ne sera pas sauvegardé dans la base de données.

Figure 26–2

Le tableau commandes du client est maintenant initialisé.

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
{
  nom: 'Obama',
  prenom: 'Barack',
  _id: 5336de7453d34a8c26594600,
  commandes: [ 5336de7553d34a8c26594609 ] > sauvegardé
}
< _v: 0,
  date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
  effectuePar: 5336de7453d34a8c26594600,
  _id: 5336de7553d34a8c26594609 > sauvegardée
```

Le tableau `commandes` du client est maintenant initialisé. L'identifiant inscrit dans le tableau est celui de la commande effectuée.

Recherche de documents dans les collections

Après avoir inséré des documents dans les collections `clients` et `commandes`, il reste à savoir comment les retrouver afin de les utiliser.

Pour cela, on utilise les méthodes précédemment étudiées, notamment la méthode `find()`. Utilisons cette méthode afin d'afficher la liste des clients et la liste des commandes stockées dans la base de données.

Afficher la liste des clients et la liste des commandes

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});

var commandeSchema = mongoose.Schema({
  date : Date,
  effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"},
});

var Client = mongoose.model("Client", clientSchema);
var Commande = mongoose.model("Commande", commandeSchema);

Client.find(function(err, clients) {
  console.log("Liste des clients :\n" + clients);
});
```

```
Commande.find(function(err, commandes) {
  console.log("\nListe des commandes :\n" + commandes);
});
```

Figure 26-3

Affichage de la liste des clients et des commandes

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Liste des clients :
{
  _v: 1,
  _id: 5336de7453d34a8c26594608,
  nom: 'Obama',
  prenom: 'Barack',
  commandes: [ 5336de7553d34a8c26594609 ]
}

Liste des commandes :
{
  date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
  effectuéPar: 5336de7453d34a8c26594608,
  _id: 5336de7553d34a8c26594609,
  __v: 0
}
```

La liste des clients (ici, un seul) s'affiche, chaque client étant composé des champs `_id`, `nom`, `prenom` et `commandes`. Remarquez que le tableau `commandes` comporte l'identifiant de chacune des commandes (ici, une seule). De même, la liste des commandes s'affiche, chaque commande comportant les champs `_id`, `date` et `effectuéPar` qui correspond à l'identifiant du client qui a effectué la commande.

On peut améliorer l'obtention des résultats. En effet, le tableau `commandes` d'un client comporte l'identifiant de la commande. Si l'on souhaite plus de détails sur cette commande, il faudrait la rechercher dans la collection `commandes`, à partir de son identifiant.

Mongoose effectue cette recherche de façon automatique si on le lui demande. Il peut en effet remplacer l'identifiant de la commande dans le tableau `commandes` par la commande elle-même. Cela sera plus pratique d'avoir l'objet `Commande` inclus dans le tableau `commandes`, plutôt que l'identifiant de la commande. On utilise pour cela la méthode `populate(attribut)` après la méthode `find()`, indiquant de remplacer la valeur de l'attribut par l'objet qu'il représente.

Remplacer l'identifiant de la commande par la commande elle-même dans le tableau commandes

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});

var commandeSchema = mongoose.Schema({
  date : Date,
```

```

effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"},  
});  
  
var Client = mongoose.model("Client", clientSchema);  
var Commande = mongoose.model("Commande", commandeSchema);  
  
Client.find().populate("commandes").exec(function(err, clients) {  
    console.log("Liste des clients :\n" + clients);  
});  
  
Commande.find(function(err, commandes) {  
    console.log("\nListe des commandes :\n" + commandes);  
});
```

L'instruction `find()` est suivie de la méthode `populate("commandes")`, elle-même suivie de l'appel de la méthode `exec()`. En effet, la méthode `populate()` ne déclenche pas directement la recherche, d'où l'utilisation de la méthode `exec(callback)`.

Figure 26–4

Le tableau commandes contient maintenant le descriptif de chaque commande.

```

C:\ Invité de commandes - node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js

Liste des commandes :
< date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
  effectuePar: 5336de7453d34a8c26594608,
  _id: 5336de7553d34a8c26594609,
  __v: 0 >
Liste des clients :
< __v: 1,
  _id: 5336de7453d34a8c26594608,
  nom: 'Obama',
  prenom: 'Barack',
  commandes:
    [ < date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
      effectuePar: 5336de7453d34a8c26594608,
      _id: 5336de7553d34a8c26594609,
      __v: 0 > ] >
```

Bien que la recherche de la liste des clients soit effectuée avant celle des commandes, elle s'affiche après celle-ci dans la console. En effet, la méthode `populate()` effectue une recherche supplémentaire dans la base de données, ce qui retarde l'appel de la fonction de callback en paramètre de la méthode `exec(callback)`.

La méthode `populate()` peut également s'utiliser sur l'autre liaison inscrite dans le schéma des commandes, qui utilise l'attribut `effectuePar`. Cela permettra d'accéder au client plutôt qu'à son identifiant seulement. On peut alors écrire :

Remplacer l'identifiant du client par le client lui-même dans une commande

```

var mongoose = require("mongoose");  
  
var db = mongoose.connect("mongodb://localhost/mydb");
```

```

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});

var commandeSchema = mongoose.Schema({
  date : Date,
  effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"},
});

var Client = mongoose.model("Client", clientSchema);
var Commande = mongoose.model("Commande", commandeSchema);

Client.find().populate("commandes").exec(function(err, clients) {
  console.log("Liste des clients :\n" + clients);
});

Commande.find().populate("effectuePar").exec(function(err, commandes) {
  console.log("\nListe des commandes :\n" + commandes);
});

```

Figure 26–5

Le descriptif du client est inséré dans chaque commande.

```

C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Liste des clients :
[
  {
    _v: 1,
    _id: 5336de7453d34a8c26594608,
    nom: 'Obama',
    prenom: 'Barack',
    commandes: [
      {
        date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
        effectuePar: 5336de7453d34a8c26594608,
        _id: 5336de7553d34a8c26594609,
        _v: 0
      }
    ]
  }
]

Liste des commandes :
[
  {
    date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
    effectuePar:
      {
        _v: 1,
        _id: 5336de7453d34a8c26594608,
        nom: 'Obama',
        prenom: 'Barack',
        commandes: [
          5336de7553d34a8c26594609
        ],
        _v: 0
      }
  }
]

```

Le champ `effectuePar` situé dans une commande permet maintenant d'accéder complètement au client (attributs `_id`, `nom`, `prenom` et `commandes`).

Utiliser la méthode `populate()`

La méthode `populate(attribut)` utilisée précédemment permet de remplacer la valeur d'un attribut par un objet récupéré dans une collection de la base de données. La relation entre l'attribut et l'objet est établie au moyen de la description réalisée dans le schéma.

La méthode `populate()` peut s'utiliser sur différents types d'objets. Nous l'avons précédemment utilisée sur l'objet de classe `mongoose.Query` retourné par la méthode `find()`. Elle peut également être utilisée sur le modèle ou sur un document particulier. Nous examinons ci-après ces différentes possibilités.

Utiliser la méthode `populate()` sur l'objet `mongoose.Query`

C'est l'utilisation la plus fréquente de la méthode `populate()`. Les documents retournés par la méthode `find()` (ou ses dérivées telle que `findOne()`) sont remplis en remplaçant les attributs indiqués en paramètres par les documents correspondants.

Tableau 26-2 Méthode `populate(attribut, select)`

Méthode	Signification
<code>mongoose.Query.populate(attribut, select)</code>	Remplace l'attribut (ou les attributs, séparés par un espace) dans les documents retournés par l'objet <code>mongoose.Query</code> qui utilise la méthode, par les documents correspondants. Le paramètre <code>attribut</code> est une chaîne de caractères contenant un ou plusieurs attributs, séparés par des espaces, et indiquant les documents à rechercher. Le paramètre <code>select</code> est une chaîne de caractères contenant un ou plusieurs attributs, séparés par des espaces, et indiquant les attributs à récupérer dans les documents recherchés.

La méthode `populate()` ici décrite doit toujours être suivie de la méthode `exec()`, car c'est cette dernière qui déclenche la recherche dans la base de données.

On souhaite ici afficher toutes les commandes des clients, en incluant seulement le nom du client. La méthode `populate("effectuePar", "nom")` exécutée sur les objets retournés par `Commande.find()` devrait nous permettre d'y arriver.

Afficher tous les noms des clients qui ont effectué des commandes

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});
```

```

var commandeSchema = mongoose.Schema({
  date : Date,
  effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"},
});

var Client = mongoose.model("Client", clientSchema);
var Commande = mongoose.model("Commande", commandeSchema);

Commande.find().populate("effectuePar", "nom").exec(function(err, commandes) {
  console.log("\nListe des commandes :\n" + commandes);
});

```

Figure 26–6

Seul le nom du client est récupéré dans la commande.

```

C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Liste des commandes :
{
  date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
  effectuePar: {
    _id: 5336de7453d34a8c26594608,
    nom: 'Obama',
    _v: 0
  }
}

```

Seul le nom (et l'identifiant) du client est maintenant récupéré par la méthode `populate()`, qui ne récupère plus ni le prénom ni le tableau des commandes.

Utiliser la méthode `populate()` sur le modèle

On utilisera cette forme de la méthode `populate()` si on a déjà obtenu les documents recherchés, mais que l'on souhaite remplacer certaines valeurs d'attributs par des objets issus d'autres documents. Cela permet de faire comme la méthode `populate()` précédente, mais en agissant après que les documents ont été obtenus (et non pas pendant).

Tableau 26–3 Méthode `populate(docs, options, callback)`

Méthode	Signification
<code>ModelName.populate(docs, options, callback)</code>	Peuple le ou les documents indiqués, à partir des options indiquées. Le paramètre <code>options</code> est un objet comprenant les champs suivants : <ul style="list-style-type: none"> - <code>path</code> : chaîne de caractères contenant les attributs (séparés par des espaces) à remplacer par les documents correspondants ; - <code>select</code> : chaîne de caractères contenant les attributs (séparés par des espaces) récupérés dans les documents recherchés. La fonction de callback (optionnelle) est de la forme <code>function (err, docs)</code> dans laquelle <code>docs</code> représente les documents retournés (un tableau de documents si <code>mongoose.Query</code> retourne un tableau, sinon un seul document). Si la fonction de callback n'est pas indiquée, utiliser alors la méthode <code>exec(callback)</code> à la suite.

Utilisons cette forme de la méthode `populate()` afin de récupérer la liste des commandes, en associant chaque commande au client qui l'a effectuée. Le code suivant équivaut à celui de la section précédente.

Utiliser Commande.populate()

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});

var commandeSchema = mongoose.Schema({
  date : Date,
  effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"},
});

var Client = mongoose.model("Client", clientSchema);
var Commande = mongoose.model("Commande", commandeSchema);

Commande.find(function(err, commandes) {
  Commande.populate(commandes, { path : "effectuePar", select : "nom" },
    function (err, cdes) {
      console.log("\nListe des commandes :\n" + cdes);
    });
});
```

La méthode `populate()` est ici appelée sur l'objet `commandes` retourné par la méthode `find()`.

Figure 26–7

Le nom du client a été associé à la commande.

```
C:\ Invité de commandes - node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Liste des commandes :
  { date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
    effectuePar: { _id: 5336de7453d34a8c26594608, nom: 'Obama' },
    _id: 5336de7553d34a8c26594609,
    __v: 0 }
```

Utiliser la méthode `populate()` sur un document

Lorsqu'un seul document doit être peuplé, il est possible d'utiliser la méthode `populate(attribut, select, callback)` sous la forme d'une méthode d'instance de l'objet associé au document.

Tableau 26–4 Méthode populate(attribut, select, callback)

Méthode	Signification
doc.populate(attribut, select, callback)	Peuple le document pour le ou les attributs indiqués, sous forme de chaîne de caractères, séparés par des espaces. Le paramètre <code>select</code> (optionnel) est une chaîne de caractères indiquant les attributs à récupérer dans les documents recherchés. La fonction de callback est de la forme <code>function(err, doc)</code> dans laquelle <code>doc</code> est le document peuplé.

Peupler un seul document

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number,
  commandes : [{type : mongoose.Schema.Types.ObjectId, ref : "Commande"}]
});

var commandeSchema = mongoose.Schema({
  date : Date,
  effectuePar : {type : mongoose.Schema.Types.ObjectId, ref : "Client"},
});

var Client = mongoose.model("Client", clientSchema);
var Commande = mongoose.model("Commande", commandeSchema);

Commande.findOne(function(err, commande) {
  commande.populate("effectuePar", "nom", function (err, cde) {
    console.log("\nListe des commandes :\n" + cde);
  });
});
```

Figure 26–8

Le nom du client a été associé à la commande.

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Liste des commandes :
{
  date: Sat Mar 29 2014 15:53:41 GMT+0100 (Paris, Madrid),
  effectuePar: {
    _id: 5336de7453d34a8c26594608,
    nom: 'Obama',
    __v: 0
  }
}
```

27

Utiliser les middlewares dans Mongoose

Mongoose permet de définir des middlewares tels que ceux définis dans les modules Connect et Express. Ces morceaux de programmes se déclencheront à des moments précis de la vie des données :

- lors de la sauvegarde de documents (méthodes `save()` et `create()`) ;
- lors de la validation d'un document ;
- lors de la suppression d'un document (méthode `remove()`).

Les autres méthodes telles que `update()`, `findOneAndUpdate/Remove()` et `findByIdAndUpdate/Remove()` ne sont pas concernées par les middlewares décrits ici.

De plus, certains middlewares permettront de décider si l'on poursuit ou non le traitement en cours sur la donnée, par exemple :

- accepter la sauvegarde de la donnée ou non ;
- accepter sa validation ou non ;
- accepter sa suppression ou non.

Deux sortes de middlewares sont définis dans Mongoose.

- Les `pre` middlewares : ils se déclenchent avant que l'opération ait lieu (sauvegarde, validation ou suppression) et peuvent accepter ou non que cette opération ait lieu.
- Les `post` middlewares : ils se déclenchent après la fin de l'opération (sauvegarde, validation ou suppression) et ne peuvent donc pas interférer avec la fin de celle-ci.

Ces deux types de middlewares sont étudiés dans les sections suivantes.

Utiliser les pre middlewares

Les `pre` middlewares peuvent s'utiliser sur les opérations de sauvegarde, validation ou suppression de documents.

Méthode `pre()` définissant un middleware

La méthode `pre(type, callback)`, utilisable sur le schéma, permet de créer un middleware du type indiqué. Selon l'opération que l'on souhaite traiter, on utilisera un des trois types suivants :

- `"save"` : permet de créer un middleware gérant la sauvegarde des documents associés au schéma ;
- `"validate"` : permet de créer un middleware gérant la validation des documents associés au schéma ;
- `"remove"` : permet de créer un middleware gérant la suppression des documents associés au schéma.

Tableau 27-1 Méthode `pre(type, callback)` définie sur le schéma

Méthode	Signification
<code>pre(type, callback)</code>	Définit un <code>pre</code> middleware, qui sera déclenché avant que l'opération ait lieu. Cette dernière est définie par le paramètre <code>type</code> (<code>"save"</code> , <code>"validate"</code> ou <code>"remove"</code>). La fonction de callback est de la forme <code>function(next)</code> dans laquelle <code>next</code> est une autre fonction de callback à appeler pour indiquer de continuer l'opération. Si la méthode <code>next()</code> n'est pas appelée, l'opération est annulée.

Le mot-clé `this` utilisé dans la fonction de callback représente le document en cours de traitement.

Voyons comment utiliser la méthode `pre()` pour chaque type d'opération possible.

Utiliser un pre middleware lors de la sauvegarde d'un document

On peut utiliser un `pre` middleware qui sera déclenché lors de la sauvegarde d'un document (création ou mise à jour). On aura également la possibilité d'indiquer si la sauvegarde peut être effectuée ou non.

Utiliser un pre middleware pour sauvegarder un document

```
var mongoose = require("mongoose");
var db = mongoose.connect("mongodb://localhost/mydb");
```

```

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number
});

clientSchema.pre("save", function (next) {
  console.log("Avant la sauvegarde du document");
  next(); // continuer l'opération
});

var Client = mongoose.model("Client", clientSchema);

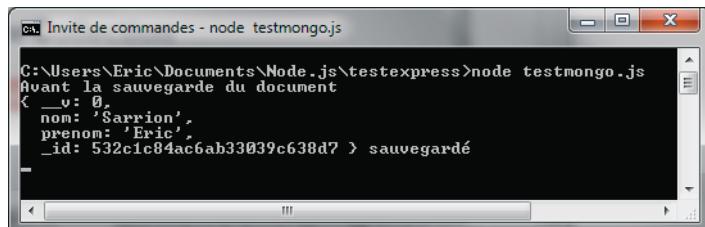
var c = new Client({ nom : "Sarrion", prenom : "Eric" });
c.save(function(err, client) {
  if (client) console.log(client + " sauvegardé");
});

```

On définit la méthode `pre("save", callback)` sur le schéma. La fonction de callback est de la forme `function(next)` dans laquelle `next` est une fonction à appeler pour continuer l'opération. Si elle n'est pas appelée, l'opération n'est pas poursuivie (ici, le document ne serait donc pas sauvegardé).

Figure 27–1

Un message a été affiché avant la sauvegarde du document.



L'affichage du message "`Avant la sauvegarde du document`" dans la console du serveur montre que le middleware est appelé avant que le document soit sauvegardé.

Testons le fait de ne pas appeler la méthode `next()` dans le middleware. On tente d'ajouter un nouveau client de nom "`Barack Obama`".

Ajouter un nouveau client Barack Obama

```

var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,

```

```

        prenom : String,
        age : Number
    });

clientSchema.pre("save", function (next) {
    console.log("Avant la sauvegarde du document");
// next(); // continuer l'opération
});

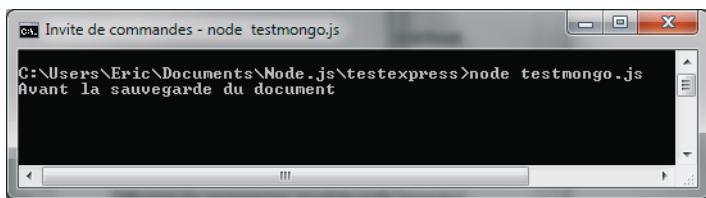
var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Obama", prenom : "Barack" });
c.save(function(err, client) {
    if (client) console.log(client + " sauvegardé");
});

```

Figure 27–2

Le document n'a pas été sauvegardé.



