Les fondamentaux de l'API

Ce chapitre explore en détail l'API de CouchDB. Il aborde les choses sérieuses et les astuces, de même que les bonnes pratiques ; il aide à éviter les embûches.

Nous commencerons par revoir les opérations de base décrites dans le chapitre précédent et profiterons de cette occasion pour entrer plus en profondeur. Nous montrerons aussi ce que Futon doit incorporer dans sa partie « métier » pour nous fournir ces fonctionnalités ravissantes.

Ce chapitre est à la fois une introduction aux fondamentaux de l'API de CouchDB et une référence. Si vous ne parvenez pas à vous souvenir comment exécuter telle requête, ou pourquoi certains paramètres sont requis, vous pouvez toujours vous référer à ce chapitre (nous sommes nous-mêmes certainement les premiers utilisateurs de ce chapitre).

Pour expliquer chaque détail de l'API, nous avons parfois besoin d'expliquer aussi le raisonnement qui le sous-tend. Nous saisissons ces occasions pour vous expliquer pourquoi CouchDB se comporte ainsi.

L'API peut être catégorisée de la manière suivante. Nous explorerons chacune des catégories :

- Serveur
- Bases de données
- Documents
- Réplication

Serveur

Cette requête est simple et basique. Elle peut permettre de vérifier que CouchDB est lancé. Elle peut aussi permettre à certaines bibliothèques de vérifier la version du moteur. Nous utilisons de nouveau l'outil `curl` :

```
curl http://127.0.0.1:5984/
```

CouchDB répond, tout heureux de participer :

```
{"couchdb":"Welcome","version":"0.10.1"}
```

Vous récupérez une chaîne de caractères JSON qui, transformée en un objet natif ou une structure de données de votre langage de programmation, vous permet d'accéder aux deux chaînes que sont la bienvenue et la version.

Ce n'est pas d'une utilité folle, mais cela illustre bien la manière dont CouchDB se comporte. Vous envoyez une requête HTTP et vous recevez une chaîne JSON dans la réponse HTTP.

Bases de données

Faisons quelque chose d'un peu plus utile : créons des bases de données. Pour les puristes, CouchDB est un *système de gestion de bases de données* (SGBD). Cela signifie qu'il peut accueillir plusieurs *bases de données*. Une base de données est un conteneur qui stocke des données corrélées ; nous reviendrons sur ce que cela signifie. Dans la pratique, les termes se recouvrent : on parle d'un SGBD comme d'une « base de données » et réciproquement. Il se peut que nous suivions cette tendance, aussi ne vous en formalisez-vous pas. De manière générale, le contexte est explicite et indique si nous parlons du système CouchDB dans son entièreté ou d'une seule base de données.

Bref, créons-en une ! Nous voulons stocker nos albums de musique préférés, aussi nommons-nous notre base, de manière très inventive, `albums`. Notez que nous utilisons à nouveau l'option `-X` pour dire à `curl` d'envoyer une requête de type `PUT` plutôt qu'une

requête de type `GET` comme il le ferait par défaut.

```
curl -X PUT http://127.0.0.1:5984/albums
```

CouchDB répond :

```
{"ok":true}
```

C'est fait ! Vous venez de créer une base de données et CouchDB vous indique que tout s'est bien passé. Que se passe-t-il si vous tentez de créer une base de données qui existe déjà ? Essayons :

```
curl -X PUT http://127.0.0.1:5984/albums
```

CouchDB répond :

```
{"error":"file_exists","reason":"The database could not be created, the file already exists."}
```

Nous obtenons une erreur ; voilà qui est pratique. Nous en apprenons aussi un peu plus sur la manière dont CouchDB fonctionne : il stocke chaque base de données dans un seul fichier. Très simple. Cela n'est pas sans conséquence, mais passons les détails pour le moment car nous explorons le système de stockage sous-jacent dans l'**Annexe F, La force des arbres-B (btree.html)**.

