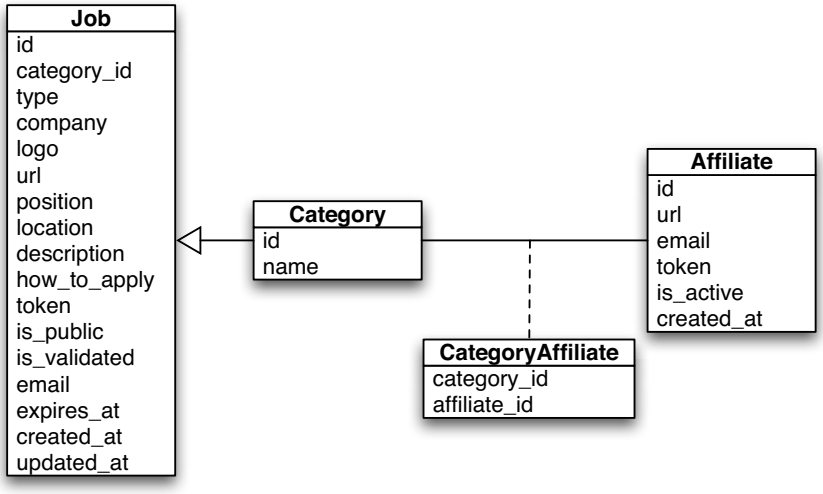
chapitre 3


Job
id
category_id
type
company
logo
url
position
location
description
how_to_apply
token
is_public
is_validated
email
expires_at
created_at
updated_at
Category
id
name
Affiliate
id
url
email
token
is_active
created_at
CategoryAffiliate
category_id
affiliate_id

Concevoir le modèle de données

La base de données est l'un des piliers de toute application web mais sa nature et sa structure peuvent rendre difficile son intégration dans le développement de l'application. Heureusement, Symfony en simplifie la manipulation grâce à la couche d'ORM embarquée Doctrine, qui automatise la création de la base de données à partir d'un schéma de description et de quelques fichiers de données initiales.

MOTS-CLÉS :

- L'ORM Doctrine
- Base de données
- Programmation orientée objet

REMARQUE Parcimonie du code écrit

Symfony réalise une majeure partie du travail à la place du développeur ; le module web ainsi créé sera entièrement fonctionnel sans que vous ayez à écrire beaucoup de code PHP.

Ce troisième chapitre se consacre principalement à la base de données de Jobeet. Les notions de modèle de données, de couche d'abstraction de bases de données, de librairie d'ORM ou bien encore de génération de code seront abordées. Enfin, le tout premier module fonctionnel de l'application sera développé malgré le peu de code que nous aurons à écrire.

Installer la base de données

Le framework Symfony supporte toutes les bases de données compatibles avec PDO (MySQL, PostgreSQL, SQLite, Oracle, MSSQL…). PDO est la couche native d'abstraction de bases de données de PHP. Dans le cadre de ce projet, c'est MySQL qui a été choisi.

Créer la base de données MySQL

La première étape consiste bien évidemment à créer une base de données locale dans laquelle seront sauvegardées et récupérées les données de Jobeet. Pour ce faire, la commande `mysqladmin` suffit amplement, mais un outil graphique comme `PHPMyAdmin` ou bien `MySQL Query Browser` fait aussi très bien l'affaire.

```
$ mysqladmin –uroot –pmYsEcret create jobeet
```

Le parti pris d'utiliser MySQL pour Jobeet tient juste dans le fait que c'est le plus connu et le plus accessible pour tous. Un autre moteur de base de données aurait pu être choisi à la place dans la mesure où le code SQL sera automatiquement généré par l'ORM. C'est ce dernier qui se préoccupe d'écrire les bonnes requêtes SQL pour le moteur de base de données installé.

Configurer la base de données pour le projet Symfony

Maintenant que la base de données est créée, il faut spécifier à Symfony sa configuration afin qu'elle puisse être reliée à l'application via l'ORM. La commande `configure:database` configure Symfony pour fonctionner avec la base de données :

```
$ php symfony configure:database --name=doctrine
  ➥ --class=sfDoctrineDatabase
  ➥ "mysql:host=localhost;dbname=jobeet" root mYsEcret
```

Après avoir configuré la connexion à la base de données, toutes les connexions qui référencent Propel dans le fichier `config/databases.yml`

doivent être supprimées manuellement. Le fichier de configuration de la base de données doit finalement ressembler à celui-ci :

```
all:
  doctrine:
    class: sfDoctrineDatabase
    param:
      dsn: 'mysql:host=localhost;dbname=jobeet'
      username: root
      password: mYsEcret
```

La tâche `configure:database` accepte trois arguments : le DSN (Data Set Name, lien vers la base de données) PDO, le nom d'utilisateur et le mot de passe permettant d'accéder à la base de données. Si aucun mot de passe n'est requis pour accéder à la base de données du serveur de développement, le troisième argument peut être omis.

REMARQUE **Fichier databases.yml**

La tâche `configure:database` stocke la configuration de la base de données dans le fichier de configuration `config/databases.yml`. Au lieu d'utiliser cette tâche, le fichier peut être édité manuellement.

Présentation de la couche d'ORM Doctrine

Symfony fournit de base deux bibliothèques d'ORM Open-Source pour interagir avec les bases de données : Propel et Doctrine, agissant toutes deux comme des couches d'abstraction. Cependant, cet ouvrage ne s'intéresse qu'à l'utilisation de Symfony avec l'ORM Doctrine.