Le `pre` middleware est bien appelé, mais le document n'est pas sauvegardé car la méthode `next()` n'est pas appelée.

Utiliser un `pre` middleware lors de la validation d'un document

Un `pre` middleware peut également être utilisé pour la validation d'un document. Le middleware est appelé lors de la sauvegarde du document, même si aucune validation n'est demandée sur celui-ci.

Utiliser un `pre` middleware pour la validation d'un document

```

var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
    nom : String,
    prenom : String,
    age : Number
});

```

```

clientSchema.pre("validate", function (next) {
  console.log("Avant la validation du document");
  next(); // continuer l'opération
});

clientSchema.pre("save", function (next) {
  console.log("Avant la sauvegarde du document");
  next(); // continuer l'opération
});

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Obama", prenom : "Barack" });
c.save(function(err, client) {
  if (client) console.log(client + " sauvegardé");
});

```

Nous avons utilisé les deux `pre` middlewares `save` et `validate`, ce qui permet de voir l'ordre des appels.

Figure 27–3
Pre middlewares sur
la validation et la sauvegarde
d'un document

La validation est effectuée (même si aucune n'est présente dans le schéma), puis la sauvegarde.

Supprimons l'instruction `next()` dans la partie `validate`, en la mettant en commentaire. On obtient alors :

Figure 27–4
Le document n'a pas
été sauvegardé.

Le document n'est plus sauvegardé dans la base de données car sa validation a été interrompue.

Utiliser un pre middleware lors de la suppression d'un document

La méthode d'instance `doc.remove()` fait appel au middleware `remove` positionné sur le schéma.

Utiliser un pre middleware pour supprimer un document

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number
});

clientSchema.pre("validate", function (next) {
  console.log("Avant la validation du document");
  next(); // continuer l'opération
});

clientSchema.pre("save", function (next) {
  console.log("Avant la sauvegarde du document");
  next(); // continuer l'opération
});

clientSchema.pre("remove", function (next) {
  console.log("Avant la suppression du document");
  next(); // continuer l'opération
});

var Client = mongoose.model("Client", clientSchema);

Client.findOne({ nom : "Sarrion", prenom : "Eric" }, function(err, client) {
  if (client) {
    client.remove(function(err, c ) {
      console.log(client + " supprimé");
    });
  }
});
```

Nous recherchons le document à supprimer (le premier client de nom "Eric Sarrion"), puis nous le supprimons de la collection. Le `pre` middleware positionné sur `remove` doit s'exécuter.

Figure 27-5

Utilisation d'un pre middleware "remove"

```
c:\ Invité de commandes - node testmongo.js
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Avant la suppression du document
{
  nom: 'Sarrion',
  prenom: 'Eric',
  _id: 532db19deaa73e343998527e,
  __v: 0
} supprimé
```

Le `pre` middleware `remove` s'est exécuté, ainsi que la suppression réelle du document dans la collection. Supprimons l'instruction `next()` dans le code du middleware : le middleware `remove` est toujours appelé, mais le document n'est plus supprimé.

Si vous souhaitez utiliser la méthode de classe `remove()` (au lieu de la méthode d'instance `doc.remove()` comme précédemment), cela est évidemment possible mais il faut savoir que celle-ci ne fait pas appel aux middlewares, comme on peut le vérifier dans l'exemple suivant.

La méthode de classe `remove()` ne fait pas appel aux middlewares

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number
});

clientSchema.pre("validate", function (next) {
  console.log("Avant la validation du document");
  next(); // continuer l'opération
});

clientSchema.pre("save", function (next) {
  console.log("Avant la sauvegarde du document");
  next(); // continuer l'opération
});

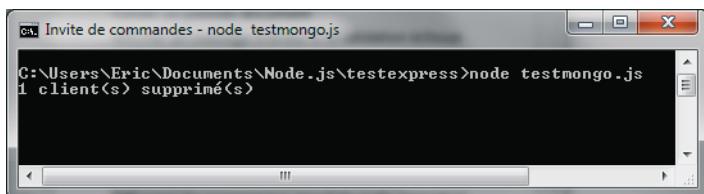
clientSchema.pre("remove", function (next) {
  console.log("Avant la suppression du document");
  next(); // continuer l'opération
});
```

```
var Client = mongoose.model("Client", clientSchema);

Client.remove({ nom : "Sarrion", prenom : "Eric" }, function(err, countRemoved)
{
  if (countRemoved) console.log(countRemoved + " client(s) supprimé(s)");
});
```

Figure 27–6

La méthode de classe `remove()` ne fait pas appel aux middlewares.



Bien que le `pre` middleware `remove` ait été installé, il n'est pas appelé car la méthode de classe `remove()` ne fait pas appel aux middlewares.

Utiliser le mot-clé `this` dans un `pre` middleware

Le mot-clé `this` utilisé dans la fonction de callback d'un `pre` middleware, représente le document en cours de traitement (validation, sauvegarde ou suppression). On peut l'utiliser pour décider du traitement à effectuer dans la fonction de callback du middleware.

Par exemple, on souhaite empêcher que certains documents soient supprimés. En particulier, on interdit la suppression du client "Obama".

Interdire la suppression du client "Obama"

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number
});

clientSchema.pre("remove", function (next) {
  if (this.nom == "Obama")
    console.log("On ne peut pas supprimer le client " + this.nom );
  else next(); // continuer l'opération
});
```

```

var Client = mongoose.model("Client", clientSchema);

Client.findOne({ nom : "Obama", prenom : "Barack" }, function(err, client) {
  if (client) {
    client.remove(function(err, c ) {
      console.log(client + " supprimé");
    });
  }
});

```

Si le nom du client est "Obama", on affiche un message d'erreur, sinon on accepte la suppression du document en appelant `next()`.

Figure 27-7

Le client ne peut pas être supprimé.



Utiliser les post middlewares

De la même façon que les `pre` middlewares, les `post` middlewares peuvent s'utiliser sur les opérations de sauvegarde, validation ou suppression de documents.

Étant appelés une fois l'opération terminée, les `post` middlewares ne peuvent pas être utilisés pour arrêter l'opération, qui ira à son terme sauf si un `pre` middleware l'arrête.

Méthode `post()` définissant un middleware

La méthode `post(type, callback)` utilisable sur le schéma permet de créer un middleware du type indiqué. Selon l'opération que l'on souhaite traiter, on utilisera un des trois types suivants (les mêmes que pour les `pre` middlewares) :

- "`save`" : permet de créer un middleware gérant la sauvegarde des documents associés au schéma ;
- "`validate`" : permet de créer un middleware gérant la validation des documents associés au schéma ;
- "`remove`" : permet de créer un middleware gérant la suppression des documents associés au schéma.

Tableau 27-2 Méthode post(type, callback) définie sur le schéma

Méthode	Signification
post(type, callback)	Définit un <code>post</code> middleware, qui sera déclenché après que l'opération a lieu. L'opération est définie par le paramètre <code>type</code> ("save", "validate" ou "remove"). La fonction de callback est de la forme <code>function()</code> et elle est appelée à la fin de l'opération, sauf si un <code>pre</code> middleware a arrêté l'opération.

Le mot-clé `this` utilisé dans la fonction de callback représente le document en cours de traitement.

Utiliser un post middleware

Le programme suivant implémente les trois `post` middlewares, mais également les trois `pre` middlewares. Cela permet de voir l'enchaînement d'appel des méthodes.

Enchaînement d'appel des middlewares

```
var mongoose = require("mongoose");

var db = mongoose.connect("mongodb://localhost/mydb");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  age : Number
});

clientSchema.pre("validate", function (next) {
  console.log("Avant la validation du document");
  next(); // continuer l'opération
});

clientSchema.pre("save", function (next) {
  console.log("Avant la sauvegarde du document");
  next(); // continuer l'opération
});

clientSchema.pre("remove", function (next) {
  console.log("Avant la suppression du document");
  next(); // continuer l'opération
});

clientSchema.post("validate", function (next) {
  console.log("Après la validation du document");
});
```

```

clientSchema.post("save", function (next) {
  console.log("Après la sauvegarde du document");
});

clientSchema.post("remove", function (next) {
  console.log("Après la suppression du document");
});

var Client = mongoose.model("Client", clientSchema);

var c = new Client({ nom : "Obama", prenom : "Barack" });
c.save(function(err, client) {
  if (client) console.log(client + " sauvé");
  Client.findOne({ nom : "Obama", prenom : "Barack" }, function(err, client) {
    if (client) {
      client.remove(function(err, c ) {
        console.log(client + " supprimé");
      });
    }
  });
});

```

Nous effectuons ici la création d'un client, suivie de sa suppression. Nous obtenons :

Figure 27-8
Enchaînement des middlewares

```

C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Avant la validation du document
Après la validation du document
Avant la sauvegarde du document
Après la sauvegarde du document
{
  __v: 0,
  nom: 'Obama',
  prenom: 'Barack',
  _id: 532dd750003fdadc2f260191 } sauvé
Avant la suppression du document
Après la suppression du document
{
  nom: 'Obama',
  prenom: 'Barack',
  _id: 532dd750003fdadc2f260191,
  __v: 0 } supprimé

```

Les middlewares sont enchaînés les uns à la suite des autres, le `pre` middleware étant suivi du `post` middleware.

Enlevons l'appel à la méthode `next()` dans le `pre` middleware de validation. On obtient alors le résultat suivant.

Figure 27–9

Suppression de l'enchaînement des middlewares avec next()

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Avant la validation du document
```

Toutes les opérations suivantes (sauvegarde, suppression) sont arrêtées, ainsi que les middlewares correspondants.

Remettons l'appel à `next()` dans le `pre` middleware de validation et supprimons celui inscrit dans la sauvegarde. On obtient alors le résultat suivant.

Figure 27–10

Suppression de l'enchaînement des middlewares avec next()

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Avant la validation du document
Après la validation du document
Avant la sauvegarde du document
```

La validation est effectuée mais pas la sauvegarde ni la suppression.

Enfin, remettons tous les appels à `next()` dans les `pre` middlewares, sauf pour celui de suppression. On obtient :

Figure 27–11

Suppression de l'enchaînement des middlewares avec next()

```
C:\Users\Eric\Documents\Node.js\testexpress>node testmongo.js
Avant la validation du document
Après la validation du document
Avant la sauvegarde du document
Après la sauvegarde du document
{
  "_v": 0,
  "nom": "Obama",
  "prenom": "Barack",
  "_id": 532dd944b878fcfc83e774b8e } sauvegardé
Avant la suppression du document
```

La validation et la sauvegarde sont effectuées, mais pas la suppression.

28

Construction d'une application client serveur

Nous avons jusqu'ici étudié les différents composants d'une application web construite avec Node, qui utilise le framework Express et la base de données MongoDB. Dans ce chapitre, nous allons mettre tous ces composants ensemble dans une même application.

Nous utiliserons pour cela la même application que celle construite dans le chapitre 18, « Crée les vues avec EJS ». Nous avions réalisé une petite application gérant une base de clients, mais celle-ci résidait en mémoire. Maintenant que nous savons stocker des informations dans une base de données MongoDB, nous l'utiliserons afin de pérenniser notre application.

Afin de montrer un exemple complet, nous réaliserons notre application selon deux organisations différentes.

- La première utilisera le principe REST, comme la précédente application que nous avions écrite (en utilisant le module `express-resource`). La nouveauté sera ici l'utilisation de la base de données MongoDB au lieu d'un simple stockage en mémoire des informations.
- La seconde n'utilisera pas REST, mais sera basée sur l'écriture des routes directement dans le fichier `app.js` (donc sans utiliser le module `express-resource` utilisant REST).

Application construite en utilisant REST

Nous reprenons ici les différents fichiers que nous avions écrits précédemment, à savoir :

- le fichier `app.js` correspondant au module principal de l'application Express ;
- le fichier `clients.js` qui est associé au module `express-resource` ;
- et les différentes vues que nous avions écrites (`clients.ejs`, `edit.ejs`, `new.ejs` et `show.ejs`).

Afin d'utiliser la base de données MongoDB, seul le fichier `clients.js` doit être modifié. Dans la version précédente, il stockait les clients dans un tableau en mémoire. Nous devons maintenant modifier cette gestion afin que les clients soient conservés dans une base de données MongoDB.

Fichier `clients.js` (qui utilise MongoDB via Mongoose)

```
var mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/clients");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});

var Client = mongoose.model("Client", clientSchema);

exports.index = function(req, res) {
  Client.find(function(err, clients) {
    res.render("clients/clients.ejs", { clients : clients });
  });
};

exports.new = function(req, res) {
  res.render("clients/new.ejs");
};

exports.create = function(req, res) {
  var nom = req.param("nom");
  var prenom = req.param("prenom");
  Client.create({ nom : nom, prenom : prenom }, function(err, client) {
    res.redirect("/clients");
  });
};
```

```
exports.show = function(req, res) {
  var id = req.params.client;
  Client.findById(id, function(err, client) {
    res.render("clients/show.ejs", { client : client });
  });
};

exports.edit = function(req, res) {
  var id = req.params.client;
  Client.findById(id, function(err, client) {
    res.render("clients/edit.ejs", { client : client });
  });
};

exports.update = function(req, res) {
  var nom = req.param("nom");
  var prenom = req.param("prenom");
  var id = req.params.client;
  Client.findByIdAndUpdate(id, { nom : nom, prenom : prenom }, function(err, client) {
    res.redirect("/clients");
  });
};

exports.destroy = function(req, res) {
  var id = req.params.client;
  Client.findByIdAndRemove(id, function(err, client) {
    res.redirect("/clients");
  });
};
```

Nous avons mis en évidence les différents appels à l'API Mongoose qui utilisent le modèle `Client` défini au moyen de `mongoose.model()`.

Même si le seul fichier à être modifié est le fichier `clients.js`, voici à nouveau le code des autres fichiers de l'application afin que vous ayez tout ensemble.

Fichier app.js

```
/**
 * Module dependencies
 */

var express = require('express');
var http = require('http');
```

```
var path = require('path');
var util = require('util');
var resource = require('express-resource');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public'))));
app.use(express.static(path.join(__dirname, 'forms'))));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.resource("clients", require("./clients.js"));

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Cette application possède les quatre vues suivantes :

- `clients.ejs` pour afficher la liste des clients ;
- `new.ejs`, qui permet d'afficher les informations permettant de créer un client ;
- `show.ejs` pour afficher les informations sur le client, sans modification possible ;
- `edit.ejs`, qui est similaire à `show.ejs`, mais permet de modifier le client et de valider sa modification.

Ces quatre vues sont situées dans le répertoire des vues de l'application, identifié par l'instruction `app.set('views', __dirname + '/views')` incluse dans `app.js`. De plus, comme les méthodes `res.render()` utilisées dans `app.js` préfixent chaque nom de fichier par la chaîne "`clients/`", cela permet d'avoir toutes les vues de l'application situées dans le répertoire `views/clients`. Comme précédemment, ces vues sont placées dans le répertoire `views/clients` de l'application.

Fichier clients.ejs (dans views/clients)

```
<h1> Liste des clients </h1>
<% if (!clients.length) { %>
<b> Pas de clients ! </b>
<br><br>
<% } else { %>
<table>
<% clients.forEach(function(client) { %>
<tr>
<td width=30><%= client.id %></td>
<td width=100><%= client.nom %></td>
<td width=100><%= client.prenom %></td>
<td>
<form style="display:inline" action="/clients/<%= client.id %>">
<input type="submit" value="Voir" />
</form>
</td>
<td>
<form style="display:inline" action="/clients/<%= client.id %>/edit">
<input type="submit" value="Modifier" />
</form>
</td>
<td>
<form style="display:inline" action="/clients/<%= client.id %>" method="post">
<input type="submit" value="Supprimer" />
<input type="hidden" name="_method" value="delete" />
</form>
</td>
</tr>
<% }); %>
</table>
<% } %>

<form action="/clients/new">
<input type="submit" value="Créer un client" />
</form>
```

Fichier new.ejs (dans views/clients)

```
<h1> Création de client </h1>

<form action="/clients" method="post">

<table>
<tr>
<td>Nom</td>
```

```
<td><input type="text" name="nom" /></td>
</tr>
<tr>
<td>Prénom</td>
<td><input type="text" name="prenom" /></td>
</tr>
</table>

<br>
<input type="submit" value="Valider" />

</form>
```

Fichier show.ejs (dans views/clients)

```
<h1> Visualisation de client <h1>

<form action="/clients" method="get">

<table>
<tr>
<td>Nom</td>
<td><input type="text" name="nom" disabled value = "<%=client.nom %>" /></td>
</tr>
<tr>
<td>Prénom</td>
<td><input type="text" name="prenom" disabled value = "<%=client.prenom %>" />
</td>
</tr>
</table>

<br>
<input type="submit" value="Fermer" />

</form>
```

Fichier edit.ejs (dans views/clients)

```
<h1> Modification de client <h1>

<form action="/clients/<%=client.id %>" method="post">

<table>
<tr>
<td>Nom</td>
<td><input type="text" name="nom" value = "<%=client.nom %>" /></td>
</tr>
```

```
<tr>
<td>Prénom</td>
<td><input type="text" name="prenom" value = "<%=client.prenom %>" /></td>
</tr>
</table>

<br>
<input type="hidden" name="_method" value="put" />
<input type="submit" value="Valider" />

</form>
```

Application construite sans utiliser REST

La même application peut être construite sans utiliser REST. Elle est plus simple d'approche car on n'a plus besoin de connaître les spécificités du module `express-resource` (tels que les noms de fichiers ou de méthodes à utiliser). En revanche, elle nécessite d'écrire toutes les routes utilisées dans le fichier `app.js` (ou dans un module inclus par celui-ci), ce qui était fait par défaut par `express-resource`.

Le programme comporte maintenant un module principal qui est le fichier `app.js`, et les quatre vues précédentes qui permettent de créer, visualiser, modifier ou lister les clients :

- `clients.ejs` pour afficher la liste des clients ;
- `new.ejs`, qui permet d'afficher les informations permettant de créer un client ;
- `show.ejs` pour afficher les informations sur le client, sans modification possible ;
- `edit.ejs`, qui est similaire à `show.ejs`, mais permet de modifier le client et de valider sa modification.

