

Tutoriel : Comment gérer l'internationalisation via une base de données, avec Spring

par Gildas Cuisinier ([Hikage](#)) ([Blog](#))

Date de publication :

Dernière mise à jour :

Qu'est-ce que l'internationalisation ? Comment implémenter celle-ci en Java ?
En quoi Spring apporte-t-il un plus dans ce domaine ?
Cet article va tenter de répondre à ces trois questions.
Il montrera, grâce à un exemple, la modularité qu'apporte Spring, en permettant de manière transparente pour une application, l'utilisation d'une base de données à la place de fichier properties pour le stockage des messages.
Bonne lecture !

- I - Introduction
- II - Internationalisation et Java
 - II-A - Locale
 - II-B - ResourceBundle
- III - Spring et Internationalisation
- IV - Internationalisation sur base de données
 - IV-A - Schéma de la base de données
 - IV-B - DAO
 - IV-C - Service
 - IV-D - Implémentation de MessageSource
 - IV-E - Exemple d'utilisation
- V - Conclusion
 - V-A - Remerciements
 - V-B - Sources de cet article

I - Introduction

A l'heure où Internet permet à n'importe qui de fournir de l'information au monde entier, la qualité d'un logiciel (ou d'un site web) est aussi définie par le nombre de personnes qui seront capables de l'utiliser.

Cela sous entend donc que son interface graphique soit intuitive et attrayante, mais aussi qu'elle soit disponible en plusieurs langues.

En effet, un logiciel usuellement parfait mais disponible en uniquement en russe ou un breton aura un impact très restreint au niveau mondial.

Bien sur, il serait possible de traduire par la suite celui-ci dans différentes langues, mais pour que cette tâche soit simple (et qu'elle puisse être délégué à un traducteur), il faut que le développement du logiciel ait prévu cela lors de sa conception.

Par exemple, si tout les messages sont intégrés directement dans le code sous forme de constantes, il devient impossible à un traducteur d'effectuer ce travail. Sans parler qu'une version du code par langue devrait être maintenue.

Afin de résoudre cela, un mécanisme d'abstraction des messages existe, communément appelé internationalisation (ou i18n).

II - Internationalisation et Java

Afin d'aider à l'internationalisation d'un logiciel, Java propose un système basique de traduction de messages, et ce par le biais de deux composants : la **Locale** et le **ResourceBundle**

II-A - Locale

La Javadoc nous informe que :

*Un objet **Locale** représente une région géographique, politique ou culturelle spécifique"*

En pratique, une **Locale** est une classe qui se base sur 2 codes, un code **langue** et un code **pays**, afin d'identifier un pays, une langue ou un dialecte.

Le code langue est composé de deux caractères minuscules, dont la liste est définie par l'**ISO-639**. Par exemple, "fr" correspondra à la langue française, tandis que "en" correspondra à la langue anglaise.

Le code pays est lui aussi composé de deux caractères, majuscules cette fois-ci, dont la liste est définie par l'**ISO-3166**. La France correspondra à "FR", la Belgique à "BE" et les Royaume Unis à "GB".

L'association d'un code langue et un code pays permet de définir une variante d'une langue, un dialecte. Ainsi la locale **fr_FR** identifie la langue française parlée en France, tandis que locale **fr_BE** est assimilée à la langue "belge".

Voici comment est utilisé un objet **Locale** :

```
// Locale française
Locale france = new Locale("fr", "FR");

// Locale langue française générale
Locale francais = new Locale("fr");

// Locale belge
Locale belge = new Locale("fr", "BE");
```

Qui plus est, certaines locales sont prédéfinies en tant que constantes de classe. C'est le cas de la langue française (**Locale.FRENCH**), de la locale française (**Locale.FRANCE**) ou de la langue anglaise (**Locale.ENGLISH**).

II-B - ResourceBundle

Le deuxième composant i18n de Java est le **ResourceBundle**. Celui-ci est responsable de la récupération pour une locale **Locale** donnée.

Cependant, le **ResourceBundle** est une classe abstraite, c'est donc une implémentation concrète de **ResourceBundle** qui est utilisée, **PropertyResourceBundle**. Cette implémentation s'appuie sur un nom de fichier properties de base, et va rechercher pour une **Locale** donnée, si une traduction existe.

Et pour cela, elle va vérifier l'existence d'un fichier properties dont le nom est : <nom du fichier de base>_<le code langue>_<le code pays>

Le cas échéant, il va chercher une traduction plus générale, c'est à dire basée uniquement sur la langue, dont le fichier properties aurait comme nom <nom du fichier de base>_<le code langue>.

Et si ce n'est pas suffisant, il va utiliser le fichier de base pour récupérer une traduction.