CHOIX DE CONCEPTION **Pourquoi Doctrine plutôt que Propel ?**

Le choix de la librairie Doctrine par rapport à Propel s'impose de lui-même pour plusieurs raisons. Bien que la librairie Propel soit aujourd'hui mature, il n'en résulte pas moins que son âge lui fait défaut. En effet, ce projet Open-Source rendit bien des services aux développeurs jusqu'à aujourd'hui, mais malheureusement son support et sa communauté ne sont plus aussi actifs qu'auparavant. Propel est clairement sur le point de mourir, laissant place à des outils plus récents comme Doctrine.

La librairie Doctrine dispose de plusieurs atouts par rapport à Propel tels qu'une API simple et fluide pour définir des requêtes SQL, de meilleures performances avec les requêtes complexes, la gestion native des migrations, la validation des données, l'héritage de tables ou bien encore le support de différents comportements utiles (« sluggification », ensembles imbriqués, suppressions virtuelles, recherches…).

De surcroît, le projet Doctrine jouit aujourd'hui d'une communauté toujours plus active et d'une documentation abondante. D'ailleurs, à l'heure où nous écrivons ces lignes, un livre contenant toute la documentation technique de Doctrine est en préparation.

Enfin, le projet Doctrine est supporté par Sensio Labs, société éditrice du framework Symfony. Le développeur principal du projet Doctrine, Jonathan Wage, a rejoint l'équipe de production de la société en 2008 pour se consacrer davantage au développement et à l'intégration de Doctrine dans Symfony.

Qu'est-ce qu'une couche d'abstraction de base de données ?

Une couche d'abstraction de bases de données est une interface logicielle qui permet de rendre indépendant le système de gestion de base de données de l'application. Ainsi, une application fonctionnant sur un système de base de données relationnel (SGBDR), comme MySQL, doit pouvoir fonctionner de la même manière avec un système de base de données différent (Oracle par exemple) sans avoir à modifier son code fonctionnel. Une simple ligne de configuration dans un fichier doit suffire à indiquer que le gestionnaire de base de données n'est plus le même.

Depuis la version 5.1.0, PHP dispose de sa propre couche d'accès aux bases de données : PDO. PDO est l'abréviation pour « PHP Data Objects ». Il s'agit en fait surtout d'une interface commune d'accès aux bases de données plus qu'une véritable couche d'abstraction. En effet, avec PDO, il est nécessaire d'écrire soi-même les requêtes SQL pour interroger la base de données à laquelle l'application est connectée. Or, les requêtes sont largement dépendantes du système de base de données, bien que SQL soit un langage standard et normalisé. Chaque SGBDR propose en réalité ses propres fonctionnalités, et donc sa propre version enrichie de SQL pour interroger la base de données.

Doctrine s'appuie sur l'extension PDO pour tout ce qui concerne la connexion et l'interrogation des bases de données (requêtes préparées, transactions, ensembles de résultats…). En revanche, l'API se charge de convertir les requêtes SQL pour le système de gestion de base de données actif, ce qui en fait une véritable couche d'abstraction.

La librairie Doctrine supporte toutes les bases de données compatibles avec PDO telles que MySQL, PostgreSQL, SQLite, Oracle, MSSQL, Sybase, IBM DB2, IBM Informix…

Qu'est-ce qu'un ORM ?

ORM est le sigle de « Object-Relational Mapping » ou « Mapping Objet Relationnel » en français. Une couche d'ORM est une interface logicielle qui permet de représenter et de manipuler sous forme d'objet tous les éléments qui composent une base de données relationnelle.

Ainsi, une table ou bien un enregistrement de celle-ci est perçu comme un objet du langage sur lequel il est possible d'appliquer des actions (les méthodes). L'avantage de cette approche est de s'abstraire complètement de la technologie de gestion de la base de données qui fonctionne en arrière-plan, et de ne travailler qu'avec des objets ayant des liaisons entre eux.

À partir d'un modèle de données défini plus loin dans ce chapitre, Doctrine construit entièrement la base de données pour le SGBDR choisi,

ainsi que les classes permettant d'interroger la base de données au travers d'objets. Grâce aux relations qui lient les tables entre elles, Doctrine est par exemple capable de retrouver tous les enregistrements d'une table qui dépendent d'un autre dans une seconde table.

Activer l'ORM Doctrine pour Symfony

Les deux ORMs du framework, Propel et Doctrine, sont tous deux fournis nativement sous forme de plug-ins internes. À ce jour, Propel est encore la couche d'ORM activée par défaut dans la configuration d'un projet Symfony. Il faut donc commencer par le désactiver, puis activer le plug-in `sfDoctrinePlugin` qui contient toute la bibliothèque Doctrine. La manipulation est triviale puisqu'elle ne nécessite qu'une unique modification dans le fichier de configuration générale du projet `config/ProjectConfiguration.class.php` comme le montre le code suivant :

```
public function setup()
{
  $this->enablePlugins(array('sfDoctrinePlugin'));
  $this->disablePlugins(array('sfPropelPlugin'));
}
```

Le même résultat peut également être obtenu en une seule ligne de code. Le listing ci-dessous active par défaut tous les plug-ins du projet, hormis ceux spécifiés dans le tableau passé en paramètre.

```
public function setup()
{
  $this->enableAllPluginsExcept(array('sfPropelPlugin',
'sfCompat10Plugin'));
}
```

L'une ou l'autre de ces deux opérations nécessite de vider le cache du projet Symfony.

```
$ php symfony cache:clear
```

D'autre part, certains plug-ins comme `sfDoctrinePlugin` embarquent des ressources supplémentaires qui doivent être accessibles depuis un navigateur web comme des images, des feuilles de style ou bien encore des fichiers JavaScript. Lorsqu'un plug-in est nouvellement installé et activé, l'exécution de la tâche `plugin:publish-assets` crée les liens symboliques qui permettent de publier toutes les ressources web nécessaires.

```
$ php symfony plugin:publish-assets
```

Comme Propel n'est pas utilisé pour ce projet Jobeet, le lien symbolique vers le répertoire `web/sfPropelPlugin` qui subsiste peut être supprimé en toute sécurité.

```
$ rm web/sfPropelPlugin
```

Il est temps à présent de s'intéresser à l'architecture de la base de données qui accueille toutes les données de Jobeet.