Fichier app.js

```
/***
 * Module dependencies.
 */

var mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/clients");

var clientSchema = mongoose.Schema({
  nom : String,
  prenom : String,
  adresse : String
});
```

```
var Client = mongoose.model("Client", clientSchema);

var express = require('express');
var http = require('http');
var path = require('path');
var util = require('util');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public'))));
app.use(express.static(path.join(__dirname, 'forms'))));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

// lister les clients
app.get("/clients", function(req, res) {
  Client.find(function(err, clients) {
    res.render("clients/clients.ejs", { clients : clients });
  });
});

// visualiser un client sans modification possible
app.get("/clients/voir/:id", function(req, res) {
  var id = req.params.id;
  Client.findById(id, function(err, client) {
    res.render("clients/show.ejs", { client : client });
  });
});

// visualiser un client avec modification possible
app.get("/clients/modifier/:id", function(req, res) {
  var id = req.params.id;
  Client.findById(id, function(err, client) {
    res.render("clients/edit.ejs", { client : client });
  });
});
```

```
// supprimer un client
app.get("/clients/supprimer/:id", function(req, res) {
    var id = req.params.id;
    Client.findByIdAndRemove(id, function(err, client) {
        res.redirect("/clients");
    });
});

// valider un client dans la base de données (création ou modification)
app.get("/clients/valider/:id?", function(req, res) {
    var id = req.params.id;
    var nom = req.param("nom");
    var prenom = req.param("prenom");
    if (id) {
        // modification
        Client.findByIdAndUpdate(id, { nom : nom, prenom : prenom }, function(err,
client) {
            res.redirect("/clients");
        });
    }
    else {
        // création
        Client.create({ nom : nom, prenom : prenom }, function(err, client) {
            res.redirect("/clients");
        });
    }
});

// création d'un client
app.get("/clients/new", function(req, res) {
    res.render("clients/new.ejs");
});

http.createServer(app).listen(app.get('port'), function(){
    console.log('Express server listening on port ' + app.get('port'));
});
```

Les routes sont écrites dans des méthodes `app.get()`, mais on pourrait aussi utiliser `app.post()` pour certaines d'entre elles qui servent à valider des données dans la base de données.

Remarquez la route `GET /clients/valider/:id?`. L'identifiant `:id` est ici facultatif (grâce au `"?"`) car cette route est utilisée dans le cas d'une création de client (il n'a pas encore d'identifiant) ou dans le cas d'une mise à jour (il en a déjà un).

Les fichiers associés aux vues sont les suivants. Ils sont similaires à ceux déjà écrits précédemment, avec des différences liées aux nouvelles routes utilisées.

Fichier clients.ejs (dans views/clients)

```
<link rel="stylesheet" href="/stylesheets/style.css" />

<h1> Liste des clients </h1>
<% if (!clients.length) { %>
  <b> Pas de clients ! </b>
  <br><br>
<% } else { %>
  <table>
    <% clients.forEach(function(client) { %>
      <tr>
        <td width=30><%= client.id %></td>
        <td width=100><%= client.nom %></td>
        <td width=100><%= client.prenom %></td>
        <td>
          <form style="display:inline" action="/clients/voir/<%= client.id %>">
            <input type="submit" value="Voir" />
          </form>
        </td>
        <td>
          <form style="display:inline" action="/clients/modifier/<%= client.id %>">
            <input type="submit" value="Modifier" />
          </form>
        </td>
        <td>
          <form style="display:inline" action="/clients/supprimer/<%= client.id %>">
            <input type="submit" value="Supprimer" />
          </form>
        </td>
      </tr>
    <% }); %>
  </table>
<% } %>

<form action="/clients/new">
  <input type="submit" value="Créer un client" />
</form>
```

Fichier new.ejs (dans views/clients)

```
<h1> Création de client <h1>

<form action="/clients/valider">

<table>
<tr>
<td>Nom</td>
<td><input type="text" name="nom" /></td>
</tr>
<tr>
<td>Prénom</td>
<td><input type="text" name="prenom" /></td>
</tr>
</table>

<br>
<input type="submit" value="Valider" />

</form>
```

Fichier show.ejs (dans views/clients)

```
<h1> Visualisation de client <h1>

<form action="/clients">

<table>
<tr>
<td>Nom</td>
<td><input type="text" name="nom" disabled value = "<%=client.nom %>" /></td>
</tr>
<tr>
<td>Prénom</td>
<td><input type="text" name="prenom" disabled value = "<%=client.prenom %>" /></td>
</tr>
</table>

<br>
<input type="submit" value="Fermer" />

</form>
```

Fichier edit.ejs (dans views/clients)

```
<h1> Modification de client <h1>

<form action="/clients/valider/<%=client.id %>">

  <table>
    <tr>
      <td>Nom</td>
      <td><input type="text" name="nom" value = "<%=client.nom %>" /></td>
    </tr>
    <tr>
      <td>Prénom</td>
      <td><input type="text" name="prenom" value = "<%=client.prenom %>" /></td>
    </tr>
  </table>

  <br>
  <input type="submit" value="Valider" />

</form>
```

QUATRIÈME PARTIE

Quelques modules Node (très) utiles

29

Le module `async`

Le module `async` (pour asynchrone) permet de gérer plus facilement les fonctions de callback utilisées dans les programmes Node. Par exemple :

- attendre que plusieurs fonctions de callback s'exécutent et lorsqu'elles sont toutes terminées, effectuer un traitement final ;
- exécuter dans un ordre séquentiel ou parallèle plusieurs fonctions de callback ;
- exécuter un traitement asynchrone sur un ensemble de données et lorsque ce traitement est terminé, effectuer un traitement final.

Ce module est constitué d'un fichier nommé `async.js`, qui peut être inclus dans un programme Node, mais est également compatible avec les navigateurs côté client. On pourrait donc aussi l'utiliser pour effectuer des actions dans le code JavaScript du navigateur.

Installer le module `async`

Le module `async` s'installe au moyen de la commande `npm install async`. Nous ne traiterons ici que de son utilisation côté serveur, c'est-à-dire dans un programme Node. Son utilisation dans un programme JavaScript côté client est similaire à ce que nous allons découvrir ici.

Lançons la commande `npm install async` depuis un interpréteur de commandes. Le module s'installe dans le répertoire `node_modules` du répertoire courant.

Figure 29-1
Installation du module `async`

```
C:\Users\Eric\Documents\Node.js>npm install async
npm WARN package.json @ No README.md file found!
npm http GET https://registry.npmjs.org/async
npm http 200 https://registry.npmjs.org/async
npm http GET https://registry.npmjs.org/async/-/async-0.7.0.tgz
npm http 200 https://registry.npmjs.org/async/-/async-0.7.0.tgz
async@0.7.0 node_modules/async
```

Les méthodes du module `async` sont divisées en deux parties.

- Les méthodes utilitaires permettant d'effectuer des traitements sur un tableau de données : ce sont les méthodes classiques telles que `each()`, `map()`, `filter()`, etc. Le module `async` y ajoute des particularités dans la gestion de ces méthodes, que l'on découvrira par la suite.
- Les méthodes gérant l'enchaînement des fonctions de callback, de façon à les exécuter en parallèle, en série, etc. Ces méthodes sont très pratiques dans une application Node car elles permettent de s'affranchir de l'imbrication des fonctions de callback dans le programme.

Nous étudierons tout d'abord les méthodes agissant sur les tableaux de données, puis celles agissant sur les fonctions de callback.

Méthodes agissant sur les tableaux de données

Les méthodes qui suivent possèdent toutes en premier paramètre un tableau de données sur lequel un traitement va s'effectuer.

Méthode `each()`

La méthode `async.each(arr, iterator, callback)` permet de parcourir les éléments du tableau `arr` au moyen de la fonction d'itération `iterator(elem, callback)`, puis d'exécuter la fonction de callback finale (indiquée en dernier paramètre de la méthode `async.each()`) lorsque la fonction d'itération est terminée pour tous les éléments du tableau.

Par rapport à une boucle de traitement traditionnelle, l'intérêt ici est que la fonction d'itération peut être une fonction asynchrone (par exemple, un appel à la base de données, ou un timer d'attente...). Même dans ce cas, la fonction de callback indiquée en dernier paramètre ne s'exécutera que si toutes les fonctions d'itération sont terminées ou si l'une d'elle retourne une erreur.

Chaque itération sur le tableau `arr` doit donc produire l'appel à `callback(false)` (si tout va bien) ou `callback(true)` (si une erreur s'est produite). Dès qu'une erreur est rencontrée, la méthode `async.each()` appelle la fonction de callback indiquée en dernier paramètre de `async.each()`, sinon cette fonction de callback n'est appelée que lorsque l'appel `callback(false)` de la dernière itération est effectué. Ceci garantit que la fonction de callback indiquée en dernier paramètre de la méthode `async.each()` est appelée après que toutes les itérations sont faites (sauf en cas d'erreur).

Exemple d'utilisation de la méthode `async.each()`

```
var async = require("async");

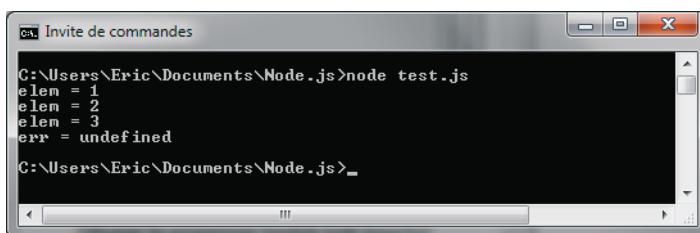
async.each([1, 2, 3], function(elem, callback){
    console.log("elem = " + elem);
    callback(false);
}, function(err){
    console.log("err = " + err);
});
```

Ce programme est la forme d'utilisation la plus simple de la méthode `async.each()`. Le premier paramètre (ici, `[1, 2, 3]`) est le tableau à parcourir. Pour chacun des éléments du tableau, la fonction d'itération passée en second paramètre est appelée. La fonction d'itération possède elle-même deux paramètres.

- Le premier (nommé `elem`) est l'élément du tableau `arr` en cours de traitement (1, puis 2, puis 3).
- Le second (nommé `callback`) correspond à une fonction à appeler obligatoirement pour indiquer si la fonction d'itération s'est bien déroulée (pour chacun des éléments du tableau). Si l'on considère que la fonction d'itération ne s'est pas bien déroulée, on appelle `callback(param)` avec un paramètre `param` dont la valeur est `true` (c'est-à-dire tout sauf `null`, `undefined`, `false`, `0` ou `" "`), sinon le paramètre `param` doit correspondre à une valeur `false` (ou `null`, `undefined`, `0`, `" "`).

Figure 29–2

Utilisation de `async.each()`



```
C:\> node test.js
elem = 1
elem = 2
elem = 3
err = undefined
C:\>
```

On voit sur la précédente exécution que le tableau `[1, 2, 3]` est parcouru, puis la fonction de callback de fin est appelée en dernier avec un code d'erreur qui est `undefined` car tout s'est bien déroulé.

Modifions le programme pour appeler `callback(false)` uniquement si `elem` vaut 1 (premier élément du tableau), et `callback(true)` pour les autres éléments.

Appeler callback(false) et callback(true) dans la fonction d'itération

```
var async = require("async");

async.each([1, 2, 3], function(elem, callback){
  console.log("elem = " + elem);
  if (elem == 1) callback(false);
  else callback(true);
}, function(err){
  console.log("err = " + err);
});
```

Figure 29–3
Erreur lors de l'exécution de
async.each()

```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
err = undefined

C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
err = true
elem = 3

C:\Users\Eric\Documents\Node.js>_
```

On voit maintenant que `callback(true)` étant appelée dès le deuxième élément du tableau, la fonction de callback finale est immédiatement appelée, avec le paramètre `err` valant `true` (la même valeur que celle qui a provoqué l'erreur).

Utilisons maintenant une fonction d'itération asynchrone, c'est-à-dire qui diffère son résultat. On utilise pour cela un timer.

Utiliser un timer dans la fonction d'itération

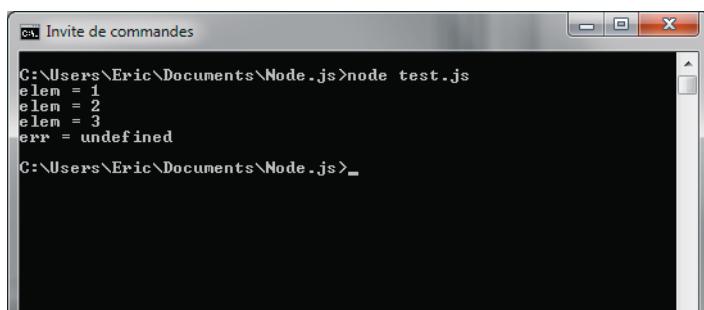
```
var async = require("async");

async.each([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    callback(false);
  }, elem * 100);
}, function(err){
  console.log("err = " + err);
});
```

Le résultat de la fonction d'itération indiqué par `callback(false)` est maintenant différé de 100 millisecondes pour le premier élément du tableau, de 200 pour le deuxième et de 300 pour le troisième. La méthode `async.each()` gère l'attente des résultats, et lorsque tous les résultats sont obtenus, elle appelle la fonction de callback finale.

Figure 29–4

Utilisation de `async.each()` avec une fonction d'itération asynchrone



```
C:\ Invité de commandes
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
err = undefined
C:\Users\Eric\Documents\Node.js>-
```

Que se passerait-il si les fonctions de callback avaient des timers inversement proportionnels à l'ordre des éléments du tableau, c'est-à-dire 300 millisecondes pour le troisième élément, 200 pour le deuxième et 100 pour le premier ?

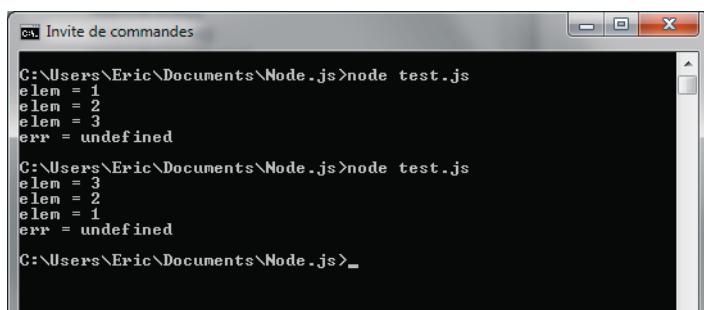
Inverser l'ordre des timers dans la fonction d'itération

```
var async = require("async");

async.each([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    callback(false);
  }, (4 - elem) * 100); // d'abord 300, puis 200, puis 100 millisecondes
}, function(err){
  console.log("err = " + err);
});
```

Figure 29–5

La fonction d'itération a une durée différente selon les éléments du tableau qui l'utilise.



```
C:\ Invité de commandes
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
err = undefined
C:\Users\Eric\Documents\Node.js>-
```

On obtient maintenant un ordre d'appel inversé (3, 2, 1 au lieu de 1, 2, 3), ce qui est normal car le timer du troisième et dernier élément échoit au bout de 100 millisecondes, tandis que le deuxième survient au bout de 200 et le premier au bout de 300.

La question est alors : « Comment conserver l'ordre d'appel des éléments même si les fonctions d'itération ont des durées différentes ? ». On fait alors appel à la méthode `async.eachSeries()`.

Méthode eachSeries()

À la différence de la méthode `async.each()`, la méthode `async.eachSeries(arr, iterator, callback)` garantit que la fonction d'itération sera appelée dans le même ordre que celui des éléments du tableau `arr`. La fonction d'itération d'un élément n'est exécutée que si la fonction d'itération de l'élément précédent est terminée.

Prenons le même exemple que le précédent pour lequel nous avons remarqué que les fonctions d'itération étaient exécutées dans l'ordre inverse des éléments du tableau. Utilisons la méthode `async.eachSeries()` au lieu de `async.each()`.

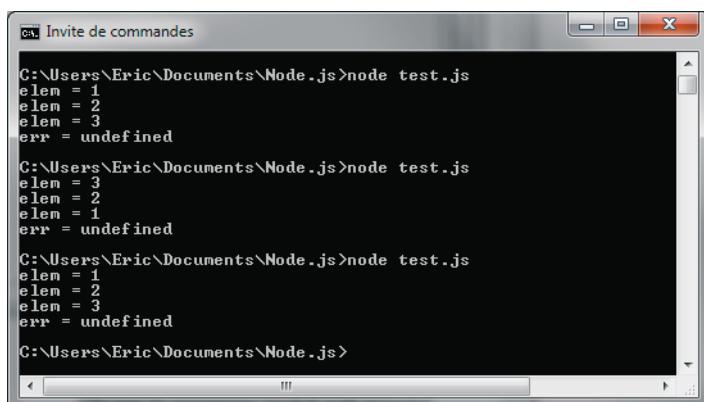
Utiliser `async.eachSeries()`

```
var async = require("async");

async.eachSeries([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    callback(false);
  }, (4 - elem) * 100);
}, function(err){
  console.log("err = " + err);
});
```

Figure 29–6

Conservation de l'ordre d'appels des fonctions d'itération malgré des durées différentes pour chacune



On constate maintenant que l'ordre d'exécution des fonctions d'itération est conservé. Une fonction d'itération n'est exécutée que si la précédente est terminée.

Méthode `map()`

La méthode `async.map(arr, iterator, callback)` fonctionne sur le même principe que la méthode `async.each()` vue précédemment. Elle permet à la fonction d'itération `iterator(elem, callback)` de retourner un nouveau tableau construit à partir du tableau `arr`. La fonction de callback passée en paramètres de la fonction `iterator(elem, callback)` est de la forme `callback(err, transformed)` dans laquelle `err` est une erreur éventuelle (`false` si aucune) et `transformed` est le résultat que l'on souhaite obtenir à la place de l'élément `elem` en cours d'itération.

La fonction de callback finale (indiquée en dernier paramètre de la méthode `async.map()`) est appelée lorsque toutes les itérations sont terminées, ou lorsque l'une d'elles retourne une erreur. Cette fonction de callback est de la forme `callback(err, results)` dans laquelle `err` est l'erreur éventuelle (`undefined` si aucune) et `results` est le nouveau tableau issu de la transformation.

Comme pour la fonction `async.each()`, utilisons la forme la plus simple de la méthode `async.map()` pour, par exemple, retourner un tableau dont les valeurs sont le double des valeurs du tableau d'origine.