Créons une nouvelle base de données, cette fois avec l'option `-v` (pour verbeux) de `curl`. Cette option invite `curl` à nous montrer tous les détails de la requête et de la réponse :

```
curl -vX PUT http://127.0.0.1:5984/albums-backup
```

`curl` affiche :

```
* About to connect() to 127.0.0.1 port 5984 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 5984 (#0)
> PUT /albums-backup HTTP/1.1
> User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
> Host: 127.0.0.1:5984
> Accept: */*
>
< HTTP/1.1 201 Created
< Server: CouchDB/0.9.0 (Erlang OTP/R12B)
< Date: Sun, 05 Jul 2009 22:48:28 GMT
< Content-Type: text/plain;charset=utf-8
< Content-Length: 12
< Cache-Control: must-revalidate
<
{"ok":true}
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

Quel bavard ! Expliquons ligne après ligne pour comprendre ce qu'il se passe et relevons ce qui est important. Quand vous aurez vu ce type d'affichage quelques fois, vous repérerez plus vite l'essentiel.

```
* About to connect() to 127.0.0.1 port 5984 (#0)
```

`curl` nous dit qu'il est sur le point d'établir la connexion *TCP* avec le serveur CouchDB que nous avons spécifié dans notre URI. Pas très important, sauf lors de la résolution de problèmes réseau.

```
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 5984 (#0)
```

`curl` indique qu'il s'est bien connecté à CouchDB. Sauf pour des problèmes réseau, ce n'est pas important non plus.

Les lignes suivantes commencent toutes par les caractères > et <. > signifie que la ligne a été envoyée à CouchDB (sans le >) ; < signifie que la ligne a été reçue par `curl`.

```
> PUT /albums-backup HTTP/1.1
```

Cela initialise la requête HTTP. Sa *méthode* [NdT : « Dans le protocole HTTP, une méthode est une commande spécifiant un type de requête, c'est-à-dire qu'elle demande au serveur d'effectuer une action. » Wikipédia] est `PUT`, son *URI* est `/albums-backup` et la version du protocole HTTP est `HTTP/1.1`. Il existe aussi `HTTP/1.0`, qui peut s'avérer plus simple dans certains cas, mais pour des raisons pratiques, vous devriez utiliser `HTTP/1.1`.

Ensuite, nous voyons plusieurs *en-têtes*. Ils permettent d'adjoindre des informations à la requête.

```
> User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
```

L'en-tête `User-Agent` indique à CouchDB quel client est utilisé pour envoyer la requête HTTP. Nous n'apprenons rien de nouveau : c'est `curl`. Cet en-tête est souvent utilisé pour différencier les clients avec lesquels des problèmes de compatibilités sont connus et pour lesquels la réponse devra être différente. Il aide aussi à déterminer la plateforme de l'utilisateur. Cette information peut être utilisée à des fins statistiques ou techniques. Pour CouchDB, l'en-tête `User-Agent` n'importe pas.

```
> Host: 127.0.0.1:5984
```

L'en-tête `Host` est requis par HTTP 1.1. Il indique l'origine de la requête.

```
> Accept: */*
```

L'en-tête `Accept` indique à CouchDB que `curl` accepte tous les types de médias. Nous verrons par la suite en quoi il est utile.

```
>
```

Une ligne vide indique que la partie des en-têtes de la requête est terminée et que la suite contient les données que nous envoyons au serveur. Dans ce cas, nous n'envoyons pas la moindre donnée, donc le reste de l'affichage de *curl* concerne la réponse HTTP.

```
< HTTP/1.1 201 Created
```

La première ligne de la réponse HTTP envoyée par CouchDB inclut la version du protocole HTTP (pour permettre le traitement de son format), un *code d'état* et un *message d'état*. Des requêtes différentes appellent des réponses différentes. Il en existe une ribambelle qui indique au client (`curl` dans notre cas) quel effet a eu la requête, que ce soit un succès ou un échec. La RFC 2616 (spécifiant HTTP 1.1) définit le comportement à avoir selon le code d'état. CouchDB se conforme parfaitement à cette RFC.

Le code *201 Created* indique au client que la ressource qu'il a sollicitée a été créée. Aucune surprise ici, mais si vous vous souvenez du code d'erreur que nous avons eu quand nous avons tenté de recréer une base de données existante, vous savez qu'il aurait pu en être tout autrement. Réagir aux réponses en se basant sur le code d'état est une pratique courante. Par exemple, tous les codes supérieurs ou égaux à 400 signalent une erreur. Si vous vouliez simplifier votre logique de traitement et réagir immédiatement à une erreur, vous pourriez simplement vérifier si le code d'état est `>= 400`.