Concevoir le modèle de données

Le chapitre précédent a décrit les cas d'utilisation de l'application Jobeet ainsi que tous les composants nécessaires : les offres d'emploi, les affiliations et les catégories. Néanmoins, cette définition des besoins fonctionnels n'est pas suffisante. Une transposition sous forme d'un diagramme UML apporte plus de visibilité sur chaque objet et les relations qui les lient les uns aux autres.

Découvrir le diagramme UML « entité-relation »

D'après l'étude des besoins fonctionnels de Jobeet, on détermine clairement les différentes relations suivantes :

- une offre d'emploi est associée à une catégorie ;
- une catégorie a entre 0 et N offres d'emploi associées ;
- une affiliation possède entre 1 et N catégories ;
- une catégorie a entre 0 et N affiliations.

Il en résulte presque naturellement le modèle « entité-relation » suivant.

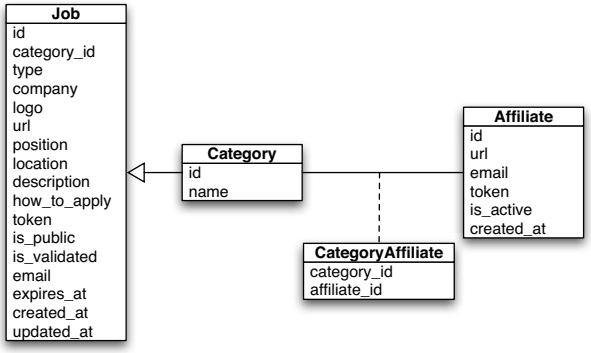


Figure 3–1
Diagramme entité-relation de Jobeet

Ce diagramme décrit également les différentes propriétés de chaque objet qui deviendront les noms des colonnes de chaque table de la base

de données. Un champ `created_at` a été ajouté à quelques tables. Symfony reconnaît ce type de champ et fixe sa valeur avec la date courante du serveur lorsqu'un enregistrement est créé. Il en va de même pour les champs `updated_at` : leur valeur est définie à partir de la date courante du serveur lorsque l'enregistrement est mis à jour dans la table.

Mise en place du schéma de définition de la base

De l'importance du schéma de définition de la base de données…

Les offres d'emploi, les affiliations et les catégories doivent être stockées dans la base de données relationnelle installée plus haut. Symfony est un framework qui a la particularité d'être entièrement orienté Objet, ce qui permet au développeur de manipuler des objets aussi souvent que possible. Par exemple, au lieu d'écrire des requêtes SQL pour retrouver des enregistrements de la base de données, il sera plus naturel et logique de manipuler des objets.

Dans Symfony, les informations de la base de données relationnelle sont représentées (« mappées » en langage informatique) en un modèle objet. La génération et la gestion de modèles objets sont entièrement laissées à la charge de l'ORM Doctrine. Pour ce faire, Doctrine a besoin d'une description des tables et de leurs relations pour créer toutes les classes correspondantes. Il existe deux manières pour établir ce schéma de description. La première consiste à analyser une base de données existante par rétro-ingénierie (*reverse engineering* pour les puristes) ou bien en le créant manuellement.

Écrire le schéma de définition de la base de données

Comme la base de données n'existe pas encore et que nous souhaitons la garder agnostique, le schéma de définition de la base de données doit être écrit à la main dans le fichier `config/doctrine/schema.yml`. Le répertoire `config/doctrine/` n'existe pas encore, il doit être créé à la main.

```
$ mkdir config/doctrine
$ touch config/doctrine/schema.yml
```

Le fichier `config/doctrine/schema.yml` contient une définition au format YAML de tous les éléments qui composent la base de données. Cette description indique la structure de chaque table et de ses champs respectifs, les contraintes appliquées sur chacun d'eux ainsi que toutes les relations qui les lient les unes aux autres. Le code suivant correspond au schéma de définition de la base de données de l'application Jobeet.

Contenu du fichier config/doctrine/schema.yml

```
JobeetCategory:
  actAs: { Timestampable: ~ }
  columns:
    name: { type: string(255), notnull: true, unique: true }

JobeetJob:
  actAs: { Timestampable: ~ }
  columns:
    category_id: { type: integer, notnull: true }
    type:         { type: string(255) }
    company:      { type: string(255), notnull: true }
    logo:         { type: string(255) }
    url:          { type: string(255) }
    position:     { type: string(255), notnull: true }
    location:     { type: string(255), notnull: true }
    description:  { type: string(4000), notnull: true }
    how_to_apply: { type: string(4000), notnull: true }
    token:        { type: string(255), notnull: true,
                    ➥ unique: true }
    is_public:    { type: boolean, notnull: true, default: 1 }
    is_activated: { type: boolean, notnull: true, default: 0 }
    email:        { type: string(255), notnull: true }
    expires_at:   { type: timestamp, notnull: true }
  relations:
    JobeetCategory: { local: category_id, foreign: id,
foreignAlias: JobeetJobs }

JobeetAffiliate:
  actAs: { Timestampable: ~ }
  columns:
    url:       { type: string(255), notnull: true }
    email:     { type: string(255), notnull: true,
                 ➥ unique: true }
    token:     { type: string(255), notnull: true }
    is_active: { type: boolean, notnull: true, default: 0 }
  relations:
    JobeetCategories:
      class: JobeetCategory
      refClass: JobeetCategoryAffiliate
      local: affiliate_id
      foreign: category_id
      foreignAlias: JobeetAffiliates

JobeetCategoryAffiliate:
  columns:
    category_id: { type: integer, primary: true }
    affiliate_id: { type: integer, primary: true }
  relations:
    JobeetCategory: { onDelete: CASCADE, local: category_id,
                      ➥ foreign: id }
    JobeetAffiliate: { onDelete: CASCADE, local: affiliate_id,
                       ➥ foreign: id }
```