Utiliser la méthode `async.map()`

```
var async = require("async");

async.map([1, 2, 3], function(elem, callback){
  console.log("elem = " + elem);
  callback(false, 2*elem);
}, function(err, results){
  console.log("results = " + results);
});
```

La nouvelle valeur renournée dans le tableau (ici, `2*elem`) est indiquée en second paramètre lors de l'appel à la fonction `callback(err, transformed)`.

Figure 29–7
Le tableau d'origine a été transformé.

```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
results = 2,4,6
C:\Users\Eric\Documents\Node.js>
```

Le tableau retourné en résultat contient maintenant le double des valeurs d'origine.

Utilisons la méthode `async.map()` en produisant une erreur lors de l'appel de la fonction de callback dans la méthode d'itération. Par exemple, on produit une erreur sur le premier élément transmis.

Produire une erreur dans la fonction d'itération

```
var async = require("async");

async.map([1, 2, 3], function(elem, callback){
  console.log("elem = " + elem);
  if (elem == 1) callback(true, 2*elem);
  else callback(false, 2*elem);
}, function(err, results){
  console.log("results = " + results);
  console.log("err = " + err);
});
```

Figure 29–8
Une fonction d'itération produit une erreur, le résultat est incomplet.

```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
results = 2
err = true
elem = 2
elem = 3
C:\Users\Eric\Documents\Node.js>
```

Le premier élément du tableau est traité, mais comme la fonction d'itération retourne une erreur, les autres éléments ne sont pas incorporés au tableau final `results`.

Comme pour la méthode `async.each()`, l'intérêt de la méthode `async.map()` est d'utiliser une fonction d'itération asynchrone, c'est-à-dire qui ne délivre pas son résultat immédiatement. Pour cela, insérons un timer dans le code de la fonction d'itération.

Le premier appel (pour le premier élément) sera de 100 millisecondes, le deuxième appel (pour le deuxième élément du tableau) sera de 200 millisecondes, et de 300 pour le troisième élément.

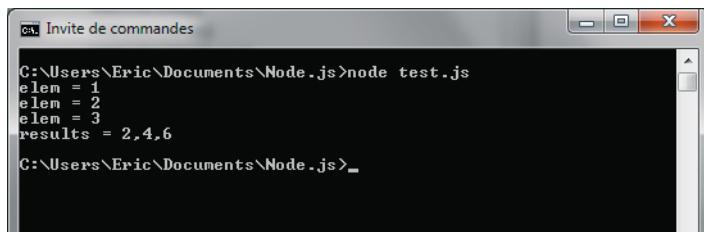
Utiliser un timer dans la fonction d'itération

```
var async = require("async");

async.map([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    callback(false, 2*elem);
  }, elem * 100);
}, function(err, results){
  console.log("results = " + results);
});
```

Figure 29–9

La fonction de callback finale est appelée lorsque toutes les fonctions d'itération sont terminées.



Bien que la fonction d'itération rende ses résultats de manière asynchrone, la fonction de callback finale est appelée à l'issue de tous les traitements et restitue le nouveau tableau résultat.

Inversons maintenant la durée du timer : 300 millisecondes pour le premier élément, 200 pour le deuxième et 100 pour le troisième.

Utiliser un timer décroissant pour la fonction d'itération

```
var async = require("async");

async.map([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    callback(false, 2*elem);
  }, (4 - elem) * 100);
}, function(err, results){
  console.log("results = " + results);
});
```

Figure 29–10

Les fonctions d’itération sont appelées dans l’ordre d’échéance des timers.

```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
results = 2,4,6

C:\Users\Eric\Documents\Node.js>node test.js
elem = 3
elem = 2
elem = 1
results = 2,4,6
C:\Users\Eric\Documents\Node.js>_
```

Les éléments du tableau sont traités dans l’ordre dans lequel le timer arrive à échéance : d’abord le troisième (de 100 millisecondes), puis le deuxième (de 200 millisecondes), puis le premier (de 300 millisecondes). En revanche, les éléments restitués dans le tableau final sont dans le bon ordre, indépendamment de la vitesse à laquelle la fonction d’itération les a restitués.

Utilisons la méthode `async.map()` pour rechercher des informations sur un ensemble de fichiers. On utilise sur chacun des fichiers la méthode `fs.stat()` du module `fs`. Cette méthode est asynchrone et elle est de la même forme que la fonction d’itération utilisée précédemment. On indique ci-après le descriptif de cette méthode (déjà décrite au chapitre 6, « Gestion des fichiers »).

Tableau 29–1 Méthode d’obtention d’informations sur un fichier ou un répertoire

Méthode	Signification
<code>fs.stat(path, callback)</code>	Analyse le chemin indiqué (fichier ou répertoire) et appelle la fonction de callback de la forme <code>function(err, stat)</code> dans laquelle <code>stat</code> est un objet contenant les propriétés ou méthodes suivantes : <ul style="list-style-type: none"> - <code>stat.size</code> : taille en octets du fichier ; - <code>stat.ctime</code> : date de création ; - <code>stat.mtime</code> : date de dernière modification ; - <code>stat.isFile()</code> : retourne <code>true</code> si le path représente un fichier, sinon <code>false</code> ; - <code>stat.isDirectory()</code> : retourne <code>true</code> si le path représente un répertoire, sinon <code>false</code>.

La fonction de callback utilisée dans la méthode `fs.stat()` est de la forme `callback(err, stat)`, c’est-à-dire de la même forme que celle utilisée dans la fonction d’itération de `async.map(arr, iterator, callback)`. Le code suivant fournit donc les renseignements sur les fichiers transmis dans le tableau `arr` dans la fonction de callback finale.

Utiliser `fs.stat()` en tant que fonction d'itération dans `async.map()`

```
var async = require("async");
var fs = require("fs");
var util = require("util");

async.map(["test.js", "client.js", "server.js"], fs.stat, function(err,
results){
  results.forEach(function(result) {
    console.log("\nresult = " + JSON.stringify(result));
  });
});
```

Figure 29-11

La fonction d'itération est `fs.stat()`.

```
C:\Users\Eric\Documents\Node.js>node test.js
result = {"dev":0,"mode":33206,"nlink":1,"uid":0,"gid":0,"rdev":0:"1131,"atime":"2013-04-09T15:54:56.000Z","mtime":"2014-04-18T13:me":"2013-04-09T15:54:56.000Z"}

result = {"dev":0,"mode":33206,"nlink":1,"uid":0,"gid":0,"rdev":0:"617,"atime":"2013-06-24T11:41:16.000Z","mtime":"2014-04-15T08:0e":"2013-06-21T14:49:42.000Z"}

result = {"dev":0,"mode":33206,"nlink":1,"uid":0,"gid":0,"rdev":0:"741,"atime":"2013-06-24T11:43:54.000Z","mtime":"2014-04-06T12:2e":"2013-06-24T11:43:54.000Z"}

C:\Users\Eric\Documents\Node.js>
```

Dans chaque élément du tableau `results`, on trouve le descriptif du fichier auquel l'élément correspond.

Méthode `mapSeries()`

La méthode `async.mapSeries(arr, iterator, callback)` est une variante de la méthode `async.map()` précédente. En effet, elle permet de s'assurer que la fonction d'itération est appelée pour un élément du tableau uniquement si la précédente (appelée pour l'élément précédent du tableau) est terminée.

Utilisons le programme écrit précédemment et remplaçons la méthode `map()` par `mapSeries()`.

Utiliser la méthode `async.mapSeries()`

```
var async = require("async");

async.mapSeries([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
```

```

        callback(false, 2*elem);
    }, (4 - elem) * 100);
}, function(err, results){
    console.log("results = " + results);
});

```

Les fonctions d’itération rendent leur résultat dans l’ordre inverse de leur appel. Mais grâce à la méthode `async.mapSeries()`, elles sont séquencées dans l’ordre des éléments du tableau, comme on peut le voir sur la figure 29-12.

Figure 29-12

Les fonctions d’itération sont appelées dans l’ordre des éléments du tableau.

```

C:\Invite de commandes
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
results = 2,4,6
C:\Users\Eric\Documents\Node.js>_

```

Méthode filter()

La méthode `async.filter(arr, iterator, callback)` permet de filtrer les éléments du tableau `arr`, en indiquant les éléments finaux que l’on souhaite conserver ou non. Pour cela, la fonction d’itération est de la forme `iterator(elem, callback)`, dans laquelle l’appel à `callback(truthValue)` permet d’indiquer si l’élément est retenu (`truthValue` vaut `true`) ou rejeté (`truthValue` vaut `false`) dans le tableau final.

La fonction de callback finale est appelée lorsque la fonction d’itération a terminé de s’exécuter pour chacun des éléments du tableau, sans possibilité de l’activer avant la fin (ce qui diffère des méthodes de type `each()` et `map()` vues précédemment). Cette fonction de callback est de la forme `function callback(results)` dans laquelle `results` est le tableau résultat.

Utiliser la méthode `async.filter()` pour retourner les éléments du tableau supérieurs ou égaux à 2

```

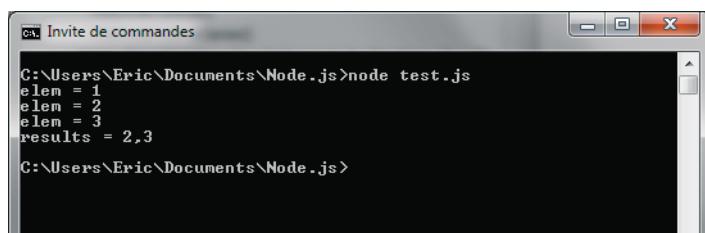
var async = require("async");

async.filter([1, 2, 3], function(elem, callback){
    console.log("elem = " + elem);
    if (elem >= 2) callback(true); // accepter l'élément
    else callback(false); // refuser l'élément
}, function(results){
    console.log("results = " + results);
});

```

Figure 29–13

Le tableau résultat ne contient plus que les éléments supérieurs ou égaux à 2.



The screenshot shows a terminal window titled "Invite de commandes". The command run is "node test.js". The output is:
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
results = 2,3
C:\Users\Eric\Documents\Node.js>

Seuls les éléments supérieurs ou égaux à 2 sont retournés dans le tableau final.

On peut gérer la fonction d’itération de manière asynchrone, comme pour les méthodes précédentes.

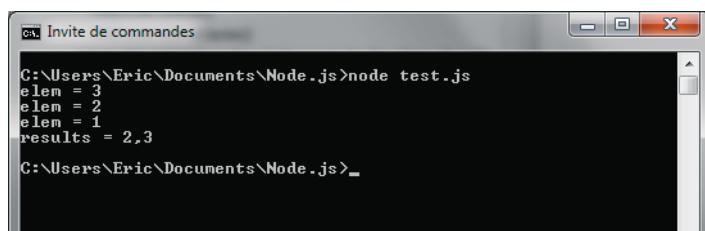
Utiliser la méthode `async.filter()` avec une fonction d’itération asynchrone

```
var async = require("async");

async.filter([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    if (elem >= 2) callback(true);
    else callback(false);
  }, (4 - elem) * 100)
}, function(results){
  console.log("results = " + results);
});
```

Figure 29–14

Le tableau a été filtré, mais dans l’ordre d’échéance des timers.



The screenshot shows a terminal window titled "Invite de commandes". The command run is "node test.js". The output is:
C:\Users\Eric\Documents\Node.js>node test.js
elem = 3
elem = 2
elem = 1
results = 2,3
C:\Users\Eric\Documents\Node.js>

Du fait de l’ordre des timers, les fonctions d’itération se sont déclenchées dans l’ordre des éléments du tableau, mais le résultat a été obtenu en sens inverse. En revanche, le tableau final associé au paramètre `results` est mis dans l’ordre des éléments du tableau.

Utilisons la méthode `async.filter()` pour retourner, à partir d’un ensemble de noms de fichiers, un tableau des fichiers existants sur le serveur. On utilisera la fonction d’itération `fs.exists()` dont la description est reprise dans le tableau 29–2.

Tableau 29–2 Méthode de test de l'existence d'un fichier ou d'un répertoire

Méthode	Signification
<code>fs.exists(path, callback)</code>	Teste l'existence du fichier ou du répertoire indiqué, et appelle la fonction de callback de la forme <code>function(exists)</code> dans laquelle <code>exists</code> vaut <code>true</code> si le fichier existe, sinon <code>false</code> . Attention : la fonction de callback ne possède pas le paramètre <code>err</code> traditionnel.

La fonction de callback indiquée dans la méthode `fs.exists()` est de la même forme que celle attendue dans la fonction d'itération de la méthode `async.filter()`. Le programme est le suivant.

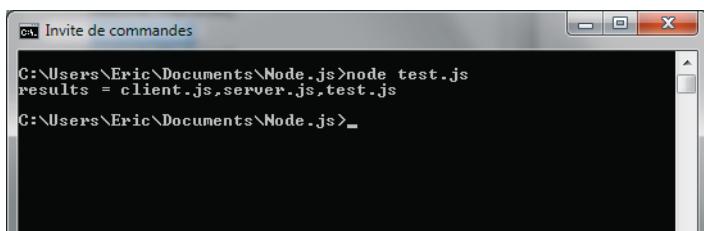
Filtrer les noms de fichiers existant sur le serveur

```
var async = require("async");
var fs = require("fs");

async.filter(["client.js", "client2.js", "server.js", "test.js", "server2.js"],
  fs.exists, function(results){
  console.log("results = " + results);
});
```

Figure 29–15

Seuls les fichiers existant sur le serveur sont retournés en résultat.



Parmi les cinq fichiers indiqués en entrée de la méthode `async.filter()`, seuls les trois fichiers retournés en résultat existent sur le serveur.

Méthode `filterSeries()`

La méthode `async.filterSeries(arr, iterator, callback)` est similaire à la méthode `async.filter(arr, iterator, callback)` précédente, avec en plus l'assurance que les appels à la fonction d'itération ne sont effectués que si le précédent est terminé.

Utilisons cette méthode pour montrer que l'enchaînement des fonctions d'itération s'effectue dans un ordre séquentiel (une itération ne peut pas commencer si la précédente n'est pas complètement terminée).

Utiliser `async.filterSeries()`

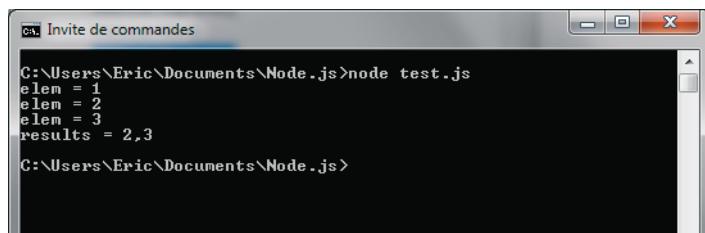
```
var async = require("async");

async.filterSeries([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    if (elem >= 2) callback(true);
    else callback(false);
  }, (4 - elem) * 100)
}, function(results){
  console.log("results = " + results);
});
```

Le timer est positionné dans l'ordre décroissant, afin que le premier élément du tableau soit l'élément pour lequel l'attente est la plus longue. Toutefois, le second élément est mis en attente de la fin du premier avant d'exécuter sa fonction d'itération.

Figure 29-16

Le tableau a été filtré,
mais dans l'ordre des éléments
dans le tableau.



```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
results = 2,3
C:\Users\Eric\Documents\Node.js>
```

On retrouve maintenant le même ordre que celui des éléments dans le tableau d'origine.

Méthode `reject()`

La méthode `async.filter(arr, iterator, callback)` avait permis de retourner tous les éléments qui satisfont une condition. À l'inverse, la méthode `async.reject(arr, iterator, callback)` va retourner tous les éléments qui ne satisfont pas la condition. Hormis cette différence, l'utilisation des deux méthodes est similaire.

Utilisons cette méthode pour retourner les fichiers qui n'existent pas sur le serveur. On utilise la fonction d'itération `fs.existsSync()` comme précédemment.

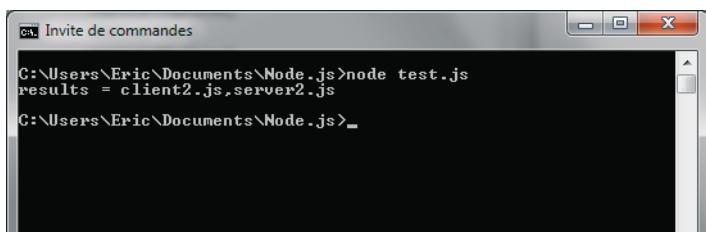
Utiliser la méthode `async.reject()` pour retourner les fichiers qui n'existent pas sur le serveur

```
var async = require("async");
var fs = require("fs");

async.reject(["client.js", "client2.js", "server.js", "test.js", "server2.js"],
  fs.exists, function(results){
  console.log("results = " + results);
});
```

Figure 29-17

Seuls les fichiers inexistant sur le serveur sont retournés en résultat.



Les fichiers retournés sont ceux qui n'existent pas sur le serveur.

Méthode `rejectSeries()`

Comme pour la méthode `async.filterSeries(arr, iterator, callback)`, la méthode `async.rejectSeries(arr, iterator, callback)` permet que la fonction d'itération pour un élément du tableau ne soit appelée que si l'appel pour l'élément précédent est terminé.

Utiliser la méthode `async.rejectSeries()`

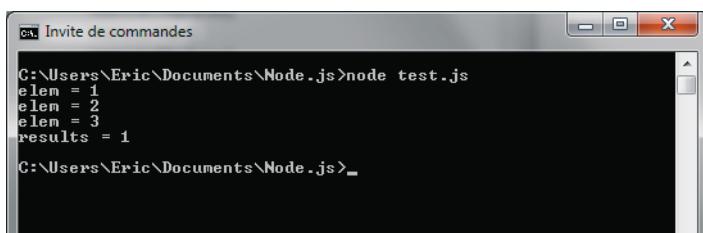
```
var async = require("async");

async.rejectSeries([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    if (elem >= 2) callback(true);
    else callback(false);
  }, (4 - elem) * 100)
}, function(results){
  console.log("results = " + results);
});
```

Le programme est le même que celui utilisé pour la méthode `async.filterSeries()`, mais les éléments retournés dans le tableau sont ceux qui étaient précédemment rejettés.

Figure 29-18

Les éléments sont filtrés selon leur ordre dans le tableau.



```
c:\ Invité de commandes
C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
elem = 2
elem = 3
results = 1
C:\Users\Eric\Documents\Node.js>
```

Méthode `detect()`

La méthode `async.detect(arr, iterator, callback)` permet de trouver un élément du tableau pour lequel la fonction d’itération de la forme `iterator(elem, callback)` effectue un appel à `callback(true)`.