```
< Server: CouchDB/0.10.1 (Erlang OTP/R13B)
```

L'en-tête `Server` est pratique pour les diagnostics. Il indique quelle version de CouchDB et quelle version sous-jacente d'Erlang sont utilisées. En général, vous pouvez ignorer cet en-tête, mais c'est bon à savoir si vous en avez besoin.

```
< Date: Sun, 05 Jul 2009 22:48:28 GMT
```

L'en-tête `Date` vous indique l'heure du serveur. Puisque les horloges du client et du serveur ne sont pas nécessairement synchronisées, cet en-tête est une simple information. Vous ne devriez pas bâtir d'applications sur ce critère !

```
< Content-Type: text/plain;charset=utf-8
```

L'en-tête `Content-Type` indique le type MIME du corps de la réponse HTTP ainsi que son jeu de caractères. Nous savons déjà que CouchDB renvoie des chaînes de caractères JSON. Le `Content-Type` approprié est donc `application/json`. Alors pourquoi voyons-nous `text/plain` ? C'est là que le pragmatisme l'emporte sur le purisme. Envoyer un en-tête avec `Content-Type` positionné à `application/json` à un navigateur va déclencher l'option de téléchargement plutôt que d'afficher le contenu. Puisqu'il est très utile de pouvoir tester CouchDB depuis un navigateur, CouchDB indique `text/plain`, ce qui fait que tous les navigateurs affichent le JSON comme un texte.

Il existe quelques greffons qui influent sur le comportement de votre navigateur vis-à-vis du JSON, mais ils ne sont pas installés par défaut.

Vous rappelez-vous de l'en-tête `Accept` qui était positionné à `\*/\* -> */*` pour dire quel type MIME nous intéressait ? Si vous envoyez `Accept: application/json` dans votre requête, CouchDB sait que vous pouvez exploiter une réponse purement JSON avec le `Content-Type` adéquat ; il l'utilisera donc à la place de `text/plain`.

```
< Content-Length: 12
```

L'en-tête `Content-Length` indique simplement combien d'octets composent le corps de la réponse.

```
< Cache-Control: must-revalidate
```

L'en-tête `Cache-Control` vous indique, à vous ou à tout serveur mandataire qui se trouverait sur le chemin du serveur CouchDB, que la réponse ne doit pas être stockée en antémémoire (*cached* en anglais).

```
<
```

Cette ligne vide sépare l'en-tête du corps du message.

```
{"ok":true}
```

Nous avons déjà vu cela auparavant.

```
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

Dans ces deux dernières lignes, `curl` nous indique qu'il a conservé la connexion TCP ouverte pendant un moment, mais qu'il l'a fermée après avoir reçu toute la réponse.

Dans les chapitres suivants, nous verrons quelques requêtes avec l'option `-v`, mais nous omettrons certains en-têtes décrits ici pour ne laisser que ceux qui sont utiles dans les cas abordés.

C'est bien beau de créer une base de données, mais comment la supprime-t-on ? Tout simplement en changeant la méthode HTTP :

```
> curl -vX DELETE http://127.0.0.1:5984/albums-backup
```

Ceci supprime la base de données. La requête va supprimer le fichier qui stocke le contenu de la base. Il n'y a pas de sécurité de type « Êtes-vous certain ? » ni de magie de type « Vider la corbeille » pour supprimer une base. Faites attention en utilisant cette commande. Vos données seront effacées sans aucune possibilité de retour-arrière si vous n'avez pas de sauvegarde.

Dans cette section, nous nous sommes penchés sur HTTP et avons posé les bases pour évoquer le reste de l'API CouchDB. Prochain arrêt : les documents !

Documents

Dans CouchDB, le document est la structure de données de base. L'idée derrière la notion de document est, aussi surprenant que cela puisse paraître, un document du monde réel, c'est-à-dire un morceau de papier tel qu'une facture, une recette ou une carte de visite. Nous savons déjà que CouchDB utilise le format JSON pour stocker les documents. Voyons maintenant comment ce stockage fonctionne à bas niveau.