Par exemple, le code suivant :

```
ResourceBundle bundle = ResourceBundle.getBundle("message");
```

Si la locale courante (qui est par exemple la langue par défaut du système d'exploitation) est fr_FR, voici par ordre les fichiers qui seront recherchés :

- 1 message_fr_FR.properties
- 2 message_fr.properties
- 3 message.properties

Bien sûr, il est possible de récupérer un **ResourceBundle** en spécifiant explicitement la Locale à utiliser :

```
ResourceBundle bundleEn = ResourceBundle.getBundle("message", Locale.ENGLISH);
```

Ensuite, pour récupérer un message internationalisé, il faut utiliser la méthode *getString(String key)* :

```
String message = bundle.getString("HelloWorld");
```

Ici, c'est la traduction d'un message identifié par "Hello World" qui sera récupéré.

III - Spring et Internationalisation

De son côté, Spring possède son propre mécanisme i18n, qui comme à l'accoutumé est configurable via le fichier de configuration xml.

Le composant de base est l'interface `MessageSource` :

```
public interface MessageSource {  
  
    String getMessage(String key, Object[] param, String defaultMessage, Locale locale);  
  
    String getMessage(String key, Object[] param, Locale locale) throws NoSuchMessageException;  
  
    String getMessage(MessageSourceResolvable messageSourceResolvable, Locale locale) throws  
    NoSuchMessageException;
```

Le principe est que lors de l'instanciation d'un **ApplicationContext**, Spring va vérifier l'existence d'un Bean ayant comme identifiant **messageSource** et implémentant cette interface.

Si c'est le cas, alors il sera utilisé pour récupérer les messages internationalisés, le cas contraire, Spring va effectuer une recherche dans le contexte parent, ainsi de suite.

Et c'est tout l'intérêt de Spring par rapport au système Java standard : Sa **modularité**. En effet, il suffit de changer l'implémentation du Bean **messageSource**, pour que les messages soit récupérés indifféremment dans des `ResourceBundles`, dans des fichiers XML, ou encore dans une base de données.

Le tout de manière totalement transparente pour l'application, qui ne sera liée au système que via l'interface, et non directement avec une des implémentations.

Au niveau des implémentations, Spring en fournit une permettant la réutilisation des `ResourceBundles` : **ResourceBundleMessageSource**. Et sa configuration est très simple :

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">  
    <property name="basename" value="message"/>  
</bean>  
  
<!-- Il est possible d'utiliser plusieurs ResourceBundle : -->  
  
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">  
    <property name="basenames">  
        <list>  
            <value>message</value>  
            <value>messageErreur</value>  
        </list>  
    </property>  
</bean>
```

La propriété **basenames** prend en paramètres des variables du même type que celui passé en paramètre dans un **ResourceBundle**. Mais il est possible d'en définir plusieurs d'un coup, ce qui permet d'avoir un seul objet pour la traduction.

Une fois le Bean **MessageSource** configuré, tout Bean possédant une référence à l'`ApplicationContext` (en implémentant `ApplicationContextAware` par exemple) ou à `MessageSource` (en implémentant `MessageSourceAware`) sera capable de traduire des messages :

```
String message = applicationContext.getMessage("helloWorld", null, Locale.FRENCH);
```

Le premier paramètre étant la clé d'identification du message, le deuxième étant un tableau d'objets et le dernier la locale.

Le tableau d'objets permet d'utiliser des paramètres dans les messages. Par exemple, un message d'erreur "L'utilisateur {0} n'existe pas", le {0} est un paramètre qui pourra être défini dynamiquement lors de l'appel :

```
String nomUtilisateur = "Gildas Cuisinier";  
String message = applicationContext.getMessage("erreurUtilisateur", new Object[]{nomUtilisateur},  
Locale.FRENCH);
```

Le résultat de cet appel est la chaîne "L'utilisateur Gildas Cuisinier n'existe pas".

IV - Internationalisation sur base de données

Afin de montrer en pratique cette modularité, nous allons maintenant créer une implémentation qui ira rechercher les traductions dans une base de données.

Cela demande plusieurs étapes :

- 1 La création des tables dans une base de données
- 2 La création d'un DAO de récupération des messages
- 3 La création d'un Service, qui va s'occuper de la logique de traductions
- 4 L'implémentation de l'interface MessageSource

IV-A - Schéma de la base de données

Voici un schéma possible pour stocker les messages :

```
CREATE TABLE messages (  
  `key` varchar(20) NOT NULL,  
  `langue` varchar(2) NOT NULL,  
  `pays` varchar(2) default NULL,  
  `texte` varchar(120) NOT NULL  
);
```

Une clé pour identifier le message, un code langue (obligatoire), un code pays (optionnel) et, bien sûr, la traduction en elle-même.

Rien de bien compliqué en soi.


IV-B - DAO

Une fois le schéma créé, il faut pouvoir y accéder. C'est à un DAO que revient cette tâche.

Notre DAO devra pouvoir récupérer un message soit d'après une clé, un code langue et un code pays, soit d'après une clé et un code langue uniquement.

Voici le contrat que le DAO doit respecter :