La fonction de callback finale est appelée lorsqu’une fonction d’itération retourne un élément. Elle est de la forme `callback(result)`, `result` étant un élément du tableau (si trouvé), ou `undefined` (si non trouvé).

Utilisons la méthode `async.detect()` afin de trouver un élément du tableau `[1, 2, 3]` dont la valeur est supérieure ou égale à 1. Les trois éléments du tableau répondent à ce critère, mais `async.detect()` retourne le premier trouvé pour lequel la valeur correspond. Le premier trouvé n’est pas forcément le premier qui correspond dans la liste, étant donné que la fonction d’itération peut comporter un caractère asynchrone, comme dans l’exemple suivant.

Utiliser la méthode `async.detect()` afin de trouver un élément supérieur ou égal à 1

```
var async = require("async");

async.detect([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    if (elem >= 1) callback(true);
    else callback(false);
  }, (4 - elem) * 100)
}, function(result){
  console.log("result = " + result);
});
```

Les fonctions d’itération sont exécutées en parallèle, et la première qui effectue un appel à `callback(true)` déclenche la fonction de callback finale en lui transmettant le résultat.

Figure 29-19

La première fonction d'itération qui trouve le résultat le transmet à la fonction de callback finale.

```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 3
result = 3
elem = 2
elem = 1
C:\Users\Eric\Documents\Node.js>
```

Le troisième élément du tableau est celui pour lequel la fonction d'itération retourne le plus vite le résultat, car elle est lancée avant les autres (au bout de 100 millisecondes, au lieu de 200 et 300 pour les autres). Même si d'autres éléments de la liste satisfont le critère, ce n'est pas le premier dans la liste qui est choisi. Pour que ce premier élément soit celui qui sera sélectionné, il faut utiliser la méthode `async.detectSeries()`.

Méthode `detectSeries()`

La méthode `async.detectSeries(arr, iterator, callback)` fonctionne sur le même principe que `async.detect(arr, iterator, callback)`, mais assure qu'une fonction d'itération ne sera lancée sur un élément du tableau que si la précédente (lancée pour l'élément précédent) est terminée.

Cela permet de trouver le premier élément de la liste qui satisfait un critère, quelle que soit la durée de la fonction d'itération pour chacun des éléments.

Utilisons cette méthode pour trouver le premier élément supérieur ou égal à 1, selon le même programme que précédemment.

Utiliser `async.detectSeries()` pour trouver le premier élément du tableau supérieur ou égal à 1

```
var async = require("async");

async.detectSeries([1, 2, 3], function(elem, callback){
  setTimeout(function() {
    console.log("elem = " + elem);
    if (elem >= 1) callback(true);
    else callback(false);
  }, (4 - elem) * 100)
}, function(result){
  console.log("result = " + result);
});
```

Figure 29–20

Les fonctions d'itération sont exécutées selon l'ordre des éléments dans le tableau.

```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 3
result = 3
elem = 2
result = 1

C:\Users\Eric\Documents\Node.js>node test.js
elem = 1
result = 1

C:\Users\Eric\Documents\Node.js>
```

Le premier élément de la liste correspond au critère, son résultat est donc envoyé à la fonction de callback finale, tandis que les autres fonctions d'itération (sur les éléments suivants du tableau) ne sont pas exécutées.

Utilisons la méthode `async.detectSeries()` afin de déterminer le premier fichier du tableau qui existe. On utilise pour cela la fonction d'itération `fs.exists()` utilisée précédemment dans la méthode `async.filter()`.

Utiliser `async.detectSeries()` pour déterminer le premier fichier de la liste qui existe

```
var async = require("async");
var fs = require("fs");

async.detectSeries(["client.js", "client2.js", "server.js", "test.js",
  "server2.js"], fs.exists, function(result){
  console.log("result = " + result);
});
```

Figure 29–21

Le premier fichier qui existe est retourné en résultat.

```
C:\Users\Eric\Documents\Node.js>node test.js
result = client.js

C:\Users\Eric\Documents\Node.js>
```

Méthode `sortBy()`

La méthode `async.sortBy(arr, iterator, callback)` permet de trier les éléments du tableau selon la fonction d'itération transmise en paramètre. Cette fonction d'itération est de la forme `iterator(elem, callback)`, pour laquelle :

- le paramètre `elem` est un élément du tableau passé en paramètre ;

- le paramètre `callback` est de la forme `callback(err, sortValue)`. Le paramètre `sortValue` sert à trier les éléments du tableau en leur affectant un rang symbolisé par `sortValue`. Le paramètre `err` doit valoir `false` (ou une valeur équivalente) sinon la fonction de callback finale est immédiatement appelée et le tri n'est pas effectué.

Lorsque la fonction d'itération a été appelée pour tous les éléments du tableau, la fonction de callback finale est appelée. Elle est de la forme `function(err, results)` dans laquelle `results` est le tableau trié.

Utilisons la méthode `async.sortBy()` pour trier un tableau de valeurs selon les valeurs croissantes.

Trie par ordre croissant

```
var async = require("async");
var fs = require("fs");

async.sortBy([6, 5, 10, 2, 3, 7], function(elem, callback) {
  console.log("elem = " + elem);
  callback(false, elem);
}, function(err, results){
  console.log("results = " + results);
});
```

Le paramètre `sortValue` transmis à la fonction `callback(err, sortValue)` est positionné à la valeur de l'élément dans le tableau, ce qui permet de trier le tableau selon les valeurs croissantes.

Figure 29–22

Le tableau a été trié selon l'ordre des valeurs croissantes.

```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 6
elem = 5
elem = 10
elem = 2
elem = 3
elem = 7
results = 2,3,5,6,7,10
C:\Users\Eric\Documents\Node.js>
```

Trions maintenant le tableau par valeur décroissante.

Tri par ordre décroissant

```
var async = require("async");
var fs = require("fs");
```

```
async.sortBy([6, 5, 10, 2, 3, 7], function(elem, callback) {
  console.log("elem = " + elem);
  callback(false, elem * (-1));
}, function(err, results){
  console.log("results = " + results);
});
```

La fonction `callback(err, sortValue)` transmet maintenant `elem * (-1)` pour le paramètre `sortValue`, ce qui permet de changer le signe de la valeur transmise, et ainsi d'inverser le sens du tri.

Figure 29–23

Le tableau a été trié selon l'ordre des valeurs décroissantes.

```
C:\Users\Eric\Documents\Node.js>node test.js
elem = 6
elem = 5
elem = 10
elem = 2
elem = 3
elem = 7
results = 2,3,5,6,7,10

C:\Users\Eric\Documents\Node.js>node test.js
elem = 6
elem = 5
elem = 10
elem = 2
elem = 3
elem = 7
results = 10,7,6,5,3,2
C:\Users\Eric\Documents\Node.js>
```

Utilisons maintenant la méthode `async.sortBy()` afin de trier un ensemble de fichiers selon la date de leur dernière modification.

Trier un ensemble de fichiers selon la date de dernière modification

```
var async = require("async");
var fs = require("fs");

async.sortBy(["client.js", "server.js", "test.js"], function(elem, callback) {
  fs.stat(elem, function(err, stat) {
    console.log("elem = " + elem);
    callback(null, stat.mtime);
  });
}, function(err, results){
  console.log("results = " + results);
});
```

Figure 29–24

Les fichiers ont été triés selon leur date de dernière modification.

```
C:\Users\Eric\Documents\Node.js>node test.js
client.js
server.js
test.js
results = server.js,client.js,test.js
C:\Users\Eric\Documents\Node.js>
```

Le tableau `results` est ordonné selon la date de modification des fichiers, de la plus ancienne à la plus récente.

Méthode `some()`

La méthode `async.some(arr, iterator, callback)` permet de vérifier qu'une condition exprimée dans la fonction d'itération est vraie pour au moins un des éléments du tableau. Si c'est le cas, la fonction de callback finale, de la forme `function callback(result)` est appelée avec un argument `result` positionné à `true`, sinon l'argument `result` est positionné à `false`.

La fonction d'itération est de la forme `iterator(elem, callback)`, dans laquelle :

- le paramètre `elem` est l'élément du tableau en cours d'analyse ;
- le paramètre `callback` est une fonction de callback de la forme `callback(truthValue)`, à appeler pour indiquer si l'élément satisfait les conditions de recherche. Si c'est le cas, on utilise `callback(true)`, sinon on utilise `callback(false)`.

La fonction de callback finale, de la forme `function callback(result)` est appelée dès qu'une fonction d'itération utilise `callback(true)`, ou lorsque tous les appels aux fonctions d'itération effectuent `callback(false)`. Le paramètre `result` indique si la condition a été satisfaite pour au moins un élément du tableau (`result` vaut `true`), ou aucun (`result` vaut `false`).

Utilisons la méthode `async.some()` afin de vérifier que la valeur 2 se trouve dans le tableau `[1, 2, 3]` transmis en paramètre.

Vérifier que le tableau `[1, 2, 3]` contient la valeur 2

```
var async = require("async");

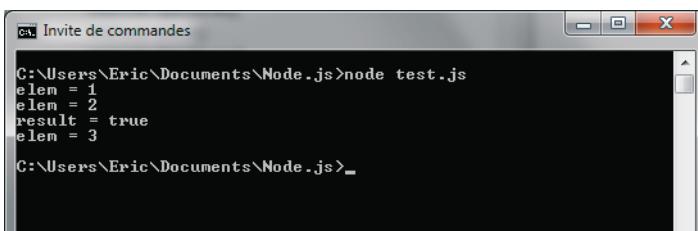
async.some([1, 2, 3], function(elem, callback) {
  console.log("elem = " + elem);
  if (elem == 2) callback(true);
  else callback(false);
```

```
}, function(result){  
    console.log("result = " + result);  
});
```

Pour chaque élément du tableau, on appelle `callback(true)` ou `callback(false)` selon que l'élément satisfait ou non la condition.

Figure 29–25

La valeur 2 est bien présente dans le tableau.



```
c:\ Invité de commandes  
C:\Users\Eric\Documents\Node.js>node test.js  
elem = 1  
elem = 2  
result = true  
elem = 3  
C:\Users\Eric\Documents\Node.js>
```

Dès que la condition est satisfaite (pour l'élément 2 du tableau), la fonction de callback finale est appelée avec la valeur `true` en paramètre.

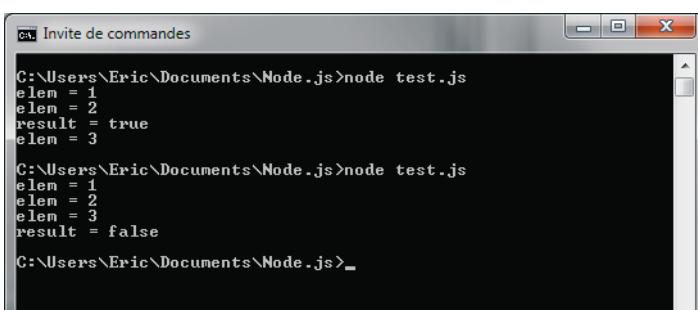
Supposons que la condition exprimée dans la fonction d'itération ne soit jamais satisfaite. Par exemple, on recherche si au moins un élément du tableau est supérieur à 10.

Rechercher si un élément du tableau est supérieur à 10

```
var async = require("async");  
  
async.some([1, 2, 3], function(elem, callback) {  
    console.log("elem = " + elem);  
    if (elem > 10) callback(true);  
    else callback(false);  
}, function(result){  
    console.log("result = " + result);  
});
```

Figure 29–26

Aucun élément du tableau n'est supérieur à 10.



```
c:\ Invité de commandes  
C:\Users\Eric\Documents\Node.js>node test.js  
elem = 1  
elem = 2  
result = true  
elem = 3  
C:\Users\Eric\Documents\Node.js>  
elem = 1  
elem = 2  
elem = 3  
result = false  
C:\Users\Eric\Documents\Node.js>
```

Ce n'est que lorsque tous les appels à la fonction d'itération ont exécuté `callback(false)` que la fonction de callback finale est appelée avec `false` en paramètre.

Méthode every()

La méthode `async.every(arr, iterator, callback)` permet de vérifier que tous les éléments du tableau satisfont la condition exprimée dans la fonction d'itération. Si c'est le cas, la fonction de callback finale, de la forme `function callback(result)` est appelée avec un argument `result` positionné à `true`, sinon l'argument `result` est positionné à `false`.

La fonction d'itération est de la forme `iterator(elem, callback)`, dans laquelle :

- le paramètre `elem` est l'élément du tableau en cours d'analyse ;
- le paramètre `callback` est une fonction de callback de la forme `callback(truthValue)`, à appeler pour indiquer si l'élément satisfait les conditions de recherche. Si c'est le cas, on utilise `callback(true)`, sinon on utilise `callback(false)`.

La fonction de callback finale, de la forme `function callback(result)` est appelée lorsque tous les appels à la fonction d'itération utilisent `callback(true)`, ou lorsque l'un d'eux effectue `callback(false)`. Le paramètre `result` indique si la condition a été satisfaite pour tous les éléments du tableau (`result` vaut `true`), ou si l'un d'eux au moins ne la satisfait pas (`result` vaut `false`).

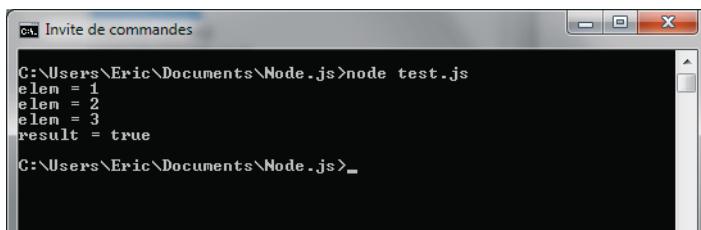
Vérifier que tous les éléments du tableau sont positifs

```
var async = require("async");

async.every([1, 2, 3], function(elem, callback) {
  console.log("elem = " + elem);
  if (elem > 0) callback(true);
  else callback(false);
}, function(result){
  console.log("result = " + result);
});
```

Figure 29-27

Tous les éléments du tableau sont positifs.



Utilisons la méthode `async.every()` afin de vérifier que les fichiers transmis dans le tableau en paramètre existent sur le serveur.

Vérifier que les fichiers indiqués existent sur le serveur

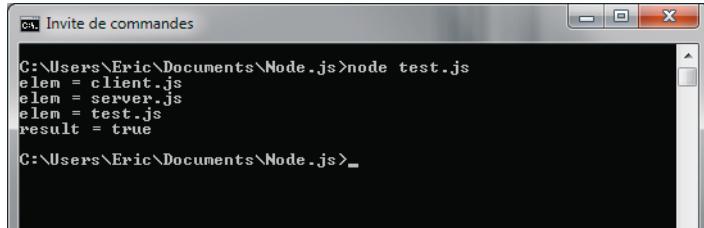
```
var async = require("async");
var fs = require("fs");

async.every(["client.js", "server.js", "test.js"], function(elem, callback) {
  console.log("elem = " + elem);
  fs.exists(elem, function(exists) {
    callback(exists);
  })
}, function(result){
  console.log("result = " + result);
});
```

La méthode asynchrone `fs.exists()` est appelée dans la fonction d'itération afin de vérifier si le fichier existe ou non. Le résultat `exists` (`true` ou `false`) est ensuite transmis par `callback(exists)`.

Figure 29–28

Tous les fichiers indiqués dans le tableau existent sur le serveur.



Une forme condensée du programme précédent peut également s'écrire de la façon suivante.

Tester l'existence des fichiers transmis en paramètres

```
var async = require("async");
var fs = require("fs");

async.every(["client.js", "server.js", "test.js"], fs.exists, function(result){
  console.log("result = " + result);
});
```

Méthodes agissant sur l'enchaînement des fonctions de callback

Les méthodes décrites ci-après servent à gérer l'enchaînement des fonctions de callback dans les programmes. En effet, il est usuel d'imbriquer les traitements de ces fonctions les uns dans les autres, nuisant souvent à la lisibilité du programme. Le module `async` propose un ensemble de méthodes permettant d'améliorer l'écriture de ces programmes, et également d'offrir plus de souplesse dans les traitements des fonctions de callback (par exemple, les exécuter en série, en parallèle, etc.).

Méthode `series()`

La méthode `async.series(tasks, callback)` permet d'exécuter plusieurs fonctions de callback les unes à la suite des autres. Le paramètre `tasks` désigne les fonctions de callback (dans un tableau ou un objet), tandis que le paramètre `callback` est une fonction de callback optionnelle qui est appelée en cas d'erreur ou lorsque toutes les fonctions de callback de la liste `tasks` ont été exécutées.

Chaque fonction de callback, indiquée dans le paramètre `tasks`, est de la forme `function(callback)`, dans laquelle le paramètre `callback` représente la prochaine fonction de callback à appeler à l'aide de l'instruction `callback(err, result)`. Le paramètre `err` représente une erreur éventuelle (`false` ou équivalent si pas d'erreur), et `result` est le résultat que l'on souhaite transmettre pour cette fonction.

La fonction de callback finale, paramètre optionnel de la méthode `async.series()`, est de la forme `function(err, results)`. Elle est appelée dans les cas suivants (lorsqu'elle est indiquée dans les paramètres) :

- si une fonction de callback de la liste `tasks` utilise `callback(true, result)` (signalant ainsi une erreur), cela interrompt le traitement des fonctions de callback suivantes de la liste ;
- si toutes les fonctions de callback de la liste utilisent `callback(false, result)` (ne signalant ainsi aucune erreur), le dernier appel à `callback(false, result)` provoque l'appel de la fonction de callback finale.

Les paramètres `err` et `results` de la fonction de callback finale sont les suivants :

- le paramètre `err` correspond à une erreur transmise lors des appels à `callback(err, result)` dans les fonctions de callback ;
- le paramètre `results` est un tableau des paramètres `result` transmis lors des appels à `callback(err, result)`.

Utilisons la méthode `async.series()` pour exécuter en série les trois fonctions de callback qui lui sont transmises en paramètres.