Tout document dans CouchDB a un *ID* (identifiant). Cet identifiant est unique pour une base de données. Vous pouvez choisir l'ID qui vous convient, mais pour de meilleurs résultats, nous vous recommandons un UUID (ou GUID), c'est-à-dire un identifiant universel (ou global) unique [NdT : *Universally (or Globally) Unique IDentifier*]. Les UUID sont des nombres aléatoires avec une faible probabilité de collision et permettent d'en générer des milliers à la minute pour des millions d'années sans doublons. C'est un bon moyen de garantir que deux personnes indépendantes ne peuvent pas créer deux documents différents avec le même identifiant. Pourquoi devriez-vous vous préoccuper de ce que quelqu'un d'autre est en train de faire ? Tout d'abord, cette autre personne pourrait être vous par la suite (plus tard ou sur un autre ordinateur). Ensuite, le mécanisme de réplication de CouchDB vous permet de partager des documents avec autrui et recourir aux UUID garantit que ça va fonctionner. Bref ! Nous reviendrons sur ce sujet par la suite. Pour le moment, créons quelques documents :

```
curl -X PUT http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af -d '{"title":"There is
Nothing Left to Lose","artist":"Foo Fighters"}'
```

CouchDB répond :

```
{"ok":true,"id":"6e1295ed6c29495e54cc05947f18c8af","rev":"1-2902191555"}
```

La commande `curl` peut apparaître complexe, alors expliquons-la. Tout d'abord, `-X PUT` dit à `curl` de forger une requête de type PUT. S'en suit une URL qui indique l'adresse IP et le port de votre instance de CouchDB. La ressource incluse dans l'URL (`/albums/6e1295ed6c29495e54cc05947f18c8af`) indique la position du document dans notre base de données `albums`. La suite alphanumérique barbare est notre UUID. Cet UUID est donc l'ID du document. Enfin, le paramètre `-d` ordonne à `curl` d'utiliser la suite comme corps de la requête PUT. La chaîne est une structure JSON simple incluant les attributs de titre et d'artiste avec leur valeur respective.

> Si vous ne savez pas générer votre UUID, vous pouvez demander à CouchDB d'en générer un (en fait, c'est ce que nous venons de faire sans vous le montrer). Pour cela, envoyez une requête GET à `/_uuids` :
>
> ```
> curl -X GET http://127.0.0.1:5984/_uuids
> ```
>
> CouchDB répond :
>
> ```
> {"uuids":["6e1295ed6c29495e54cc05947f18c8af"]}
> ```
>
> Voilà un UUID. Si vous en voulez plus, ajoutez le paramètre `?count=10` à la requête HTTP et vous en obtiendrez 10. Bien sûr, vous pouvez préciser le nombre qui vous convient.

Pour s'assurer que CouchDB ne nous ment pas en disant avoir sauvegardé notre document (il n'a pas pour habitude de mentir), tentez de le récupérer avec une requête GET :

```
curl -X GET http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af
```

Nous espérons que vous distinguez les différents éléments de la requête ici. Toute chose dans CouchDB a une adresse, une URI, et vous pouvez utiliser diverses méthodes HTTP pour agir sur ces URIs.

CouchDB répond :

```
{"_id":"6e1295ed6c29495e54cc05947f18c8af","_rev":"1-2902191555","title":"There is Nothing Left
to Lose","artist":"Foo Fighters"}
```

Cela ressemble beaucoup au document que nous avons soumis à CouchDB ; tant mieux ! Mais vous devriez aussi noter que CouchDB a jouté deux champs à votre structure JSON. Le premier est `_id` qui stocke l'UUID que nous avons spécifié. Nous connaissons toujours l'identifiant d'un document, ce qui est très pratique (si le champ est retourné par la vue). Le deuxième est `_rev` qui signifie *numéro de version* [NdT : *revision number* en anglais].