Le schéma est la traduction directe, en format YAML, du diagramme « entité-relation ». On identifie ici clairement quatre entités dans ce modèle : `JobeetCategory`, `JobeetJob`, `JobeetAffiliate` et `JobeetCategoryAffiliate`. Les relations entre les objets `JobeetCategory` et `JobeetJob` découvertes plus haut sont bien retranscrites au même titre que celles entre `JobeetCategory` et `JobeetAffiliate` via la quatrième entité `JobeetCategoryAffiliate`.

De plus, ce modèle définit des comportements pour certaines entités grâce à la section `actAs`. Les comportements (*behaviors* en anglais) sont des outils internes de Doctrine qui permettent d'automatiser des traitements sur les données lorsqu'elles sont écrites dans la base. Ici, le comportement `Timestampable` permet de créer et de fixer les valeurs des champs `created_at` et `updated_at` à la volée par Doctrine.

Si les tables de la base de données sont directement créées grâce aux requêtes SQL ou bien à l'aide d'un éditeur graphique, le fichier de configuration `schema.yml` correspondant peut être construit en exécutant la tâche `doctrine:build-schema`.

```
$ php symfony doctrine:build-schema
```

FORMAT YAML pour la sérialisation des données

D'après le site officiel de YAML, YAML est « un standard de sérialisation des données, facile à utiliser pour un être humain quel que soit le langage de programmation ».
En d'autres termes, YAML est un langage simple pour décrire des données (chaîne de caractères, entiers, dates, tableaux ou tableaux associatifs).
En YAML, la structure est présentée grâce à l'indentation. Les listes d'éléments sont identifiées par un tiret, et les paires clé/valeur d'une section par une virgule. YAML dispose égalementd'une syntaxe raccourcie pour décrire la même structure en moins de lignes. Les tableaux sont explicitement identifiés par des crochets [] et les tableaux associatifs avec des accolades {}.
Si vous n'êtes pas familier avec YAML, c'est le moment de commencer à vous y intéresser dans la mesure où le framework Symfony l'emploie excessivement pour ses fichiers de configuration.
Il y a enfin une chose importante dont il faut absolument se souvenir lorsque l'on édite un fichier YAML : l'indentation doit toujours être composée d'un ou de plusieurs espaces, mais jamais de tabulations.

Déclaration des attributs des colonnes d'une table en format YAML

Le fichier `schema.yml` contient la description de toutes les tables et de leurs colonnes. Chaque colonne est décrite au moyen des attributs suivants :

1. `type` : le type de la colonne (`float`, `decimal`, `string`, `array`, `object`, `blob`, `clob`, `timestamp`, `time`, `date`, `enum`, `gzip`) ;
2. `notnull` : placé à la valeur `true`, cet attribut rend la colonne obligatoire ;

À PROPOS Comportements supportés par Doctrine

L'attribut `onDelete` détermine le comportement `ON DELETE` des clés étrangères. Doctrine supporte les comportements `CASCADE`, `SET NULL` et `RESTRICT`. Par exemple, lorsqu'un enregistrement de la table `job` est supprimé, tous les enregistrements associés à la table `jobeet_category_affiliate` seront automatiquement effacés de la base de données.

3 `unique` : placé à la valeur `true`, l'attribut `unique` crée automatiquement un index d'unicité sur la colonne.

La base de données existe et est configurée pour fonctionner avec Symfony, et le schéma de description de cette dernière est désormais écrit. Néanmoins, la base de données est toujours vierge et rien ne permet pour le moment de la manipuler. La partie suivante couvre ces problématiques en présentant comment l'ORM Doctrine génère tout le nécessaire pour rendre la base de données opérationnelle.

Générer la base de données et les classes du modèle avec Doctrine

Grâce à la description de la base de données présente dans le fichier `config/doctrine/schema.yml`, Doctrine est capable de générer les ordres SQL nécessaires pour créer les tables de la base de données ainsi que toutes les classes PHP qui permettent de l'attaquer au travers d'objets métiers.

Construire la base de données automatiquement

La construction de la base de données est réalisée en trois temps : la génération des classes du modèle de données, puis la génération du fichier contenant toutes les requêtes SQL à exécuter, et enfin l'exécution de ce dernier pour créer physiquement toutes les tables.

La première étape consiste tout d'abord à générer le modèle de données, autrement dit les classes PHP relatives à chaque table et enregistrement de la base de données.

```
$ php symfony doctrine:build-model
```

Cette commande génère un ensemble de fichiers PHP dans le répertoire `lib/model/doctrine` qui correspondent à une entité du schéma de définition de la base de données.