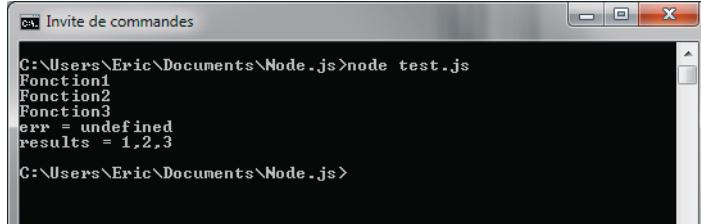
Exécuter trois fonctions de callback en série

```
var async = require("async");

async.series([function(callback) {
    console.log("Fonction1");
    callback(false, 1);          // de la forme callback(err, result)
}, function(callback) {
    console.log("Fonction2");
    callback(false, 2);          // de la forme callback(err, result)
}, function(callback) {
    console.log("Fonction3");
    callback(false, 3);          // de la forme callback(err, result)
}], function(err, results) {
    console.log("err = " + err);
    console.log("results = " + results);
});
```

Chaque fonction de callback utilise le paramètre `callback` pour appeler `callback(err, result)`. Ici, le paramètre `err` est `false` pour indiquer qu'il n'y a pas d'erreur, tandis que le paramètre `result` vaut 1, 2, puis 3 selon la fonction qui l'utilise.

Figure 29–29
Les fonctions de callback s'exécutent en série.



Les fonctions de callback sont exécutées en série, et le retour de chacune d'elles est inséré dans le tableau `results` de la fonction de callback finale.

Produisons une erreur dans une fonction de callback, par exemple la deuxième, en appelant `callback(true, 2)` au lieu de `callback(false, 2)`.

Produire une erreur dans la seconde fonction de callback

```
var async = require("async");

async.series([function(callback) {
    console.log("Fonction1");
    callback(false, 1);          // de la forme callback(err, result)
}, function(callback) {
    console.log("Fonction2");
    callback(true, 2);          // erreur : callback(true, result)
}], function(err, results) {
    console.log("err = " + err);
    console.log("results = " + results);
});
```

```

    callback(true, 2);           // de la forme callback(err, result)
}, function(callback) {
  console.log("Fonction3");
  callback(false, 3);         // de la forme callback(err, result)
}], function(err, results) {
  console.log("err = " + err);
  console.log("results = " + results);
});

```

L'erreur dans la deuxième fonction de callback est simulée par l'appel à `callback(true)`.

Figure 29–30

Une fonction de callback retourne une erreur, ce qui provoque la fin de la série.

```

C:\Users\Eric\Documents\Node.js>node test.js
Fonction1
Fonction2
Fonction3
err = undefined
results = 1,2,3

C:\Users\Eric\Documents\Node.js>node test.js
Fonction1
Fonction2
err = true
results = 1,2
C:\Users\Eric\Documents\Node.js>_

```

Dès que l'erreur est transmise, l'enchaînement des fonctions de callback s'interrompt et la fonction de callback finale est immédiatement appelée avec le tableau des résultats incomplet.

La méthode `async.series()` est intéressante lorsque le traitement dans la fonction de callback est asynchrone, c'est-à-dire que son résultat est différé. Par exemple, si un appel à la base de données est effectué ou si on utilise un timer. Dans ce cas, la fonction de callback n'est pas considérée comme terminée tant que l'appel à l'instruction `callback(err, result)` n'est pas effectué.

Utilisons un timer dans chacune des fonctions de callback afin de montrer que l'exécution différée du traitement ne nuit pas à l'enchaînement des fonctions en série.

Utiliser un timer dans les fonctions de callback pour les exécuter en différé

```

var async = require("async");

async.series([function(callback) {
  setTimeout(function() {
    console.log("Fonction1");
    callback(false, 1);
  }, 300);
}

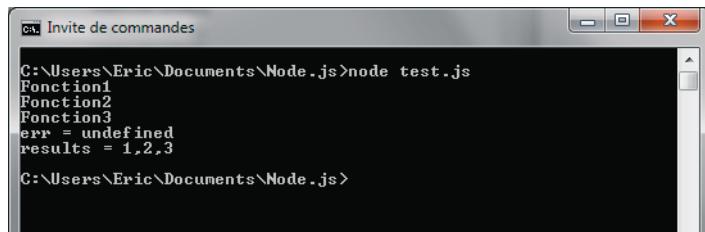
```

```
    }, function(callback) {
      setTimeout(function() {
        console.log("Fonction2");
        callback(false, 2);
      }, 200);
    }, function(callback) {
      setTimeout(function() {
        console.log("Fonction3");
        callback(false, 3);
      }, 100);
  ], function(err, results) {
  console.log("err = " + err);
  console.log("results = " + results);
});
```

La première fonction de callback s'exécute avec un délai de 300 millisecondes, tandis que la deuxième utilise un délai de 200 et la troisième un délai de 100.

Figure 29–31

Les fonctions de callback s'exécutent en série, même si elles sont asynchrones.



Malgré les délais variables positionnés dans les fonctions de callback, elles sont exécutées dans l'ordre indiqué.

Une autre forme d'écriture du paramètre `tasks` est possible. Au lieu de l'écrire sous forme de tableau, on le fait sous forme d'objet JSON. Les fonctions de callback sont enchaînées dans l'ordre où elles sont annoncées dans l'objet `tasks`. Le paramètre `results` indiqué dans la fonction de callback finale n'est alors plus un tableau de valeurs mais un objet contenant ces valeurs.

Écrivons le précédent programme au moyen d'un objet `tasks`, au lieu du tableau précédent.

Utiliser un objet décrivant le paramètre `tasks`

```
var async = require("async");
var util = require("util");

async.series({
```

```

f1 : function(callback) {
  setTimeout(function() {
    console.log("Fonction1");
    callback(false, 1);
  }, 300);
},
f2 : function(callback) {
  setTimeout(function() {
    console.log("Fonction2");
    callback(false, 2);
  }, 200);
},
f3 : function(callback) {
  setTimeout(function() {
    console.log("Fonction3");
    callback(false, 3);
  }, 100);
}
}, function(err, results) {
  console.log("err = " + err);
  console.log("results = " + util.inspect(results));
});

```

Les fonctions de callback sont ici nommées `f1`, `f2` et `f3`. L'objet `results` produit par la fonction de callback finale est affiché au moyen de la méthode `util.inspect()` du module `util`.

Figure 29-32

Le paramètre `tasks` est ici utilisé sous forme d'objet et non de tableau.

```

C:\Users\Eric\Documents\Node.js>node test.js
Fonction1
Fonction2
Fonction3
err = undefined
results = [ f1: 1, f2: 2, f3: 3 ]
C:\Users\Eric\Documents\Node.js>

```

Le résultat est similaire à celui obtenu par l'utilisation du tableau `tasks`. Seul le paramètre `results` de la fonction de callback finale est affiché de façon différente (un objet au lieu d'un tableau).

Méthode `parallel()`

Plutôt que d'exécuter les fonctions de callback en série, on peut décider de les exécuter en parallèle (donc sans attendre que la précédente soit terminée avant d'exécuter la suivante). On utilise pour cela la méthode `async.parallel(tasks, callback)` dans laquelle :

- le paramètre `tasks` représente les fonctions de callback à exécuter, de la forme `function(callback)`. Le paramètre `callback` est utilisé dans la fonction de callback pour transmettre une éventuelle erreur et un résultat. On l'utilise sous la forme `callback(err, result)`. Si aucune erreur, indiquer `false` (ou l'équivalent : `null`, `undefined`, `" "`, `0`) dans le paramètre `err`. Dès qu'une fonction de callback renvoie une erreur, la fonction de callback finale est appelée ;
- le paramètre `callback` représente la fonction de callback finale, appelée lorsqu'une fonction de callback renvoie une erreur ou lorsque toutes les fonctions de callback ont renvoyé leur résultat. Elle est de la forme `function(err, results)` dans laquelle `err` est l'erreur éventuelle transmise par une fonction de callback (`undefined` si pas d'erreur), et `results` est un tableau contenant les résultats de chaque fonction de callback (transmis par `callback(err, result)`).

Utilisons la méthode `async.parallel()` pour exécuter trois fonctions de callback en parallèle. Ces fonctions de callback comportent un timer afin d'avoir un comportement asynchrone dans leur traitement.

Utiliser `async.parallel()` pour exécuter trois fonctions de callback en parallèle

```
var async = require("async");

async.parallel([function(callback) {
    setTimeout(function() {
        console.log("Fonction1");
        callback(false, 1);
    }, 300);
}, function(callback) {
    setTimeout(function() {
        console.log("Fonction2");
        callback(false, 2);
    }, 200);
}, function(callback) {
    setTimeout(function() {
        console.log("Fonction3");
        callback(false, 3);
    }, 100);
}], function(err, results) {
    console.log("err = " + err);
    console.log("results = " + results);
});
```

La première fonction de callback retourne son résultat au bout de 300 millisecondes, la deuxième au bout de 200 et la troisième au bout de 100.

Figure 29–33
Les fonctions de callback s'exécutent en parallèle.

```
C:\Users\Eric\Documents\Node.js>node test.js
Fonction3
Fonction2
Fonction1
err = undefined
results = 1,2,3
C:\Users\Eric\Documents\Node.js>
```

Les fonctions de callback s'exécutant en parallèle, la troisième est la plus rapide à donner son résultat, puis la deuxième et enfin la première. Toutefois, le tableau `results` contient les résultats retournés dans l'ordre d'écriture des fonctions dans le code.

Le paramètre `tasks` peut également être écrit sous forme d'objet JSON.

Utiliser un objet dans le paramètre tasks de `async.parallel(tasks, callback)`

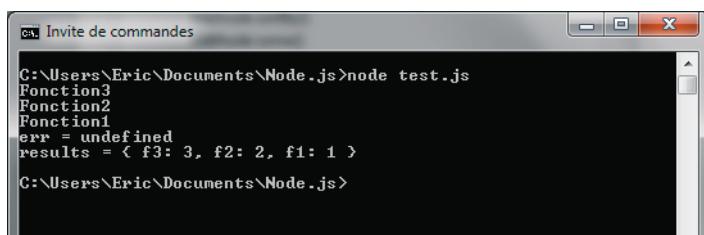
```
var async = require("async");
var util = require("util");

async.parallel({
  f1 : function(callback) {
    setTimeout(function() {
      console.log("Fonction1");
      callback(false, 1);
    }, 300);
  },
  f2 : function(callback) {
    setTimeout(function() {
      console.log("Fonction2");
      callback(false, 2);
    }, 200);
  },
  f3 : function(callback) {
    setTimeout(function() {
      console.log("Fonction3");
      callback(false, 3);
    }, 100);
  }
}, function(err, results) {
  console.log("err = " + err);
  console.log("results = " + util.inspect(results));
});
```

Il s'agit de la même forme d'écriture que pour la méthode `async.series()`.

Figure 29–34

Le paramètre `tasks` est ici utilisé sous forme d'objet.



L'ordre des propriétés dans l'objet `results` n'a pas d'importance, car chaque valeur est associée à une propriété qui est le nom de la fonction de callback.

Méthode `parallelLimit()`

Une variante de la méthode `async.parallel()` vue précédemment est la méthode `async.parallelLimit(tasks, limit, callback)`. Le paramètre `limit` indique le nombre maximal de fonctions de callback pouvant s'exécuter en parallèle. Si ce nombre est 1, cela revient à exécuter `async.series()`.

Dès qu'une fonction de callback est terminée, une autre peut alors commencer à s'exécuter. L'essentiel est que pas plus de `limit` fonctions de callback s'exécuteront en même temps.

Simuler `async.series()` en utilisant `async.parallelLimit()` avec `limit=1`

```
var async = require("async");

async.parallelLimit([function(callback) {
    setTimeout(function() {
        console.log("Fonction1");
        callback(false, 1);
    }, 300);
}, function(callback) {
    setTimeout(function() {
        console.log("Fonction2");
        callback(false, 2);
    }, 200);
}, function(callback) {
    setTimeout(function() {
        console.log("Fonction3");
        callback(false, 3);
    }, 100);
}], 1, function(err, results) { // limit = 1
    console.log("err = " + err);
    console.log("results = " + results);
});
```

On utilise le paramètre `limit = 1` pour indiquer qu'une seule fonction de callback peut s'exécuter à la fois. Cela revient à `async.series()`.

Figure 29-35

Le parallélisme des fonctions de callback est ici limité à 1.

```
C:\Users\Eric\Documents\Node.js>node test.js
Fonction1
Fonction2
Fonction3
err = undefined
results = 1,2,3
C:\Users\Eric\Documents\Node.js>-
```

Méthode `waterfall()`

La méthode `async.waterfall(tasks, callback)` est similaire à la méthode `async.series(tasks, callback)`. Elle exécute les fonctions de callback en série, mais permet aussi que celles-ci se transmettent des résultats l'une après l'autre. En effet, l'appel à `callback(err, result)` effectué dans une fonction de callback transmet le paramètre `result` à la fonction de callback finale, sans que la fonction de callback suivante dans la liste ne puisse l'utiliser. D'où l'intérêt de la méthode `async.waterfall()` pour transmettre des valeurs à la fonction de callback suivante dans la liste.

La fonction de callback finale est de la forme `function(err, result)`, dans laquelle `result` est le résultat transmis par la dernière fonction de callback de la liste, ou une précédente si une erreur est transmise.

Utilisons `async.waterfall()` pour décomposer une succession d'opérations. On souhaite réaliser le calcul de $(1+2)*(3+4)$ au moyen de trois fonctions de callback :

- la première fonction effectue le calcul de $1+2$, qu'elle transmet à la deuxième ;
- la deuxième fonction effectue le calcul de $3+4$, et transmet le premier et le deuxième résultat à la troisième fonction de callback ;
- la troisième fonction de callback effectue la multiplication des deux résultats précédents et transmet ce résultat à la fonction de callback finale.

Effectuer $(1+2)*(3+4)$ au moyen de trois fonctions de callback avec `async.waterfall()`

```
var async = require("async");
var a=1, b=2, c=3, d=4;

async.waterfall([function(callback) {
  setTimeout(function() {
    console.log("Fonction1");
    callback(false, a+b);
  });
}, function(callback) {
  setTimeout(function() {
    console.log("Fonction2");
    callback(false, c+d);
  });
}, function(callback) {
  setTimeout(function() {
    console.log("Fonction3");
    callback(null, (a+b)*(c+d));
  });
}]);
```

```
    }, 300);
}, function(res1, callback) {
  setTimeout(function() {
    console.log("Fonction2");
    callback(false, res1, c+d);
  }, 200);
}, function(res1, res2, callback) {
  setTimeout(function() {
    console.log("Fonction3");
    callback(false, res1 * res2);
  }, 100);
}], function(err, result) {
  console.log("err = " + err);
  console.log("result = " + result);
});
```

Les appels aux fonctions de callback sont de la forme `callback(err, arg1, arg2, etc.)`. La fonction de callback qui suit alors dans la liste doit être de la forme `function(arg1, arg2, etc., callback)` afin que les paramètres soient correctement transmis.

La fonction de callback finale est de la forme `function(err, result)` dans laquelle le paramètre `result` est celui envoyé par la dernière fonction de callback de la liste (ou une précédente si une erreur a été transmise).

Figure 29–36

Le résultat de chaque fonction de callback est transmis à la suivante.

Le résultat de $(1+2)*(3+4)$ est effectivement 21.

Récapitulatif des méthodes du module `async`

Comme on l'a vu précédemment, les méthodes du module `async` peuvent se diviser en deux groupes :

- les méthodes qui agissent sur des tableaux de données ;
- les méthodes qui agissent sur l'enchaînement des fonctions de callback.

Nous récapitulons dans les tableaux 29–3 et 29–4 les principales caractéristiques de ces méthodes.

Tableau 29–3 Méthodes agissant sur les tableaux de données

Méthode	Signification
<code>async.each(arr, iterator, callback)</code>	Appelle la fonction d'itération pour chaque élément du tableau. La fonction d'itération est de la forme <code>function(elem, callback)</code> et appelle <code>callback(err)</code> pour retourner un résultat. La fonction de callback finale est de la forme <code>function(err)</code> et elle est appelée dès qu'une erreur est rencontrée ou lorsque tous les éléments du tableau ont été analysés.
<code>async.eachSeries(arr, iterator, callback)</code>	Appelle la fonction d'itération pour chaque élément du tableau, en série. La fonction d'itération est de la forme <code>function(elem, callback)</code> et appelle <code>callback(err)</code> pour retourner un résultat. La fonction de callback finale est de la forme <code>function(err)</code> et est appelée dès qu'une erreur est rencontrée ou lorsque tous les éléments du tableau ont été analysés.
<code>async.map(arr, iterator, callback)</code>	Appelle la fonction d'itération pour chaque élément du tableau. La fonction d'itération est de la forme <code>function(elem, callback)</code> et appelle <code>callback(err, transformed)</code> pour retourner un résultat. La fonction de callback finale est de la forme <code>function(err, results)</code> et elle est appelée dès qu'une erreur est rencontrée ou lorsque tous les éléments du tableau ont été analysés.
<code>async.mapSeries(arr, iterator, callback)</code>	Appelle la fonction d'itération pour chaque élément du tableau, en série. La fonction d'itération est de la forme <code>function(elem, callback)</code> et appelle <code>callback(err, transformed)</code> pour retourner un résultat. La fonction de callback finale est de la forme <code>function(err, results)</code> et elle est appelée dès qu'une erreur est rencontrée ou lorsque tous les éléments du tableau ont été analysés.
<code>async.filter(arr, iterator, callback)</code>	Appelle la fonction d'itération pour chaque élément du tableau. La fonction d'itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(truthValue)</code> pour indiquer si <code>elem</code> fait partie du tableau résultat (<code>truthValue</code> vaut <code>true</code>) ou pas (<code>truthValue</code> vaut <code>false</code>). La fonction de callback finale est de la forme <code>function(results)</code> et elle est appelée lorsque tous les éléments du tableau ont été analysés.
<code>async.filterSeries(arr, iterator, callback)</code>	Appelle la fonction d'itération pour chaque élément du tableau, en série. La fonction d'itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(truthValue)</code> pour indiquer si <code>elem</code> fait partie du tableau résultat (<code>truthValue</code> vaut <code>true</code>) ou pas (<code>truthValue</code> vaut <code>false</code>). La fonction de callback finale est de la forme <code>function(results)</code> et elle est appelée lorsque tous les éléments du tableau ont été analysés.
<code>async.reject(arr, iterator, callback)</code>	Appelle la fonction d'itération pour chaque élément du tableau. La fonction d'itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(truthValue)</code> pour indiquer si <code>elem</code> fait partie du tableau résultat (<code>truthValue</code> vaut <code>false</code>) ou pas (<code>truthValue</code> vaut <code>true</code>). La fonction de callback finale est de la forme <code>function(results)</code> et elle est appelée lorsque tous les éléments du tableau ont été analysés.