Versions

Pour modifier un document dans CouchDB, vous n'allez pas lui dire de trouver tel document, de sélectionner tel champ et d'insérer telle valeur. Non, vous allez récupérer le document, modifier la structure JSON (ou l'équivalent dans votre langage de programmation) comme bon vous semble et sauvegarder le tout avec un nouveau numéro de version. Chaque modification est identifiée par une nouvelle valeur de `_rev`.

Pour mettre à jour ou supprimer un document, CouchDB exige que vous incluiez le champ `_rev` de la version concernée. Quand CouchDB accepte la modification, il génère un nouveau numéro de version. Ce mécanisme garantit que si quelqu'un d'autre a effectué un changement entre temps, vous devez récupérer la dernière version du document avant de pouvoir soumettre votre mise à jour. CouchDB n'acceptera pas votre requête autrement, puisque vous pourriez altérer ou effacer des données dont vous ignorez l'existence. En un mot comme en cent : celui qui modifie le document en premier a gagné. Voyons ce qui se passe si nous ne fournissons pas le champ `_rev` (ce qui revient à fournir un document obsolète) :

```
curl -X PUT http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af -d '{"title":"There is
Nothing Left to Lose","artist":"Foo Fighters","year":"1997"}'
```

CouchDB répond :

```
{"error":"conflict","reason":"Document update conflict."}
```

Si vous voyez cela, ajoutez le dernier numéro de version à votre document dans la structure JSON :

```
curl -X PUT http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af -d '{"_rev":"1-
2902191555","title":"There is Nothing Left to Lose", "artist":"Foo Fighters","year":"1997"}'
```

Maintenant, vous comprenez pourquoi il était utile que CouchDB renvoie ce `_rev` quand nous avons soumis la première requête. CouchDB répond :

```
{"ok":true,"id":"6e1295ed6c29495e54cc05947f18c8af","rev":"2-2739352689"}
```

CouchDB a accepté votre opération d'écriture et a aussi généré un nouveau numéro de version. Ce numéro correspond à l'empreinte MD5 du document sous sa forme transportable adjointe d'un préfixe `N-`, lequel indique le nombre de fois qu'un document a été mis à jour. C'est très utile pour la réplication. Référez-vous au **Chapitre 17, Gestion des conflits (conflicts.html)** pour de plus amples informations.

Il y a plusieurs raisons pour lesquelles CouchDB utilise ce système de gestion de version, aussi appelé *Multi-Version Concurrency Control (MVCC)*. Chacune est liée à l'autre et nous tenons l'occasion rêvée pour en expliquer certaines.

L'un des aspects du protocole HTTP que CouchDB exploite est qu'il est *sans états*. Qu'est-ce que cela veut dire ? Quand vous échangez avec CouchDB, vous devez *formuler des requêtes*. Pour ce faire, vous devez ouvrir une connexion réseau avec CouchDB, échanger des octets et fermer la connexion. C'est ce qui se passe à chaque fois que vous soumettez une requête. D'autres protocoles permettent d'ouvrir une connexion, d'échanger des octets, de conserver la connexion ouverte, d'échanger à nouveau des octets – qui peuvent dépendre de ceux que vous avez déjà reçus – puis, en fin de compte, fermer la connexion. Maintenir une connexion ouverte pour pouvoir gérer des opérations futures nécessite plus de travail de la part du serveur. Ce qui se fait habituellement, c'est que durant la « vie » d'une connexion, le client conserve une vue cohérente et statique des données du serveur. Aussi, gérer un grand nombre de connexions parallèles induit une charge de travail significative. De leur côté, les connexions HTTP sont traditionnellement courtes et fournir les mêmes garanties s'avère beaucoup plus facile. En conséquence, CouchDB peut gérer beaucoup plus de connexions concurrentes.

Une autre raison pour laquelle CouchDB utilise MVCC est que son modèle est conceptuellement plus simple et donc plus facile à programmer. CouchDB nécessite moins de code pour fonctionner, ce qui est toujours une bonne chose puisque le ratio de source d'erreur par ligne de code est constant.

Le système de version a aussi un impact positif sur les mécanismes de réplication et de stockage, mais nous y reviendrons plus tard dans le livre.

