

# Exploration des modèles de Wicket

par Jawher Moussa ([Accueil](#))

Date de publication : 31 Juillet 2008

Dernière mise à jour :

Cet article a pour objectif de vous présenter la notion de modèles du framework **Apache Wicket** et ce à travers un exemple pratique et réel.



I - Introduction.....	3
II - Problématique.....	4
III - Approche via Model.....	5
IV - Approche via PropertyModel.....	6
V - Approche via CompoundPropertyModel.....	8
VI - Approche via chaînage de CompoundPropertyModel et LoadableDetachableModel.....	9
VII - Conclusion.....	10
VIII - Remerciements.....	11
IX - ANNEXE: Page HTML.....	12

## I - Introduction

**Apache Wicket** est un *framework* pour la création d'applications web qui repose presque entièrement sur *Java* et *HTML* comme moyens pour bâtir ses interfaces.

L'une des notions centrales et critiques dans *Wicket* est la notion des **modèles**, et c'est généralement celle qui pose le plus de difficultés lors de sa prise en main.

Dans *Wicket*, un **modèle** est un adaptateur qui adapte les données de la couche métier (ou autres) d'une application aux composants de *Wicket*.

Ils sont indispensables dans la mesure où il n'est pas possible pour les composants de *Wicket* de pouvoir interpréter toutes les formes de données que le programmeur leur passe.

*Wicket* propose plusieurs types de modèles différents, tous implémentant l'interface *IModel*, mais chacun est adapté à un cas d'utilisation particulier.

Voici à quoi ressemble l'interface *IModel* (tiré du code source d'Apache Wicket 1.3.4):

```
public interface IModel extends IDetachable
{
    /**
     * Gets the model object.
     *
     * @return The model object
     */
    Object getObject();

    /**
     * Sets the model object.
     *
     * @param object
     *         The model object
     */
    void setObject(final Object object);
}
```

Il est à noter que la notion des modèles de *Wicket* est inspirée par cette même notion dans *Swing* (*TreeModel*, *ListModel*, etc.), à l'exception que là où *Swing* propose un type de modèle par composant, *Wicket* n'en propose qu'un seul type de base utilisable avec tous les composants.

C'est ainsi que le but de cet article est de présenter avec un exemple pratique comment maîtriser et utiliser au mieux les modèles dans *Wicket*.

Pour des informations d'ordre plus général sur *Wicket*, je vous conseille de consulter l'article "[Redécouvrez le web avec Wicket](#)" par Romain Guy.

## II - Problématique

Dans le cadre de cet article, on suppose disposer d'une entité *Person* représentant une personne, dont voici le code:

```
public class Person {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

C'est un simple *POJO* avec deux propriétés: *firstName* et *lastName* de type chaîne de caractères.

On suppose aussi disposer d'un *DAO (Data Access Object) PersonDao* implémentant l'interface suivante:

```
public interface IPersonDao {
    void insert(Person person);
}
```

La méthode *insert* permet d'ajouter une instance de *Person* à une base de données. Le but dans cet article est de développer une page *Wicket* permettant de saisir les données d'une personne pour l'ajouter dans la base de données en utilisant le DAO décrit ci-dessus.

Pour ce faire, je vais procéder par étapes, en partant de la méthode la plus simple à celle la plus optimisée.

### III - Approche via Model

C'est la méthode la plus triviale et la plus basique, mais surtout la plus lourde à mettre en oeuvre.

```

public class NewPage1 extends WebPage {
    public NewPage1() {
        final IModel firstNameModel = new Model();
        final IModel lastNameModel = new Model();

        Form form = new Form("form") {
            @Override
            protected void onSubmit() {
                Person p = new Person();
                p.setFirstName((String) firstNameModel.getObject());
                p.setLastName((String) lastNameModel.getObject());
                PersonDao dao = new PersonDao();
                dao.insert(p);
            }
        };

        form.add(new TextField("firstName", firstNameModel));
        form.add(new TextField("lastName", lastNameModel));
        add(form);
    }
}
    
```

Dans le constructeur, je crée deux modèles, un pour le champ *firstName* et l'autre pour *lastName*. J'ai utilisé le modèle **Model**, qui est un simple *POJO* avec un champ *object* ainsi que son *getter* et *setter*.

Je crée ensuite un composant de type *Form* qui sera associé à l'élément `<form>` dans la page *HTML* associée. Je passe donc l'identifiant de ce dernier comme paramètre au constructeur.

Je redéfins aussi la méthode *onSubmit* de *Form*, car c'est là l'occasion d'effectuer un traitement suite à la soumission d'un formulaire.

Dans ce cas-ci, il s'agit d'instancier une personne, de renseigner ses champs depuis les modèles qu'on a défini et de le passer à la méthode *insert* du *DAO*.

J'ajoute ensuite les autres composants de la page, c'est à dire deux champs de saisie (composant *TextField*). Je passe au constructeur du composant *TextField* l'identifiant de l'élément *html* correspondant mais surtout le modèle (*firstNameModel* et *lastNameModel*) correspondant.

Cette méthode, bien qu'elle fonctionne parfaitement, est trop lourde à mettre en oeuvre, et devient vite impraticable si le nombre de champs augmente.

## IV - Approche via PropertyModel

Passons maintenant à la seconde méthode qui utilise le modèle *PropertyModel*:

```
public class NewPage2 extends WebPage {
    private String firstName;
    private String lastName;

    public NewPage2() {
        Form form = new Form("form") {
            @Override
            protected void onSubmit() {
                Person p = new Person();
                p.setFirstName(firstName);
                p.setLastName(lastName);
                PersonDao dao = new PersonDao();
                dao.insert(p);
            }
        };

        form.add(new TextField("firstName", new PropertyModel(this, "firstName")));
        form.add(new TextField("lastName", new PropertyModel(this, "lastName")));
        add(form);
    }
}
```

J'ai donc ajouté deux champs *firstName* et *lastName* à la page (de même type que leurs correspondants dans la classe *Person*) et modifié la méthode *onSubmit* pour qu'elle utilise ces champs pour initialiser la personne à insérer dans la base de données.

J'ai aussi modifié le modèle passé aux champs texte, en utilisant le type *PropertyModel* au lieu de *Model*.

*PropertyModel* est une autre implémentation de l'interface *IModel* qui encapsule l'accès à une propriété d'un objet donné.

Son fonctionnement se présente comme suit :

- A sa création, un *PropertyModel* prend deux paramètres: une instance d'un objet donné et le nom d'une propriété dans cette classe.
- Lire la valeur d'un *PropertyModel* retourne la valeur du champ de l'objet associé (via reflection),
- Ecrire une valeur dans un *PropertyModel* revient à l'écrire dans le champ de l'objet associé (via reflection).

Pour revenir à la page Wicket, j'ai passé au premier *PropertyModel* l'instance de la page en cours et le nom du champ *firstName* pour référencer le champ *firstName*.

Idem pour le second.

Ainsi, en interrogeant ce modèle particulier sur sa valeur, il accède au champ associé et retourne sa valeur, et en essayant de lui affecter une valeur, il va l'affecter au champ associé.

C'est déjà mieux que la première approche, mais c'est toujours long à écrire (l'instantiation des *PropertyModel*) et est relatif au nombre de champs, dans la mesure où l'on doit créer autant de *PropertyModel* que de champs dans l'objet utilisé.

Notez que l'on peut fixer le second point (relatif au nombre de champs) en passant une instance de *Person* comme objet, au lieu de la page courante, ce qui donne:

```
public class NewPage21 extends WebPage {
    private Person person = new Person();

    public NewPage21() {
        Form form = new Form("form") {
            @Override
```

```
protected void onSubmit() {
    PersonDao dao = new PersonDao();
    dao.insert(person);
};

form.add(new TextField("firstName", new PropertyModel(person, "firstName")));
form.add(new TextField("lastName", new PropertyModel(person, "lastName")));
add(form);
}
```

Beaucoup mieux, non ?

Seulement, l'instanciation des *PropertyModel* est toujours bavarde et délicate à coder.



*En fait, le second paramètre du constructeur d'un *PropertyModel* est très flexible et ne se limite pas aux champs d'une classe donnée. Il est par exemple possible d'accéder à des listes où à des tableaux ("liste[5]") où encore parcourir un graphe d'objets ("parent.fils.petitFils").*

## V - Approche via CompoundPropertyModel

La version précédente peut encore être (grandement) améliorée en utilisant le modèle **CompoundPropertyModel**.

```
public class NewPage3 extends WebPage {
    private Person person = new Person();

    public NewPage3() {
        Form form = new Form("form", new CompoundPropertyModel(person)) {
            @Override
            protected void onSubmit() {
                PersonDao dao = new PersonDao();
                dao.insert(person);
            }
        };