Lorsque toutes les classes du modèle de données sont prêtes, l'étape suivante doit permettre de générer tous les scripts SQL qui créent physiquement les tables dans la base de données. Cette fois encore, Symfony facilite grandement le travail du développeur grâce à la tâche automatique `doctrine:build-sql` qu'il suffit d'exécuter.

```
$ php symfony doctrine:build-sql
```

La tâche `doctrine:build-sql` crée les requêtes SQL dans le répertoire `data/sql/`, optimisées pour le moteur de base de données configuré :

Échantillon de code du fichier data/sql/schema.sql

```
CREATE TABLE jobeet_category (id BIGINT AUTO_INCREMENT, name VARCHAR(255)
NOT NULL COMMENT 'test', created_at DATETIME, updated_at DATETIME, slug
VARCHAR(255), UNIQUE INDEX sluggable_idx (slug), PRIMARY KEY(id))
ENGINE = INNODB;
```

Enfin, il ne reste plus que la dernière étape à franchir. Il s'agit de créer physiquement toutes les tables dans la base de données. Une fois de plus, c'est un jeu d'enfant grâce aux tâches automatiques fournies par le framework. Le plug-in `sfDoctrinePlugin` comporte une tâche `doctrine:insert-sql` qui se charge d'exécuter le script SQL généré précédemment pour monter toute la base de données.

```
$ php symfony doctrine:insert-sql
```

Ça y est, la base de données est prête à accueillir des informations. Toutes les tables ont été créées ainsi que les contraintes d'intégrité référentielle qui lient les enregistrements des tables entre eux. Il est désormais temps de s'intéresser aux classes du modèle qui ont été générées.

À PROPOS Aide sur les tâches automatiques

Comme n'importe quel outil en ligne de commande, les tâches automatiques de Symfony peuvent prendre des arguments et des options. Chaque tâche est livrée avec un manuel d'utilisation qui peut être affiché grâce à la commande `help`.

```
$ php symfony help
  ➥ doctrine:insert-sql
```

Le message d'aide liste tous les arguments et options possibles, donne la valeur par défaut de chacun d'eux, et donne quelques exemples pratiques d'utilisation.

Découvrir les classes du modèle de données

À la première étape de construction de la base de données, les fichiers PHP du modèle de données ont été générés à l'aide de la tâche `doctrine:build-model`. Ces fichiers correspondent aux classes PHP qui transforment les enregistrements d'une table en objets métiers pour l'application.

La tâche `doctrine:build-model` construit les fichiers PHP dans le répertoire `lib/model/doctrine/` qui permettent d'interagir avec la base de données. En parcourant ces derniers, il est important de remarquer que Doctrine génère trois classes par table. Par exemple, pour la table `jobeet_job` :

1. `JobeetJob` : un objet de cette classe représente un seul enregistrement de la table `jobeet_job`. La classe est vide par défaut ;
2. `BaseJobeetJob` : c'est la superclasse de `JobeetJob`. Chaque fois que l'on exécute la tâche `doctrine:build-model`, cette classe est régénérée, c'est pourquoi toutes les personnalisations doivent être écrites dans la classe `JobeetJob` ;
3. `JobeetJobTable` : la classe définit des méthodes qui retournent principalement des collections d'objets `JobeetJob`. Cette classe est vide par défaut.

Les valeurs des colonnes d'un enregistrement sont manipulées à partir d'un objet modèle en utilisant quelques accesseurs (méthodes `get*()`) et mutateurs (méthodes `set*()`) :

```
$job = new JobeetJob();
$job->setPosition('Web developer');
$job->save();

echo $job->getPosition();

$job->delete();
```

Au vu de cette syntaxe, il est évident que manipuler des objets plutôt que des requêtes SQL devient à la fois plus naturel, plus aisé mais aussi plus sécurisé, puisque l'échappement des données est laissé à la charge de l'ORM.

Le modèle de données de Jobeet établit une relation entre les offres d'emploi et les catégories. Au moment de générer les classes du modèle de données, Doctrine a deviné les relations possibles entre les entités et les a reportées fidèlement dans les classes PHP générées. Ainsi, un objet `JobeetJob` dispose de méthodes pour définir ou bien récupérer l'objet `JobeetCategory` qui lui est associé comme le présente l'exemple de code ci-dessous.

```
$category = new JobeetCategory();
$category->setName('Programming');

$job = new JobeetJob();
$job->setCategory($category);
```

Doctrine fonctionne bilatéralement, c'est-à-dire que les liaisons entre les objets sont gérées aussi bien d'un côté que d'un autre. La classe `JobeetJob` possède des méthodes pour agir sur l'objet `JobeetCategory` qui lui est associé mais la classe `JobeetCategory` possède elle aussi des méthodes pour définir les objets `JobeetJob` qui lui appartiennent.

Générer la base de données et le modèle en une seule passe

La tâche `doctrine:build-all` est un raccourci pour les tâches exécutées dans cette section et bien d'autres. Il est temps maintenant de générer les formulaires et les validateurs pour les classes de modèle de Jobeet.

```
$ php symfony doctrine:build-all --no-confirmation
```

Les validateurs seront présentés à la fin de ce chapitre, tandis que les formulaires seront expliqués en détail au cours du chapitre 10.

Symfony charge automatiquement les classes PHP à la place du développeur, ce qui signifie que nul appel à `require` n'est requis dans le code.

C'est l'une des innombrables fonctionnalités que le framework automatise, bien que cela entraîne un léger inconvénient. En effet, chaque fois qu'une nouvelle classe est ajoutée au projet, le cache de Symfony doit être vidé. La tâche `doctrine:build-model` a généré un certain nombre de nouvelles classes, c'est pourquoi le cache doit être réinitialisé.

```
$ php symfony cache:clear
```

Le développement d'un projet est fortement accéléré grâce aux nombreux composants et tâches automatiques que fournit nativement le framework Symfony. Dans le cas présent, la base de données ainsi que toutes les classes PHP du modèle de données ont été mises en place en un temps record. Imaginez le temps qu'il vous aurait fallu pour réaliser tout cela à la main en partant de rien !

Toutefois, il manque encore quelque chose d'essentiel pour pouvoir se lancer pleinement dans le code et le développement des fonctionnalités de Jobeet. Il s'agit bien sûr des données initiales qui permettent à l'application de s'initialiser et d'être testée. La section suivante se consacre pleinement à ce sujet.