> Le mot de *version* [NdT : en anglais, *version or revision*] peut vous être familier – si vous programmez sans système de gestion de version, laissez tomber ce livre et apprenez à en utiliser un. Utiliser une nouvelle version lors d'une modification de document est semblable à un système de gestion de version, mais il y a une différence notable : CouchDB *ne garantit pas* que les anciennes versions sont conservées.

Passer les documents à la loupe

Maintenant, regardons de plus près nos requêtes de création de document avec le paramètre `-v` de `curl` qui s'était avéré utile pour détailler l'API. C'est aussi une bonne occasion de créer des documents que nous pourrons réutiliser dans nos exemples à venir.

Ajoutons quelques-uns de nos albums de musique préférés. Récupérez un nouvel UUID à partir de la ressource `/_uuids`. Si vous ne vous souvenez plus comment faire, remontez de quelques pages.

```
curl -vX PUT http://127.0.0.1:5984/albums/70b50bfa0a4b3aed1f8aff9e92dc16a0 -d
'{"title":"Blackened Sky","artist":"Biffy Clyro","year":2002}'
```

> Soit dit en passant, si vous en savez davantage sur votre album favori, n'hésitez pas à ajouter de nouvelles propriétés. Et ne vous inquiétez pas de connaître ces mêmes propriétés pour l'ensemble des albums, car le concept de documents sans squelette de CouchDB vous permet de stocker uniquement ce que vous connaissez. Après tout, vous devriez vous détendre et ne pas vous soucier de vos données.

Maintenant, avec le paramètre `-v`, la réponse de CouchDB (tronquée pour ne laisser que ce qui nous importe), ressemble à cela :

```
> PUT /albums/70b50bfa0a4b3aed1f8aff9e92dc16a0 HTTP/1.1
>
< HTTP/1.1 201 Created
< Location: http://127.0.0.1:5984/albums/70b50bfa0a4b3aed1f8aff9e92dc16a0
< Etag: "1-2248288203"
<
{"ok":true,"id":"70b50bfa0a4b3aed1f8aff9e92dc16a0","rev":"1-2248288203"}
```

Nous retrouvons le code d'état `201 Created` dans les en-têtes de la réponse, comme nous l'avons vu précédemment lors de la création d'une base de données. L'en-tête `Location` nous donne l'URI complète vers le nouveau document. Et il y a un nouvel en-tête. Dans la terminologie HTTP, un « Etag » identifie une version spécifique d'une ressource. Dans ce cas, cela identifie la version spécifique (la première) et notre nouveau document. Cela chante doux à vos oreilles ? Oui, conceptuellement, un « Etag » correspond au numéro de version du document dans CouchDB, et cela ne devrait pas vous paraître surprenant que CouchDB utilise ses numéros de versions comme « Etag ». Les « Etag » sont utiles pour les infrastructures d'antémémoire [NdT : *caching* en anglais]. Nous apprendrons à les utiliser dans le **Chapitre 8, fonctions d'affichage (show.html)**.

Pièces jointes

Avec CouchDB, les documents peuvent avoir des pièces jointes tout comme un courriel le peut. Une pièce jointe est identifiée par un nom et spécifie son type MIME (ou *Content-Type*) ainsi que le nombre d'octets qui le compose. Les pièces jointes peuvent être de n'importe quel type de données. Il est plus facile de se représenter les pièces jointes comme des fichiers liés à un document. Ces fichiers peuvent être du texte, des images, des documents Word [NdT : plutôt Open Office ;)], de la musique ou des films. Créons-en un :

Les pièces jointes obtiennent leur propre URL où vous pouvez télécharger les données. Disons que vous voulez adjoindre la jaquette de l'album au document `6e1295ed6c29495e54cc05947f18c8af` (« There is Nothing Left to Lose »), et disons qu'elle se trouve dans un fichier `artwork.jpg` du répertoire courant :

```
> curl -vX PUT http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af/ artwork.jpg?rev=2-2739352689 --data-binary @artwork.jpg -H "Content-Type: image/jpg"
```

Le paramètre `-d@` dit à `curl` de transmettre le contenu du fichier dans le corps de la requête HTTP. Nous utilisons le paramètre `-H` pour indiquer à CouchDB que nous envoyons un fichier JPEG. CouchDB va conserver cette information et enverra l'en-tête approprié quand nous récupérerons cette pièce jointe. Dans le cas d'une image JPEG, le navigateur va l'afficher plutôt que vous inviter à l'enregistrer sur votre disque. Cela s'avèrera utile par la suite. Notez aussi que vous devez fournir le numéro de version du document auquel vous attachez la pièce jointe, puisque joindre une pièce, c'est modifier le document.