```
public interface MessageDao {  
    public String getMessage(String cle, String langue, String pays);  
  
    public String getMessage(String cle, String langue);  
}
```

 *Le détail d'une implémentation n'est pas nécessaire pour comprendre le principe, cependant, une version du DAO se basant sur JdbcTemplate est disponible dans les sources jointes.*

Reste maintenant à configurer ce DAO dans le fichier de configuration Spring :

```
<!-- Définition du DAO -->
<bean id="messageDao" class="com.developpez.hikage.spring.i18n.dao.impl.MessageDaoImpl">
<!-- Configuration des propriétés propre à l'implémentation du DAO -->
</bean>
```

IV-C - Service

Une fois le DAO implémenté, il reste à créer la couche service. Celle-ci est responsable de plusieurs choses : la gestion des transactions et la gestion du métier.

Dans le cadre de cet article, la gestion des transactions est hors sujet, et sera donc ignorée.

C'est au niveau de la couche "métier" que sera gérée la logique de recherche :


- 1 Si un message existe pour une clé, une langue et un pays donné, alors on le renvoie
- 2 Si un message existe pour une clé et une langue donnée, alors on le renvoie,
- 3 Si un message existe pour une clé et la langue par défaut, alors on le renvoie
- 4 Sinon la clé est renvoyée avec un préfixe et un suffixe

Le contrat de la couche service pourrait être défini par cette interface :

```
public interface MessageMngt {
    public String getMessage(String key, String langue, String pays);
}
```

Encore une fois, il est nécessaire de configurer ce service dans le contexte Spring :

```
<!-- Définition du service -->
<bean id="messageMngt" class="com.developpez.hikage.spring.i18n.service.impl.MessageMngtImpl">
    <property name="langueDefault" value="fr"/>
    <property name="messageDao" ref="messageDao"/>
</bean>
```

 Encore une fois, une implémentation est disponible dans les sources du projet

IV-D - Implémentation de MessageSource

Une fois que tous les éléments sont prêts, il reste à créer l'implémentation de **MessageSource** proprement dite.

Il serait tout à fait possible d'implémenter MessageSource directement, mais l'interface possède trois méthodes très similaires, qui ne diffèrent entre elles que par les types de leurs paramètres.

Afin de faciliter le développement, Spring fournit une classe (qui elle même étant MessageSource) qui va nous permettre de ne n'implémenter qu'une seule méthode.

Cette classe est : **AbstractMessageSource** :

```
protected MessageFormat resolveCode(String key, Locale locale);
```

Il suffit donc d'étendre cette classe, et dans le corps de la méthode, faire appel au service pour récupérer le message traduit :

```
public class DatabaseMessageSource extends AbstractMessageSource {  
  
    // Référence à la couche service  
    private MessageMngt messageMngt;  
  
    public void setMessageMngt(MessageMngt messageMngt) {  
        this.messageMngt = messageMngt;  
    }  
  
    protected MessageFormat resolveCode(String key, Locale locale) {  
        // Utilisation du service pour récupérer le message traduit  
        String message = messageMngt.getMessage(key, locale.getLanguage(), locale.getCountry());  
        // Et renvoie de celui-ci sous la bonne forme  
        return createMessageFormat(message, locale);  
    }  
}
```

Il reste maintenant à ajouter cette implémentation dans le contexte Spring, en utilisant bien l'ID **messageSource**. Dans le cas contraire, Spring ne l'utiliserait pas pour l'internationalisation :

```
<bean id="messageSource" class="com.developpez.hikage.spring.i18n.DatabaseMessageSource">  
    <property name="messageMngt" ref="messageMngt"/>  
</bean>
```

IV-E - Exemple d'utilisation

Une fois ces étapes réalisées, il devrait être possible d'utiliser notre mécanisme i18n via les méthodes de MessageSource (disponible via l'ApplicationContext) :

```
ApplicationContext applicationContext = new  
    ClassPathXmlApplicationContext("applicationContext.xml");  
  
System.out.println("HelloWorld en français de France : " +  
    applicationContext.getMessage("helloWorld", null, Locale.FRANCE));  
System.out.println("HelloWorld en Belge : " + applicationContext.getMessage("helloWorld", null, new  
    Locale("fr", "BE")));  
System.out.println("HelloWorld Italie : " + applicationContext.getMessage("helloWorld", null,  
    Locale.ITALY));
```

V - Conclusion

Il reste possible bien évidemment de modifier le service et le DAO afin de pouvoir faire une gestion complète des messages, en ajoutant la possibilité d'ajouter, modifier et supprimer des entrées.

De plus, la lecture en elle-même n'est pas efficace, il serait possible d'utiliser un système de cache en mémoire qui serait réinitialisé sur demande, ce qui permettrait d'améliorer les performances en réduisant le nombre des appels à la base de données.

Bref, le système pourrait être amélioré, mais cela dépasse le cadre de cet article.

V-A - Remerciements

Je remercie N1bus d'avoir pris le temps de corriger mon article, ainsi que djo.mos et Ricky81 pour leurs conseils

V-B - Sources de cet article

Les sources de cet article sont disponibles afin de tester ou d'étudier un cas pratique simple :

Téléchargez les sources ici