ASTUCE Comprendre la syntaxe des tâches automatiques de Symfony

Une tâche Symfony est constituée d'un espace de nom (*namespace* pour les puristes) et d'un nom. Chacun d'eux peut être raccourci tant qu'il n'y a pas d'ambiguïté avec d'autres tâches. Ainsi, les commandes suivantes sont équivalentes à `cache:clear` :

```
$ php symfony cache:cl
$ php symfony ca:c
```

Comme la tâche `cache:clear` est fréquemment utilisée, elle possède une autre abréviation encore plus courte :

```
$ php symfony cc
```

Préparer les données initiales de Jobeet

Les tables ont été créées dans la base de données mais celles-ci sont toujours vides. Il faut donc préparer quelques jeux de données pour remplir les tables de la base de données au fur et à mesure de l'avancée du projet.

Découvrir les différents types de données d'un projet Symfony

Pour n'importe quelle application, il existe trois types de données :

1. *les données initiales* : ce sont les données dont a besoin l'application pour fonctionner. Par exemple, Jobeet requiert quelques catégories. S'il n'y en a pas, personne ne pourra soumettre d'offre d'emploi. Un utilisateur administrateur capable de s'authentifier à l'interface d'administration (*backend* en anglais) est également nécessaire ;
2. *les données de test* : les données de test sont nécessaires pour tester l'application et pour s'assurer qu'elle se comporte comme les cas d'utilisation fonctionnels le spécifient. Bien évidemment, le meilleur moyen de le vérifier est d'écrire des séries de tests automatisés ; c'est pourquoi des tests unitaires et fonctionnels seront développés pour Jobeet. Ainsi, à chaque fois que les tests seront exécutés, une base de données saine constituée de données fraîches et prêtes à être testées sera nécessaire ;

3 *les données utilisateur* : les données utilisateur sont créées par les utilisateurs au cours du cycle de vie normal de l'application.

Pour le moment, Jobeet requiert quelques données initiales pour initialiser l'application. Symfony fournit un moyen simple et efficace de définir ce type de données à l'aide de fichiers YAML, comme l'explique la partie suivante.

Définir des jeux de données initiales pour Jobeet

Chaque fois que Symfony crée les tables dans la base de données, toutes les données sont perdues. Pour peupler la base de données avec des données initiales, nous pourrions créer un script PHP, ou bien exécuter quelques requêtes SQL avec le programme MySQL. Néanmoins, comme ce besoin est relativement fréquent, il existe une meilleure façon de procéder avec Symfony. Il s'agit de créer des fichiers YAML dans le répertoire `data/fixtures/`, puis d'exécuter la tâche `doctrine:data-load` pour les charger en base de données.

Les deux listings suivants de code YAML définissent un jeu de données initiales pour l'application. Le premier fichier déclare les données pour remplir la table `jobeet_category` tandis que le second sert à peupler la table des offres d'emploi `jobeet_job`.

Contenu du fichier data/fixtures/categories.yml

```
JobeetCategory:
  design:
    name: Design
  programming:
    name: Programming
  manager:
    name: Manager
  administrator:
    name: Administrator
```

Contenu du fichier data/fixtures/jobs.yml

```
JobeetJob:
  job_sensio_labs:
    JobeetCategory: programming
    type:         full-time
    company:      Sensio Labs
    logo:         sensio-labs.gif
    url:          http://www.sensiolabs.com/
    position:     Web Developer
    location:     Paris, France
    description: |
      You've already developed websites with Symfony and you
      want to work with Open-Source technologies. You have a
```

```
    minimum of 3 years experience in web development with PHP
    or Java and you wish to participate to development of
    Web 2.0 sites using the best frameworks available.
  how_to_apply: |
    Send your resume to fabien.potencier [at] sensio.com
  is_public:    true
  is_activated: true
  token:        job_sensio_labs
  email:        job@example.com
  expires_at:   '2010-10-10'

job_extreme_sensio:
  JobeetCategory: design
  type:         part-time
  company:      Extreme Sensio
  logo:         extreme-sensio.gif
  url:          http://www.extreme-sensio.com/
  position:     Web Designer
  location:     Paris, France
  description: |
    Lorem ipsum dolor sit amet, consectetur adipisicing elit,
    sed do eiusmod tempor incididunt ut labore et dolore magna
    aliqua. Ut enim ad minim veniam, quis nostrud exercitation
    ullamco laboris nisi ut aliquip ex ea commodo consequat.
    Duis aute irure dolor in reprehenderit in.

    Voluptate velit esse cillum dolore eu fugiat nulla
    pariatur. Excepteur sint occaecat cupidatat non proident,
    sunt in culpa qui officia deserunt mollit anim id est
    laborum.
  how_to_apply: |
    Send your resume to fabien.potencier [at] sensio.com
  is_public:    true
  is_activated: true
  token:        job_extreme_sensio
  email:        job@example.com
  expires_at:   '2010-10-10'
```

Un fichier de données est écrit au format YAML, et décrit les objets modèles référencés par un nom unique. Par exemple, les deux offres d'emploi sont intitulées `job_sensio_labs` et `job_extreme_sensio`. Cet intitulé sert à lier les objets entre eux sans avoir à exprimer explicitement les clés primaires, qui sont dans la plupart des cas auto-incrémentées et qui varient perpétuellement. La catégorie de l'offre d'emploi `job_sensio_labs` est `programming`, ce qui correspond à la catégorie nommée « Programming ».

Dans un fichier YAML, lorsqu'une chaîne de caractères contient des retours à la ligne (comme la colonne `description` dans les données initiales du fichier d'offres d'emploi), la barre verticale (*pipe* en anglais) | sert à indiquer que la chaîne occupera plusieurs lignes.