Vous devriez maintenant voir la jaquette si vous pointez votre navigateur sur `http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af/artwork.jpg`.

Si vous rapatriez à nouveau le document, vous verrez un nouveau champ :

```
curl http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af
```

CouchDB répond :

```
{"_id":"6e1295ed6c29495e54cc05947f18c8af","_rev":"3-131533518","title": "There is Nothing Left to Lose","artist":"Foo Fighters","year":"1997","_attachments":{"artwork.jpg": {"stub":true,"content_type":"image/jpg","length":52450}}}
```

`_attachments` est une liste de clés et de valeurs où les valeurs sont des objets JSON contenant les métadonnées des pièces jointes. `stub=true` nous précise que l'entrée ne contient que les métadonnées. Si nous utilisions l'option HTTP `?attachments=true` dans notre requête, nous obtiendrions une chaîne encodée en base 64 contenant les données de la pièce jointe.

Nous aurons l'occasion de découvrir d'autres options de requêtes comme nous découvrirons d'autres fonctionnalités de CouchDB telles que la réplication que nous allons aborder maintenant.

Réplication

Le mécanisme de réplication de CouchDB permet de synchroniser des bases de données. Tout comme `rsync` synchronise deux répertoires locaux ou sur un réseau, la réplication synchronise deux bases locales ou distantes.

Dans une requête de type POST, vous indiquez à CouchDB la *source* et la *destination* [NdT : *target* dans le code] d'une réplication. CouchDB va trouver quels documents et quelles nouvelles versions se trouvent dans la *source* et sont absentes de la base de *destination* pour les transférer.

Nous nous attarderons davantage sur les détails de la réplication dans un autre chapitre ; ici, nous allons simplement vous montrer comment vous en servir.

En premier lieu, nous allons créer une base de données. Notez que CouchDB ne créera pas automatiquement la base de données de destination et renverra une erreur de réplication si elle n'existe pas (il en va de même pour la source, mais c'est une erreur plus difficile à commettre).

```
curl -X PUT http://127.0.0.1:5984/albums-replica
```

Désormais, nous pouvons utiliser la base `albums-replica` comme destination de la réplication :

```
curl -vX POST http://127.0.0.1:5984/_replicate -d '{"source":"albums","target":"albums-
replica"}'
```

> Dès la version 0.11, CouchDB accepte l'option `"create_target":true` dans la chaîne JSON envoyée en POST à l'URL `_replicate`. Il créera ainsi automatiquement la base de données destinatrice si elle n'existe pas.

CouchDB répond (nous avons formaté la sortie pour la lire plus facilement) :

```
{
  "history": [
    {
      "start_last_seq": 0,
      "missing_found": 2,
      "docs_read": 2,
      "end_last_seq": 5,
      "missing_checked": 2,
      "docs_written": 2,
      "doc_write_failures": 0,
      "end_time": "Sat, 11 Jul 2009 17:36:21 GMT",
      "start_time": "Sat, 11 Jul 2009 17:36:20 GMT"
    }
  ],
  "source_last_seq": 5,
  "session_id": "924e75e914392343de89c99d29d06671",
  "ok": true
}
```

CouchDB conserve un *historique des sessions de réplication*. La réponse à une requête de réplication contient cet historique pour la *session de réplication*. Il est aussi important de noter que la connexion portant la requête de réplication demeurera *ouverte* jusqu'à ce que la réplication soit achevée. Si vous avez de nombreux documents, l'opération prendra du temps et vous n'obtiendrez pas de réponse tant que tous les documents ne seront pas répliqués. Un autre point important à relever est que le mécanisme de réplication réplique uniquement les modifications qui avaient eu lieu lors de la réception de la requête. Aussi, tout ajout ou modification ultérieure au démarrage de la réplication ne sera pas répliqué.