Tableau 29–3 Méthodes agissant sur les tableaux de données (suite)

Méthode	Signification
<code>async.rejectSeries(arr, iterator, callback)</code>	Appelle la fonction d’itération pour chaque élément du tableau, en série. La fonction d’itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(truthValue)</code> pour indiquer si <code>elem</code> fait partie du tableau résultat (<code>truthValue</code> vaut <code>false</code>) ou pas (<code>truthValue</code> vaut <code>true</code>). La fonction de callback finale est de la forme <code>function(results)</code> et elle est appelée lorsque tous les éléments du tableau ont été analysés.
<code>async.detect(arr, iterator, callback)</code>	Appelle la fonction d’itération pour chaque élément du tableau. La fonction d’itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(truthValue)</code> pour indiquer si <code>elem</code> est le résultat (<code>truthValue</code> vaut <code>true</code>) ou pas (<code>truthValue</code> vaut <code>false</code>). La fonction de callback finale est de la forme <code>function(result)</code> et elle est appelée lorsque tous les éléments du tableau ont été analysés.
<code>async.detectSeries(arr, iterator, callback)</code>	Appelle la fonction d’itération pour chaque élément du tableau, en série. La fonction d’itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(truthValue)</code> pour indiquer si <code>elem</code> est le résultat (<code>truthValue</code> vaut <code>true</code>) ou pas (<code>truthValue</code> vaut <code>false</code>). La fonction de callback finale est de la forme <code>function(result)</code> et elle est appelée lorsque tous les éléments du tableau ont été analysés.
<code>async.sortBy(arr, iterator, callback)</code>	Appelle la fonction d’itération pour chaque élément du tableau. La fonction d’itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(err, sortvalue)</code> pour retourner un résultat, qui correspond à un rang de l’élément dans le tableau résultat. La fonction de callback finale est de la forme <code>function(err, results)</code> et elle est appelée dès qu’une erreur est rencontrée ou lorsque tous les éléments du tableau ont été analysés.
<code>async.some(arr, iterator, callback)</code>	Appelle la fonction d’itération pour chaque élément du tableau. La fonction d’itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(truthValue)</code> pour indiquer si <code>elem</code> satisfait la condition (<code>truthValue</code> vaut <code>true</code>) ou pas (<code>truthValue</code> vaut <code>false</code>). La fonction de callback finale est de la forme <code>function(result)</code> et elle est appelée lorsque tous les éléments du tableau ont appelé <code>callback(false)</code> (<code>result</code> vaut <code>false</code>) ou dès qu’un premier élément a appelé <code>callback(true)</code> (<code>result</code> vaut <code>true</code>).
<code>async.every(arr, iterator, callback)</code>	Appelle la fonction d’itération pour chaque élément du tableau. La fonction d’itération est de la forme <code>function(elem, callback)</code> et elle appelle <code>callback(truthValue)</code> pour indiquer si <code>elem</code> satisfait la condition (<code>truthValue</code> vaut <code>true</code>) ou pas (<code>truthValue</code> vaut <code>false</code>). La fonction de callback finale est de la forme <code>function(result)</code> et elle est appelée lorsque tous les éléments du tableau ont appelé <code>callback(true)</code> (<code>result</code> vaut <code>true</code>) ou dès qu’un premier élément a appelé <code>callback(false)</code> (<code>result</code> vaut <code>false</code>).

Tableau 29–4 Méthodes agissant sur l'enchaînement des fonctions de callback

Méthode	Signification
<code>async.series (tasks, [callback])</code>	Effectue l'appel en série des fonctions de callback indiquées dans le tableau <code>tasks</code> . Chaque fonction de callback est de la forme <code>function(callback)</code> et appelle <code>callback(err, result)</code> pour indiquer le résultat de la fonction et déclencher l'appel de la fonction de callback suivante dans le tableau <code>tasks</code> . La fonction de callback finale est optionnelle et elle est de la forme <code>function(err, results)</code> dans laquelle <code>results</code> est le tableau de valeurs <code>result</code> retournées par chaque fonction de callback. La fonction de callback finale est appelée lorsqu'une erreur est déclenchée par l'appel à <code>callback(true, result)</code> ou lorsque toutes les fonctions de callback ont appelé <code>callback(false, result)</code> .
<code>async.parallel (tasks, [callback])</code>	Effectue l'appel en parallèle des fonctions de callback indiquées dans le tableau <code>tasks</code> . Chaque fonction de callback est de la forme <code>function(callback)</code> et appelle <code>callback(err, result)</code> pour indiquer le résultat de la fonction. La fonction de callback finale est optionnelle et elle est de la forme <code>function(err, results)</code> dans laquelle <code>results</code> est le tableau de valeurs <code>result</code> retournées par chaque fonction de callback. La fonction de callback finale est appelée lorsqu'une erreur est déclenchée par l'appel à <code>callback(true, result)</code> ou lorsque toutes les fonctions de callback ont appelé <code>callback(false, result)</code> .
<code>async.parallelLimit (tasks, limit, [callback])</code>	Même principe que <code>async.parallel()</code> , mais en limitant le nombre de fonctions de callback pouvant s'exécuter simultanément (paramètre <code>limit</code>).
<code>async.waterfall (tasks, [callback])</code>	Effectue l'appel en série des fonctions de callback indiquées dans le tableau <code>tasks</code> . Chaque fonction de callback est de la forme <code>function(arg1, arg2, ..., callback)</code> et appelle <code>callback(err, result1, result2, ...)</code> pour indiquer le résultat de la fonction et déclencher l'appel de la fonction de callback suivante dans le tableau <code>tasks</code> . La fonction de callback finale est optionnelle et elle est de la forme <code>function(err, result1, result2, ...)</code> dans laquelle chaque <code>result</code> a pour valeur celle indiquée par l'appel précédent à <code>callback(err, result1, result2, ...)</code> . La fonction de callback finale est appelée lorsqu'une erreur est déclenchée par l'appel à <code>callback(true result1, result2, ...)</code> ou lorsque toutes les fonctions de callback ont appelé <code>callback(false, result1, result2, ...)</code> .

30

Le module supervisor

Lorsqu'un serveur HTTP Node est lancé, il est à l'écoute des requêtes des utilisateurs. Si le code du serveur est modifié, il faut arrêter le serveur, puis le relancer afin de prendre en compte les modifications effectuées. L'arrêt du serveur et son redémarrage sont des opérations à effectuer manuellement.

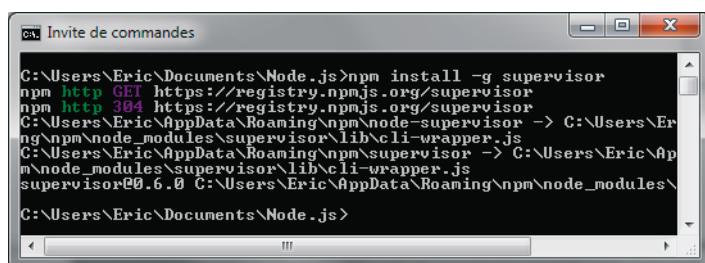
L'intérêt du module `supervisor` est qu'il permet d'effectuer ces opérations de façon automatique. Dès qu'un fichier situé dans le répertoire du serveur (ou dans ses sous-répertoires) est modifié, le serveur est arrêté et le programme principal du serveur est relancé. Cela évite d'oublier de relancer le serveur après avoir modifié son code source.

Installer le module supervisor

Le module `supervisor` s'installe au moyen de la commande `npm install -g supervisor`. Il est installé en global afin d'être accessible pour toutes les applications Node qui le souhaitent.

Figure 30-1

Installation du module supervisor



```
C:\Users\Eric\Documents\Node.js>npm install -g supervisor
npm http GET https://registry.npmjs.org/supervisor
npm http 304 https://registry.npmjs.org/supervisor
C:\Users\Eric\AppData\Roaming\npm\node-supervisor -> C:\Users\Eric\ng
nmp\node_modules\supervisor\lib\cli-wrapper.js
C:\Users\Eric\AppData\Roaming\npm\supervisor -> C:\Users\Eric\Ap
m\node_modules\supervisor\lib\cli-wrapper.js
supervisor@0.6.0 C:\Users\Eric\AppData\Roaming\npm\node_modules\

C:\Users\Eric\Documents\Node.js>
```

Utiliser le module supervisor

Le module `supervisor` est constitué d'un exécutable Node qui permet de lancer l'application Node. Plutôt que de lancer notre application `test.js` en tapant la commande `node test.js`, on introduira la commande `supervisor test.js`. Notre application `test.js` va alors s'exécuter (de la même manière que si on l'avait lancée par `node test.js`).

L'intérêt sera que toute modification d'un fichier quelconque situé dans le même répertoire que `test.js` (y compris le fichier `test.js`) ou dans un de ses sous-répertoires, provoquera le redémarrage de l'application Node, sans avoir à la redémarrer nous-mêmes.

Pour utiliser le module `supervisor`, écrivons une application de test lançant un serveur HTTP.

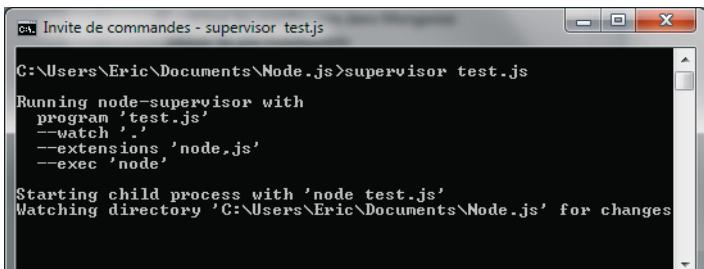
Fichier test.js

```
var http = require("http");
var server = http.createServer(function(request, response) {
    response.end("Bienvenue");
});
server.listen(3000);
```

À chaque connexion d'un utilisateur sur le serveur, le message "Bienvenue" apparaît.

Tapons la commande `supervisor test.js` pour exécuter cette application grâce au module `supervisor`.

Figure 30–2
Exécution d'une application
Node au moyen du module
supervisor

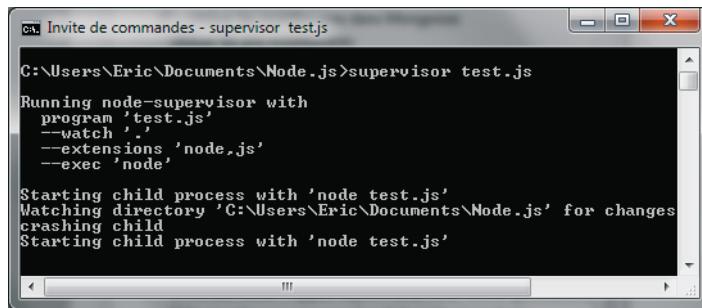


Les informations affichées à l'écran montrent que la commande `supervisor` a créé un processus fils et qu'elle observe les modifications éventuelles dans le répertoire courant (et ses sous-répertoires).

Modifions le fichier `test.js` pour afficher maintenant le message "Bienvenue à tous" au lieu de "Bienvenue". Dès que le fichier `test.js` est sauvegardé sur disque, la modification du fichier est détectée par la commande `supervisor` et le fichier `test.js` est redémarré, prenant ainsi en compte la modification effectuée.

Figure 30–3

L'application Node est relancée automatiquement par le module supervisor.



```
C:\Users\Eric\Documents\Node.js>supervisor test.js
Running node-supervisor with
  program 'test.js'
  --watch ,,
  --extensions 'node.js'
  --exec 'node'

Starting child process with 'node test.js'
Watching directory 'C:\Users\Eric\Documents\Node.js' for changes
crashing child
Starting child process with 'node test.js'
```


Le module node-inspector

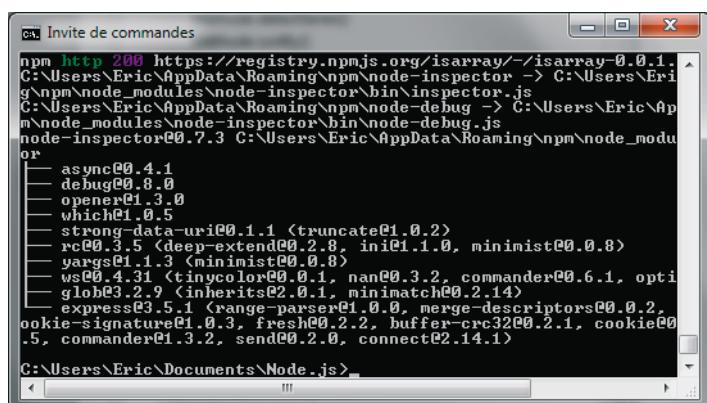
Le débogage des programmes avec Node est une tâche difficile, qui requiert l'utilisation de nombreuses instructions `console.log()`. Cependant, le module `node-inspector` permet également de déboguer les programmes sous forme visuelle, en utilisant des moyens classiques tels que la mise en place de points d'arrêt (*breakpoints*) ou l'affichage du contenu de variables (*dump*).

Installer le module node-inspector

Le module `node-inspector` est un module Node qui s'installe au moyen de la commande `npm install -g node-inspector`. On l'installe en global afin qu'il soit accessible pour l'ensemble de nos applications.

Figure 31-1

Installation du module
node-inspector



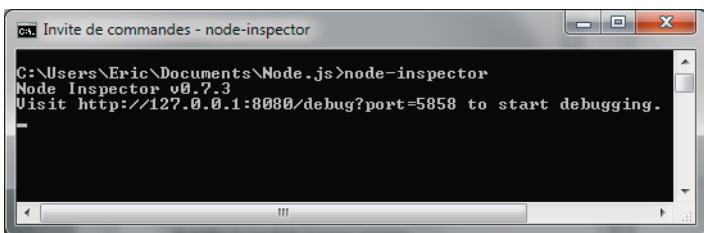
```
cmd Invite de commandes
npm http 200 https://registry.npmjs.org/isarray/-/isarray-0.0.1
C:\Users\Eric\AppData\Roaming\npm\node-inspector -> C:\Users\Eric\g
g:\npm\node_modules\node-inspector\bin\node-inspector.js
C:\Users\Eric\AppData\Roaming\npm\node-debug -> C:\Users\Eric\Ap
m\node_modules\node-inspector\bin\node-debug.js
node-inspector@0.7.3 C:\Users\Eric\AppData\Roaming\npm\node_modu
or
└── [REDACTED]
```

The screenshot shows a Windows Command Prompt window titled "Invite de commandes". The command entered was "npm install -g node-inspector". The output shows the download and extraction of the "node-inspector" module from the npm registry. The module version installed is 0.7.3. The path "C:\Users\Eric\AppData\Roaming\npm\node_modules\node-inspector\bin\node-inspector.js" is highlighted in blue, indicating it is a script file. The rest of the output is truncated at the bottom.

Utiliser le module node-inspector

Pour déboguer un programme avec `node-inspector`, il faut effectuer les opérations suivantes dans l'ordre indiqué. Tout d'abord, nous lançons la commande `node-inspector` qui exécute un processus Node se mettant à l'écoute des connexions des utilisateurs pour le débogage.

Figure 31-2
Lancement de la commande
`node-inspector`



Une fois la commande `node-inspector` lancée, on peut commencer le débogage en tapant l'une des deux commandes suivantes dans un nouvel interpréteur de commandes. Elles permettent toutes les deux de lancer l'application (ici, le fichier `app.js` que l'on a créé au chapitre 28, « Construction d'une application client serveur »). La différence entre les deux commandes est que la première lance l'application directement, la seconde s'arrête sur la première instruction du programme et nous laisse libre de l'exécuter instruction par instruction.

Commande permettant de lancer le débogueur et de mettre le serveur en attente

```
node --debug app.js
```

Une fois cette commande tapée dans un interpréteur de commandes, le serveur est lancé et se met en attente des requêtes des utilisateurs.

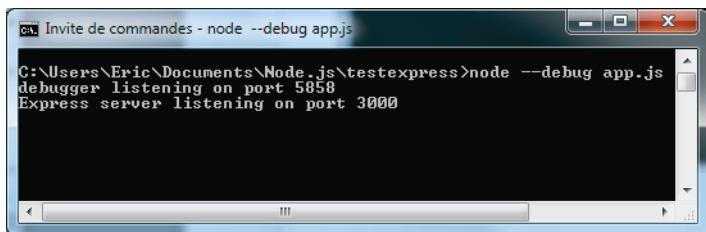
Commande permettant de lancer le débogueur et de mettre un point d'arrêt sur la première instruction du programme

```
node --debug-brk app.js
```

Une fois cette commande tapée dans un interpréteur de commandes, le programme est bloqué sur la première instruction. Il attend que l'on poursuive son exécution.

Tapons par exemple la première ligne de commande dans un nouvel interpréteur de commandes.

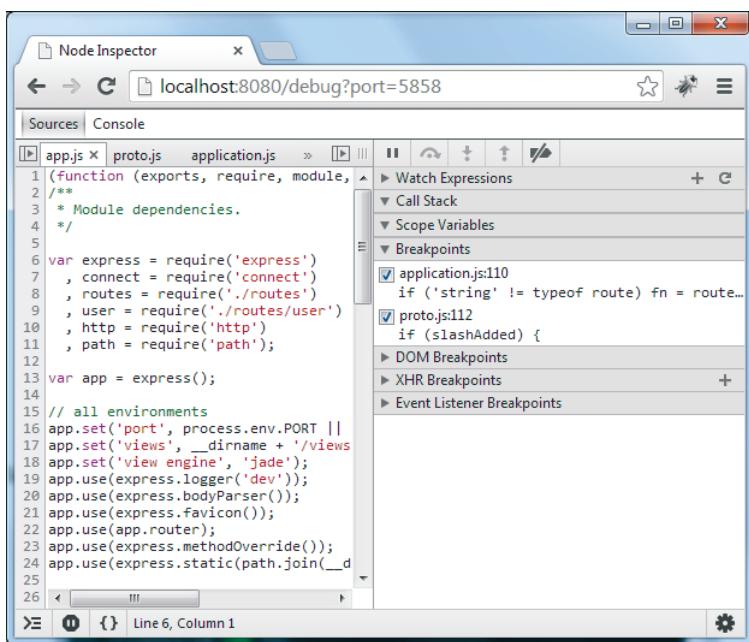
Figure 31–3
Mise en attente du débogueur pour notre application



Une fois cette commande lancée, on a donc deux interpréteurs de commandes en cours d'exécution.