REMARQUE Télécharger les images relatives aux données initiales

Le fichier de données initiales des offres référence deux images. Vous pouvez les télécharger :

- http://www.symfony-project.org/get/jobeet/sensio-labs.gif
- http://www.symfony-project.org/get/jobeet/extreme-sensio.gif

Vous devrez ensuite les placer dans le répertoire `uploads/jobs/`.

ASTUCE **Propel versus Doctrine**

Propel a besoin que les fichiers de données de test soient préfixés par des nombres pour déterminer dans quel ordre les fichiers doivent être chargés. Avec Doctrine, ce n'est pas nécessaire puisque toutes les données sont chargées et sauvegardées dans le bon ordre pour s'assurer que les clés étrangères sont définies correctement.

Bien qu'un fichier de données contienne des objets provenant d'un ou de plusieurs modèles, il est vivement recommandé de ne créer qu'un seul fichier par modèle.

Dans un fichier de données, il n'est nul besoin de définir toutes les valeurs des colonnes. Si certaines valeurs ne sont pas définies, Symfony utilisera la valeur par défaut définie dans le schéma de la base de données. Comme Symfony utilise Doctrine pour charger les données en base de données, tous les comportements natifs (comme la fixation automatique des colonnes `created_at` et `updated_at`) et les comportements personnalisés ajoutés aux classes de modèle sont activés.

Charger les jeux de données de tests en base de données

Une fois les fichiers de données initiales créés, leur chargement en base de données est aussi simple que de lancer une tâche automatique. Le plug-in `sfDoctrinePlugin` possède la commande `doctrine:data-load` qui se charge d'enregistrer toutes ces données dans la base de données.

```
$ php symfony doctrine:data-load
```

Exécuter l'une après l'autre toutes les tâches pour régénérer la base de données, construire les classes du modèle et insérer les données initiales peut se révéler très vite fastidieux. Symfony propose une tâche simple qui réalise toutes ces opérations en une seule passe comme l'explique la section suivante.

Régénérer la base de données et le modèle en une seule passe

La tâche `doctrine:build-all-reload` est un raccourci pour la `tâche doctrine:build-all` suivi de la tâche `doctrine:data-load`. Celle-ci s'occupe de régénérer toute la base de données et les classes du modèle, puis finit par charger les données initiales dans les tables.

```
$ php symfony doctrine:build-all-reload
```

Il suffit de lancer la commande `doctrine:build-all-reload` puis de s'assurer que tout a bien été généré depuis le schéma. Cette tâche génère les classes de formulaires, de filtres, de modèle, supprime la base de données existante et la recrée avec toutes les tables peuplées par les données initiales.

Profiter de toute la puissance de Symfony dans le navigateur

L'interface en ligne de commande est plutôt pratique mais n'est malgré tout pas très attrayante, qui plus est pour un projet web. À ce stade d'avancement du projet, Jobeet est déjà prêt à accueillir les pages web dynamiques qui interagissent avec la base de données.

La suite du chapitre aborde les fonctionnalités essentielles d'affichage de la liste des offres d'emploi, et d'édition et de suppression d'une offre existante. Comme cela a déjà été expliqué au premier chapitre, un projet Symfony est constitué d'*applications*. Chaque application est ensuite divisée en *modules*.

Un module est un ensemble autonome de code PHP qui représente une fonctionnalité de l'application (le module API par exemple), ou bien un ensemble de manipulations que l'utilisateur peut réaliser sur un objet du modèle (un module d'offres d'emploi par exemple).

Générer le premier module fonctionnel « job »

Le module principal de l'application Jobeet est bien évidemment celui qui permet de créer et de consulter des offres. Le framework Symfony est capable de générer automatiquement un module fonctionnel complet pour un modèle donné. Ce module intègre de base toutes les fonctionnalités de manipulation simples telles que l'ajout, la modification, la suppression et la consultation.

Ce travail est réalisé à l'aide de la commande `doctrine:generate-module` comme le présente le code ci-dessous.

```
$ php symfony doctrine:generate-module --with-show
  ➥ --non-verbose-templates frontend job JobeetJob
```

La tâche `doctrine:generate-module` génère un module `job` dans l'application `frontend` pour le modèle `JobeetJob`. Comme avec la plupart des tâches Symfony, quelques fichiers et répertoires ont été créés. Tous les fichiers du présent module ont été fabriqués sous le répertoire `apps/frontend/modules/job/`.

Composition de base d'un module généré par Symfony

Le tableau ci-après décrit les répertoires de base qui ont été générés par Symfony à l'exécution de la tâche `doctrine:generate-module` dans le répertoire `apps/frontend/modules/job/`.

Tableau 3–1 Répertoires du module apps/frontend/modules/job

Répertoire	Description
`actions/`	Les actions du module
`templates/`	Les templates du module

Le répertoire `actions/` contient la classe dans laquelle se trouvent toutes les actions CRUD (*create*, *retrieve*, *update* et *delete*) de base qui permettent de manipuler une offre d'emploi. À toutes ces actions est associé un ensemble de fichiers de templates générés dans le répertoire `templates/`.

Découvrir les actions du module « job »

Le fichier `actions/actions.class.php` définit toutes les actions possibles pour le module `job`. C'est exactement ce que décrit le tableau 3-2.