Nous attirons aussi votre attention sur le `"ok": true` à la fin qui nous signale que tout s'est bien passé. Si vous consultez la base `albums-replica`, vous devriez voir tous les documents que vous avez créés dans la base `albums`. Sympa, hein ?

Ce que vous venez de faire s'appelle une *réplication locale* dans la terminologie de CouchDB. Vous avez créé une copie locale de votre base de données. C'est pratique pour les sauvegardes ou pour conserver un instantané [NdT : en anglais, *snapshot*] de votre base à une date donnée ou dans un état que vous désirez pouvoir retrouver plus tard. Vous pouvez être intéressé par cette solution si vous développez une application et que vous voulez pouvoir revenir en arrière, à un état stable de votre code et de vos données.

Il y a d'autres types de réplication utiles dans d'autres cas. Les éléments `source` et `destination` de la requête de réplication sont en fait des liens (comme en HTML). Pour lors, nous nous sommes limités à des liens relatifs à notre instance de CouchDB (donc *locaux*). Vous pouvez aussi indiquer une base de données distante :

```
curl -vX POST http://127.0.0.1:5984/_replicate -d
'{"source":"albums","target":"http://127.0.0.1:5984/albums-replica"}'
```

Utiliser une base *source* locale et une base *destinataire* distante est appelé une *réplication propulsée* [NdT : *pushed replication* en anglais]. Nous propulsons les changements vers un serveur distant.

> Puisque nous n'avons pas encore de deuxième serveur CouchDB disponible pour le moment, nous allons simplement utiliser l'adresse complète (absolue) de notre serveur, mais vous devriez pouvoir faire de même avec un serveur distant.

C'est parfait pour partager des modifications locales avec des serveurs distants ou des voisins.

Vous pouvez aussi utiliser une *source* distante et une *destination* locale pour faire une *réplication tractée*. C'est parfait pour récupérer les dernières mises à jour d'un serveur utilisé par d'autres.

```
curl -vX POST http://127.0.0.1:5984/_replicate -d '{"source":"http://127.0.0.1:5984/albums-
replica","target":"albums"}'
```

Enfin, vous pouvez exécuter une *réplication à distance*, ce qui s'avère utile pour les opérations de gestion :

```
curl -vX POST http://127.0.0.1:5984/_replicate -d
'{"source":"http://127.0.0.1:5984/albums","target":"http://127.0.0.1:5984/albums-replica"}'
```

> **CouchDB et REST**
>
> CouchDB se vante d'avoir une *API qui adhère au modèle REST*, mais ces requêtes de réplication ne semblent pas être vraiment de type REST pour l'œil exercé. Qu'en est-il vraiment ? Si les parties de l'API liées à la base de données, aux documents et aux pièces jointes se conforment à REST, ce n'est pas le cas de toute l'API de CouchDB. L'API de réplication est un exemple parmi d'autres ; nous les verrons dans la suite de l'ouvrage.
>
> Pourquoi mélangeons-nous les API typées REST ou non REST ? Est-ce que les développeurs sont trop faignants pour se conformer partout au REST ? Rappelez-vous que REST est un type d'architecture qui se prête à certaines architectures (comme la partie de l'API qui traite les documents), mais que ce n'est pas une panacée. Déclencher un évènement comme une réplication n'a pas de sens dans l'univers REST. Cela ressemble davantage à un appel de procédure distante. Et cette différence n'est en aucun cas problématique.
>
> Nous croyons plutôt en l'adage qui veut utiliser le bon outil pour les bonnes tâches, et REST ne peut pas répondre à tous les besoins. Nous nous rassurons par les propos de Leonard Richardson et de Sam Ruby qui partagent ce point de vue dans ***RESTful Web Services*** **(http://oreilly.com/catalog/9780596529260)** (publié par O'Reilly).

Résumé

Ce n'est toujours pas toute l'API de CouchDB, mais nous en avons vu l'essentiel en détail. Nous reviendrons sur les détails passés ici sous silence au fil des besoins. Pour l'instant, nous vous jugeons prêt à bâtir des applications avec CouchDB.