        form.add(new TextField("firstName"));
        form.add(new TextField("lastName"));
        add(form);
    }
}
```

Comme vous le remarquez, ce qui a changé depuis la version précédente (utilisant les PropertyModel) est ceci:

- Je passe maintenant un modèle comme second paramètre au constructeur du formulaire (Form), qui est justement de type *CompoundPropertyModel*. Le constructeur de ce modèle prend un objet à encapsuler (ici, une instance de Person).
- Je ne passe plus de modèles aux *TextField*.

Tout d'abord, je tiens à vous rassurer que ce bout de code fonctionne parfaitement, seulement il use de trop de magie dans les coulisses, et il faut faire très attention avec ce genre de modèle.

Place maintenant aux explications: le *CompoundPropertyModel* est similaire à *PropertyModel* dans la mesure où il encapsule un *POJO Java* et peut accéder à ses champs par réflexion. Seulement, il n'est pas limité à un seul champ de l'objet associé.

Maintenant, lorsqu'un composant dans le formulaire désire agir sur son modèle (pour lire ou écrire dessus), et s'il n'a aucun modèle associé (ce qui est le cas ici), il essaie de remonter l'hierarchie de ses parents pour retrouver un composant ayant un modèle de type composé (comme *CompoundPropertyModel*). S'il en trouve un, il lui passe son identifiant pour récupérer un nouveau modèle qu'il pourra utiliser.

Ici, le parent est un composant de type *Form*, et il a justement un modèle de type composé (*CompoundPropertyModel*). Lorsque l'un de ses fils (l'un des *TextField*) demande un modèle avec son identifiant ("firstName" ou "lastName"), *CompoundPropertyModel* retourne un modèle de type *PropertyModel* initialisé avec le même objet qu'il encapsule et avec l'identifiant du composant comme nom de la propriété. Ainsi, le premier *Textfield* récupère un *PropertyModel*(person, "firstName"), et le second récupère *PropertyModel*(person, "lastName"), ce qui correspond exactement à notre besoin.

Vous l'aurez remarqué, il faut bien s'assurer que l'identifiant du champ en question est **égal** au nom du champ auquel on veut l'associer, comme ici j'ai utilisé *firstName* comme identifiant du composant *TextField* pour m'assurer qu'il soit lié au champ *firstName* de l'objet *person*.

## VI - Approche via chaînage de CompoundPropertyModel et LoadableDetachableView

La dernière optimisation qu'on puisse apporter au modèle précédent et de recourir au chaînage des modèles, qui est une facilité offerte par *Wicket*.

Le problème de l'approche précédente est qu'elle est persistante, c'est à dire que d'abord le fait d'avoir un champ dans la page (*person*) augmente sa taille dans la session, et avec des objets suffisamment complexes on peut vite saturer cet espace.

De plus, en retournant à cette même page, les champs seront renseignés avec les valeurs précédentes, car *person* n'est instancié qu'une seule fois à la création de la page.

La solution à ce problème est de fournir un moyen pour que l'objet encapsulé par le *CompoundPropertyModel* soit réinitialisé à chaque appel.

C'est justement une caractéristique du modèle ***LoadableDetachableView***, et les concepteurs de *Wicket* ont eu suffisamment de sagesse pour prévoir des cas pareils en mettant en place le chaînage de modèles:

```
public class NewPage4 extends WebPage {
    public NewPage4() {
        Form form = new Form("form",
            new CompoundPropertyModel(new LoadableDetachableView() {
                @Override
                protected Object load() {
                    return new Person();
                }
            }) {

                @Override
                protected void onSubmit() {
                    Person p = (Person) getModelObject();
                    PersonDao dao = new PersonDao();
                    dao.insert(p);
                }
            });

        form.add(new TextField("firstName"));
        form.add(new TextField("lastName"));
        add(form);
    }
}
```

Vous remarquerez qu'on a supprimé le champ *person* de la page, et qu'on passe une instance de *LoadableDetachableView* comme valeur au *CompoundPropertyModel*.

*LoadableDetachableView* est un modèle particulier qui lorsqu'il est interrogé sur sa valeur pour la première fois, invoque une méthode *load* qu'on doit définir pour la récupérer, et passe à l'état *attaché*.

Les appels suivants vont agir sur cette même valeur, et lorsque le rendu de la page est terminé, ce modèle perd sa valeur et revient vers l'état *non attaché*.

Dans ce cas-ci, dans la méthode *load*, je ne fais que retourner une nouvelle instance de *person*.

Pour ce qui est du chaînage des modèles dans *Wicket*, il s'agit simplement du fait que si un modèle A est chaîné avec un autre modèle B (en passant B comme paramètre au constructeur de A), A délègue les traitements sur la valeur au modèle qu'il encapsule.

Ainsi, comme j'ai chaîné un *CompoundPropertyModel* (A) avec un *LoadableDetachableView* (B), lorsque A a besoin d'accéder à sa valeur, il va la chercher depuis B, ce qui invoquera la méthode *load*.

On évite ainsi de stocker inutilement des objets dans la session et on s'assure qu'on disposera d'une nouvelle instance à chaque affichage de la page.

## VII - Conclusion

Cet article a servi, je l'espère, à présenter les divers modèles de *Wicket* via une approche incrémentale, en améliorant et en optimisant le résultat d'une passe à l'autre.

## VIII - Remerciements

Je remercie **lunatix** pour ses retours techniques ainsi que **fabszn** pour sa relcture orthographique.

## IX - ANNEXE: Page HTML.

Voici ci-dessous le code *HTML* correspondant aux pages *Wicket* présentées dans cet article:

```
<!--
  Document   : NewPage
  Created on : Mar 3, 2008, 5:23:07 AM
  Author    : djo
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Create New Person</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <form wicket:id="form">
      <table>
        <thead>
          <tr>
            <td>First Name</td>
            <td><input wicket:id="firstName" type="text" /></td>
          </tr>
        </thead>
        <tbody>
          <tr>
            <td>Last Name</td>
            <td><input wicket:id="lastName" type="text" /></td>
          </tr>
          <tr>
            <td>&nbsp;</td>
            <td><input type="submit" /></td>
          </tr>
        </tbody>
      </table>
    </form>
  </body>
</html>
```