Quelle que soit la commande tapée, il reste maintenant à exécuter ou arrêter le programme `app.js` en cours d'exécution. Ouvrez le navigateur Chrome (et uniquement Chrome, cela ne fonctionne qu'avec lui), puis dans la barre d'adresses saisissez l'URL `http://localhost:8080/debug?port=5858` indiquée lors du démarrage de `node-inspector`.

Figure 31–4
Débogage de notre application en temps réel



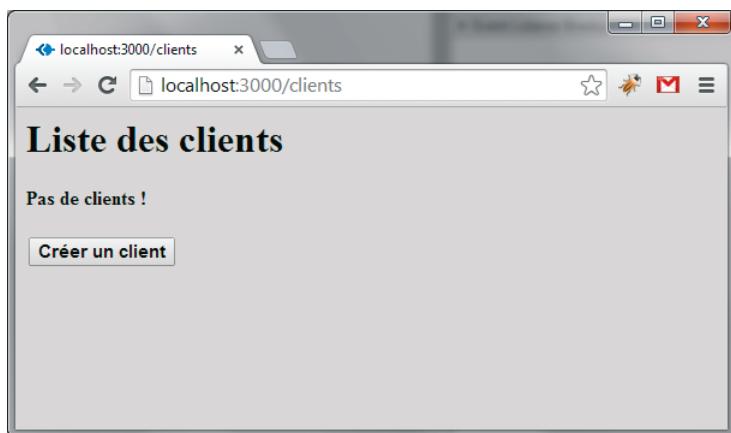
Le fichier `app.js` s'affiche dans le navigateur. Il est possible d'y placer des points d'arrêt, en cliquant simplement sur une ligne d'instruction. Les points d'arrêt positionnés s'affichent dans la partie droite de la fenêtre.

Cette fenêtre est celle du code du programme. Pour tester le programme, il faut ouvrir un nouveau navigateur web qui accède à l'application. Ici, l'URL du programme est `http://localhost:3000/clients`. Les points d'arrêt positionnés sont activés en fonction des actions de l'utilisateur dans la fenêtre de l'application.

Le navigateur ouvert pour tester l'application est quelconque (Internet Explorer, Firefox, Chrome, etc.). Seul le navigateur contenant le débogueur doit être Chrome.

Figure 31–5

La fenêtre de notre application en cours de débogage



Le module mongo-express

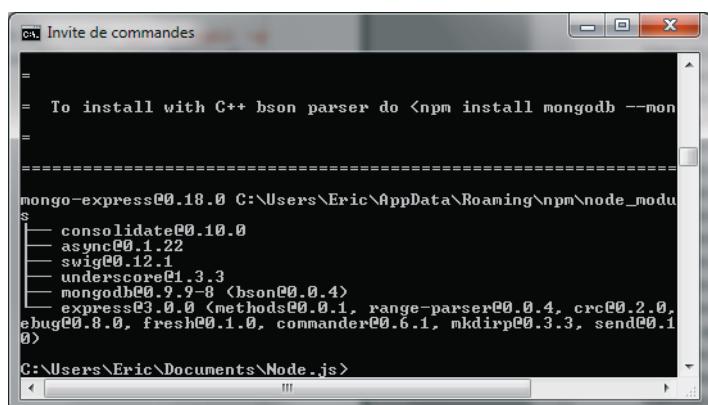
La base de données MongoDB décrite dans cet ouvrage s'utilise pour l'instant en ligne de commandes, au moyen des instructions étudiées au chapitre 19, « Introduction à MongoDB ». Le module [mongo-express](#) fournit une interface graphique pour administrer la base de données, ce qui est plus pratique pour voir ou modifier le contenu de celle-ci.

Installer le module mongo-express

Le module [mongo-express](#) s'installe à partir de [npm](#), au moyen de la commande `npm install -g mongo-express`. Le module est installé en global de façon à être accessible pour toutes les applications Node existantes ou à venir.

Figure 32-1

Installation du module
mongo-express



The screenshot shows a Windows Command Prompt window titled "Invite de commandes". The command entered is "npm install -g mongo-express". The output shows the module being installed, listing its dependencies: consolidate@0.10.0, async@0.1.22, swig@0.12.1, underscore@1.3.3, mongodb@0.9.9-8 (bson@0.0.4), express@3.0.0 (methods@0.0.1, range-parser@0.0.4, crc@0.2.0, ebug@0.8.0, fresh@0.1.0, commander@0.6.1, mkdirp@0.3.3, send@0.1.0). The path shown is C:\Users\Eric\Documents\Node.js>.

Le module `mongo-express` s'est inscrit dans le répertoire `node_modules` global, accessible pour toutes les applications Node. Il s'agit ici du répertoire indiqué dans la fenêtre de commandes, à savoir `C:\Users\Eric\AppData\Roaming\npm\node_modules\mongo-express`.

Dans ce répertoire d'installation, figure le fichier `config.default.js`. Il faut le dupliquer en un nouveau fichier nommé simplement `config.js`.

Copier le fichier config.default.js en config.js

```
> copy config.default.js config.js
```

Le fichier `config.js` est obligatoire pour que le module `mongo-express` fonctionne.

Utiliser le module mongo-express

Une fois le module `mongo-express` installé et le fichier `config.js` créé à partir du fichier `config.default.js`, on peut lancer l'application `mongo-express`. Pour cela, on tape la commande `node app.js` depuis le répertoire où `mongo-express` est installé.

Figure 32–2

Lancement du module mongo-express

```
C:\ Invite de commandes - node app.js
C:\Users\Eric\AppData\Roaming\npm\node_modules\mongo-express>copy config.default.js config.js
1 fichier(s) copié(s.)

C:\Users\Eric\AppData\Roaming\npm\node_modules\mongo-express>node app.js
Mongo Express server listening on port 8081
Database connected!
{ [MongoError: auth fails] name: 'MongoError', ok: 0, errmsg: 'a admin Database connected' }
```

Un serveur HTTP est maintenant disponible pour afficher le contenu des bases de données MongoDB situées sur le serveur. Le serveur géré par `mongo-express` est accessible à partir de l'adresse `http://localhost:8081`, comme indiqué dans la fenêtre précédente.

Figure 32–3

Affichage des bases de données MongoDB présentes sur le serveur

The screenshot shows a web browser window titled "Home - Mongo Express" with the URL "localhost:8081". The main content area is titled "Mongo Express". It displays a list of databases under four categories: "ADMIN", "CLIENTS", "MYDB", and "MYDB_SAVE". Each category has a "View Database" link next to it. The "View Database" links are blue and underlined, indicating they are clickable.

Catégorie	Bases de données
ADMIN	View Database
CLIENTS	View Database
MYDB	View Database
MYDB_SAVE	View Database

Il ne reste plus qu'à administrer chacune des bases de données MongoDB de notre système.

Index

-
- \$exists 388
 - \$gt 385
 - \$gte 385
 - \$in 385
 - \$inc 414
 - \$lt 385
 - \$lte 385
 - \$ne 385
 - \$nin 385
 - \$rename 414
 - \$set 414
 - \$type 392
 - \$unset 414, 463
 - \$where 394
 - \$where(jsexpr) 452
 - :format 295
 - :id 278
 - __dirname 26
 - __filename 26
 - _id 406
 - _method 267
 - _read() 75
 - _write(chunk, encoding, callback) 87
 - A**
 - addListener(event, listener) 34, 45
 - app.all(url, callback) 270
 - app.delete(url, callback) 266
 - app.get(url, callback) 266
 - app.js 256
 - app.param(name, callback) 341
 - app.post(url, callback) 266
 - app.put(url, callback) 266
 - app.render(name, [obj], callback) 344
 - app.resource() 291
 - app.router 302
 - app.routes 271
 - app.set("view engine", format) 332
 - app.set("views", directory) 332
 - app.use() 221, 284
 - Array 428
 - async 531
 - async.detect(arr, iterator, callback) 547
 - async.detectSeries(arr, iterator, callback) 548
 - async.each(arr, iterator, callback) 532
 - async.eachSeries(arr, iterator, callback) 536
 - async.every(arr, iterator, callback) 554
 - async.filter(arr, iterator, callback) 542, 545
 - async.filterSeries(arr, iterator, callback) 544
 - async.map(arr, iterator, callback) 537
 - async.mapSeries(arr, iterator, callback) 541
 - async.parallel(tasks, callback) 560
 - async.parallelLimit(tasks, limit, callback) 563
 - async.rejectSeries(arr, iterator, callback) 546
 - async.series(tasks, callback) 556
 - async.some(arr, iterator, callback) 552
 - async.sortBy(arr, iterator, callback) 549
 - async.waterfall(tasks, callback) 564
 - B**
 - bodyParser 239
 - Boolean 428
 - buf.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd]) 68
 - buf.slice([start], [end]) 68
 - buf.toString(encoding) 69
 - buf.write(string, offset, length, encoding) 67
 - Buffer 63, 428
 - Buffer.byteLength(string, [encoding]) 68, 70
 - Buffer.concat(list) 68
 - C**
 - child.stderr 119
 - child.stdout 119

child_process 116
 child_process.exec(cmd, callback) 116
 child_process.spawn("node", args) 119
 children 27
 clearInterval(t) 71
 clearTimeout(t) 71
 client.send(buf, offset, length, port, address, [callback]) 154
 close 133, 158, 176
 collections 374
 colors 18
 CONNECT 215, 266
 connect() 216
 connect(name) 378
 connect(nomDatabase) 422
 connect.createServer() 219
 connect.logger(format) 232
 connection 133, 199
 connect-mongo 245
 console.dir(obj) 48
 console.error([data], [...]) 48
 console.info([data], [...]) 48
 console.log([data], [...]) 47
 console.time(label) 48
 console.timeEnd(label) 48
 console.warn([data], [...]) 48
 cookieParser 242
 count(callback) 455
 count(conditions, callback) 455
 createServer() 14
 cursor.count() 407
 cursor.length() 407

D

data 74
 Date 428
 db.adminCommand("listDatabases") 376
 db.copyDatabase(origin, destination, hostname) 422
 db.dropDatabase() 422
 db.getCollectionNames() 422
 db.getName() 422
 db.nomCollection.copyTo(newCollection) 421
 db.nomCollection.drop() 421

db.nomCollection.find(query, projection) 382
 db.nomCollection.findOne(query) 408
 db.nomCollection.findOne(query, projection) 382
 db.nomCollection.insert(doc) 379
 db.nomCollection.remove(query, justOne) 417
 db.nomCollection.renameCollection(newCollection) 421
 db.nomCollection.save(document) 409
 db.nomCollection.update(query, update, options) 409
 dependencies 30
 dgram.createSocket (type, [callback]) 149
 disconnect 200
 doc.populate(attribut, select, callback) 504
 doc.remove(callback) 471
 doc.save(callback) 435, 464
 documents 374

E

EJS 353
 Embedded JavaScript 353
 emit(event) 34
 emit(event, arg1, arg2, ...) 45
 enum 484
 equals(value) 453
 error 133, 158
 errorHandler 235
 errors 479
 events.EventEmitter 31
 exec(callback) 452
 exists(boolean) 453
 exports 27
 Express 251
 express-resource 289, 518

F

favicon 241
 filename 27
 fs 95
 fs.appendFile(filename, data, callback) 104
 fs.appendFileSync(filename, data) 104
 fs.close(fd, callback) 99
 fs.closeSync(fd) 99

fs.createReadStream(path) 113
fs.createWriteStream(path) 113
fs.exists(path, callback) 111
fs.existsSync(path) 111
fs.link(srcpath, destpath, callback) 105
fs.linkSync(srcpath, destpath) 105
fs.mkdir(path, callback) 108
fs.mkdirSync(path) 108
fs.open(path, flags, callback) 98
fs.openSync(path, flags) 98
fs.read(fd, buffer, offset, length, position,
 callback) 101
fs.readdir(path, callback) 110
fs.readdirSync(path) 110
fs.readFile(filename, [options], callback) 100
fs.readFileSync(filename, [options]) 100
fs.readSync(fd, buffer, offset, length,
 position) 101
fs.rename(oldpath, newpath, callback) 107
fs.renameSync(oldpath, newpath) 107
fs.rmdir(path, callback) 109
fs.rmdirSync(path) 109
fs.stat(path, callback) 112
fs.statSync(path) 112
fs.unlink(path, callback) 106
fs.unlinkSync(path) 106
fs.writeFile(filename, data, callback) 103
fs.writeFileSync(filename, data) 103

G

gt(value) 453
gte(value) 453

H

hash 58
HEAD 266
highWaterMark 86
host 58
hostname 58, 179
href 58
http 163
http.createServer([requestListener]) 163
http.get(options, callback) 184
http.IncomingMessage 166

http.request(options, callback) 178
http.ServerResponse 169

I

id 27
in(array) 453
index.js 10
io 211
io.connect(url) 197, 202
io.sockets.emit(event, params) 205

J

JADE 332

L

listeners(event) 45
listening 128, 133, 158
loaded 27
logger 232
lt(value) 453
lte(value) 453

M

main 11, 30
match 484
max 481, 483
MemoryStore 246
message 158
method 179
methodOverride 367
methodOverride 247
min 483
Model View Controller 259
modèles 426
modelName.create(doc, callback) 437
modelName.find(conditions, callback) 442
modelName.findById(conditions) 454
modelName.findById(conditions, callback) 454
modelName.findByIdAndRemove (id,
 callback) 473
modelName.findByIdAndUpdate (id, update,
 options, callback) 468
modelName.findOne(conditions) 454
modelName.findOne(conditions, callback) 454

ModelName.findOneAndRemove (conditions, options, callback) 472
 ModelName.findOneAndUpdate (conditions, update, options, callback) 466
 ModelName.populate(docs, options, callback) 502
 ModelName.remove(conditions, callback) 469
 ModelName.update(conditions, update, options, callback) 460
 module 25
 module.exports 21
 modules 7
 mongo 376
 MongoDB 373
 mongo-express 577
 Mongoose 423
 mongoose.connect(url, options) 424
 mongoose.model(modelName, schema) 428
 mongoose.Query 450
 mongoose.Query.populate(attribut, select) 501
 mongoose.Schema(definition, options) 427
 mongoose.Schema.Types.Mixed 428
 mongoose.Schema.Types.ObjectId 428
 MongoStore 246
 M-SEARCH 266
 msg 486
 multi 416
 MVC 259

N

name 30
 ne(value) 453
 net.connect(port, [host], [callback]) 135
 net.createServer([callback]) 125
 Netcat 148
 next 219
 next("route") 312
 next() 305
 nin(array) 453
 Node Package Manager 16
 node_modules 11, 259
 node-inspector 573
 NOTIFY 267
 npm 16

Number 428

O

on(event, listener) 34, 45
 once(event, listener) 45
 OPTIONS 266

P
 package.json 10, 29, 253
 parent 27
 PATCH 266
 path 58, 179
 path.basename(p) 62
 path.dirname(p) 62
 path.extname(p) 62
 path.join([path1], [path2], [...]) 62
 path.normalize(p) 62
 path.sep 62
 pathname 58
 paths 27
 population 493
 port 58, 179
 post 514
 Postman 268
 pre 506
 projection 405
 protocol 58
 public 259

Q

query 58, 238
 querystring.parse(str, [sep], [eq]) 61
 querystring.stringify(obj, [sep], [eq]) 61

R

readable.pause() 84
 readable.pipe(writable) 90
 readable.read([size]) 81
 readable.resume() 84
 readable.setEncoding(encoding) 80
 readable.unpipe([writable]) 90
 regex(rerexp) 453
 remove 506
 removeAllListeners([event]) 38
 removeAllListeners(event) 45

- removeListener(event, listener) 37, 45
REPL 6
Representational State Transfer 247
req.body 349
req.host 350
req.ip 350
req.method 350
req.param(name) 349
req.params 349
req.path 350
req.protocol 350
req.query 349
req.url 350
req.xhr 350
request 177
request.client 167
request.headers 167
request.method 167
request.url 167
require(module) 7
required 476, 483
res.download(name) 330
res.json(obj) 328
res.redirect(url) 336
res.render(name, [obj]) 332
res.send(html) 326
res.sendFile(name) 330
res.set(name, value) 324
res.set(obj) 324
res.status(statusCode) 323
respond 488
response.end([chunk], [encoding], [callback]) 169
response.setHeader(name, value) 174
response.write(chunk, [encoding], [callback]) 169
response.writeHead(statusCode, headers) 174
REST 247, 518
routes 259, 261
- S**
- save 506
schémas 426
scripts 30
search 58
- select(attribut) 452
server.address() 130
server.bind(port, [callback]) 149
server.close([callback]) 130, 177
server.getConnections(callback) 130
server.listen(port, [callback]) 126, 130, 164, 177
server.maxConnections 130
session 242
setInterval(cb, ms) 71
setMaxListeners(n) 45
setTimeout(cb, ms) 71
socket.address() 159
socket.bind(port, [address], [callback]) 159
socket.broadcast.emit(event, params) 205
socket.close() 159
socket.emit(event, params) 197, 203
socket.get(name, callback) 207
socket.io 195
socket.on(event, callback) 202
socket.send(buf, offset, length, port, address, [callback]) 159
socket.set(name, value) 207
sort(attribut) 452
start 30
static 237
stream 74
stream.Duplex 92
stream.Readable 74
stream.Writable 86
String 428
SUBSCRIBE 267
supervisor 569
- T**
- TCP 123
Telnet 124
TRACE 266
Transmission Control Protocol 123
- U**
- UDP 147
udp4 149
unique 483
UNSUBSCRIBE 267

url.format(urlObj) 57
url.parse(urlStr, [parseQueryString]) 57
url.resolve(from, to) 57
User Datagram Protocol 147
util.format(format, [...]) 52
util.inherits(newClass, oldClass) 55
util.inspect(obj, options) 54
util.isArray(object) 57
util.isDate (object) 57
util.isError (object) 57
util.isRegExp (object) 57

V
validate 486, 506
validator 486
version 30
views 259

W
where() 450
writable.end([chunk], [encoding], [callback]) 89
writable.write(chunk, [encoding], [callback]) 87