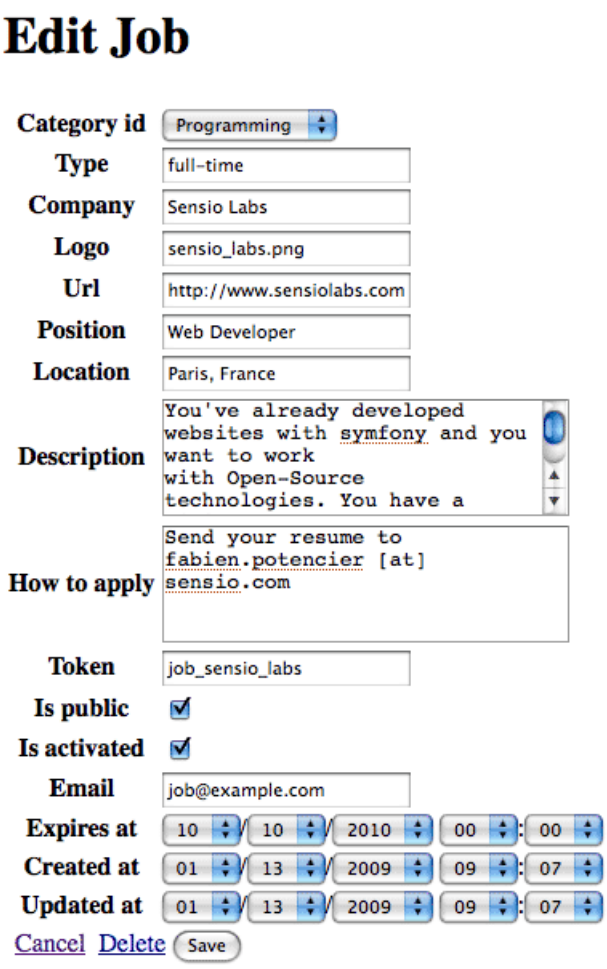


Figure 3–2
Formulaire d'édition d'une offre d'emploi

Tableau 3–2 Liste des actions du fichier apps/frontend/modules/job/actions/actions.class.php

Nom de l'action	Description
`index`	Affiche les enregistrements d'une table
`show`	Affiche les champs et les valeurs d'un enregistrement donné
`new`	Affiche un formulaire pour créer un nouvel enregistrement
`create`	Crée un nouvel enregistrement
`edit`	Affiche un formulaire pour éditer un enregistrement existant
`update`	Met à jour les informations d'un enregistrement d'après les valeurs transmises par l'utilisateur
`delete`	Supprime un enregistrement donné de la table

Le module `job` est désormais accessible et utilisable depuis un navigateur web à l'adresse suivante :

- http://jobeet.localhost/frontend_dev.php/job

Comprendre l'importance de la méthode magique __toString()

Si l'on tente d'éditer une offre d'emploi, on remarque que la liste déroulante « Category id » est une sélection de l'ensemble des catégories présentes dans la base de données. La valeur de chaque option est obtenue grâce à la méthode `__toString()`.

Doctrine essaye d'appeler nativement une méthode `__toString()` en devinant un nom de colonne descriptif tel que `title`, `name`, `subject`, etc. Si l'on désire quelque chose de plus personnalisé, il est alors nécessaire de redéfinir la méthode `__toString()` comme le présente le listing ci-après. Le modèle `JobeetCategory` est capable de deviner la méthode `__toString()` en utilisant la colonne `name` de la table `jobeet_category`.

Listing du fichier lib/model/doctrine/JobeetJob.class.php

```
class JobeetJob extends BaseJobeetJob
{
  public function __toString()
  {
    return sprintf('%s at %s (%s)', $this->getPosition(),
    ➥ $this->getCompany(), $this->getLocation());
  }
}
```

Listing du fichier lib/model/doctrine/JobeetAffiliate.class.php

```
class JobeetAffiliate extends BaseJobeetAffiliate
{
  public function __toString()
  {
    return $this->getUrl();
  }
}
```

Ajouter et éditer les offres d'emploi

Les offres d'emploi sont à présent prêtes à être ajoutées et éditées. Si un champ obligatoire est laissé vide ou bien si sa valeur est incorrecte (une date invalide par exemple), le processus de validation du formulaire provoquera une erreur, empêchant alors la mise à jour de l'enregistrement. Symfony crée effectivement les règles de validation basiques en introspectant le schéma de la base de données.

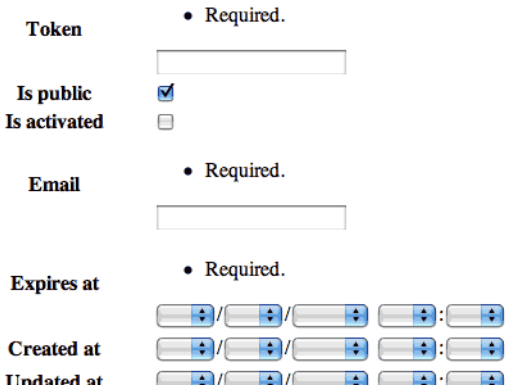


Figure 3–3
Contrôle de saisie basique dans le formulaire de création d'une offre d'emploi.

En résumé…

C'est tout pour ce troisième chapitre. L'introduction était très claire. En effet, peu de code PHP a été écrit mais Jobeet dispose déjà d'un module d'offres d'emploi entièrement fonctionnel et prêt à être amélioré et personnalisé. Souvenez-vous, moins de code PHP signifie aussi moins de risques d'y trouver un bug !

Un moyen simple de progresser avant de passer au chapitre suivant est de prendre la peine de lire le code généré pour le module et le modèle, et essayer d'en comprendre le fonctionnement.

Le chapitre qui suit aborde l'un des plus importants paradigmes utilisés dans les frameworks web : le patron de conception MVC.

Le code complet du chapitre est disponible dans le dépôt SVN de Jobeet au tag `release_day_03` :

```
$ svn co http://svn.jobeet.org/doctrine/tags/release_day_03/
jobeet/
```