More on models

More on models

Let us take a closer look at models.

The HelloWorld example program demonstrates the simplest model type in Wicket:

```
public class HelloWorld extends WicketExamplePage
{
  public HelloWorld()
  {
    add(new Label("message", "Hello World!"));
  }
}
```

The first parameter to the Label component added in the constructor is the Wicket id, which associates the `Label` with a tag in the `HelloWorld.html` markup file:

```
<html>
    <head>
        <title>Wicket Examples - helloworld</title>
        <link rel="stylesheet" type="text/css" href="style.css"/>
    </head>
    <body>
        <span wicket:id="mainNavigation">main nav will be here</span>
        <span wicket:id="message">Message goes here</span>
    </body>
</html>
```

The second parameter to Label is the model for the Label, which provides content which replaces any text inside the `<span>` tag that the Label is attached to. The model being passed to the Label constructor above is apparently a String. But if we look inside the Label constructor, we see that this constructor is merely there for convenience - it wraps its String argument in a Model object like this:

```
public Label(final String id, String label)
{
  this(id, new Model(label));
}
```

If we look up the inheritance hierarchy for Label, we see that `Label` extends `WebComponent`, which in turn extends `Component`. Each of these superclasses has only two constructors. One `Component` constructor takes a String id:

```
public Component(final String id)
{
  setId(id);
}
```

And the other takes a `String` id and an `IModel`:

```
public Component(final String id, final IModel model)
{
  setId(id);
  setModel(model);
}
```

IModel is the interface that must be implemented by all Wicket model objects. It looks like this:

```
public interface IModel extends IDetachable
{
  public Object getObject();
  public void setObject(Object object);
}
```

And the base interface IDetachable looks like this:

```
public interface IDetachable extends Serializable
{
  public void detach();
}
```

These details are not especially interesting to a beginning model user, so we will come back to this later. What we need to know right now is that Model is an IModel and therefore it can be passed to the Component(String, IModel) constructor we saw a moment ago.

If we take a peek inside Model, we'll see that the Model object is simply a wrapper that holds a Serializable object:

```
private Serializable object;
```

Model's object value is Serializable in order to support deployment of web applications in clustered environments. Note also that since IModel extends IDetachable, which extends Serializable, all IModel wrapper classes are themselves Serializable. So all Wicket models are therefore serializable, which means that they can be replicated between servers in a cluster. Don't worry if you don't fully understand this. We will talk more about clustering later.

Working with POJOs

While using a String like we did in the above example (`'Hello World'`) is convenient, you will usually want to work with domain objects directly. To illustrate how Wicket supports working with domain object, we use the following *Plain Old Java Object* (POJO) for our examples:

```
import java.util.Date;

public class Person
{
  private String name;
  private Date birthday;

  public Person()
  {
  }
  public String getName()
  {
    return name;
  }
  public void setName(String name)
  {
    this.name = name;
  }
  public Date getBirthday()
  {
    return birthday;
  }
  public void setBirthday(Date birthday)
  {
    this.birthday = birthday;
  }
}
```

Static models

Like we have seen above, the simplest way of providing e.g. a `Label` a model, is just to provide a string, and let the `Label` implicitly construct one for you.

```
add(new Label("name", person.getName()));
```

We can create the model explicitly like:

```
Model model = new Model(person.getName());
add(new Label("name", model));
```

This has the same effect as the first example, but as we now have the reference to the model, we can use it e.g. to share it with other components like we did in our little interactive example earlier.

Using models like this works fine as long as there are no changes in the underlying data that should be displayed. You could call these models *static* or *pusl* models, as you have to synchronize any data changes to the model itself.

Dynamic models

While static models are the easiest to use, it is often a good idea to let your models get their current objects dynamically from some source so that you can be certain that they display the most recent value. You could call these models *dynamic* or *pull* models.

Property models

The `PropertyModel` class allows you to create a model that accesses a particular property of its nested model object at runtime. This property is accessed using an expression language which has a syntax similar to OGNL (see http://www.ognl.org) (until version 1.1, Wicket actually did use the OGNL library, however it has been replaced by custom implementation). The simplest `PropertyModel` constructor is:

```
public PropertyModel(final Object modelObject, final String expression)
```

which takes a model object and a property expression. When the property model is asked for its value by the framework, it will use this property expression to access the model object's property.

Looking at our example POJO, the property expression `"name"` can be used to access the `"name"` property of any Person object via the `getName()` getter method. Also, when updating a model, using property expression `"name"` has the effect of calling `setName(String)`. Hence construct a property model to use our POJO, give it a property expression `"name"` and set it on a `TextField`, not only will that `TextField` display the current value of the `"name"` property, it will also update the `"name"` property with any user input provided.

More complex property expressions are possible as well. For example, you can access sub-properties via reflection using a dotted path notation, which means the property expression `"person.name"` is equivalent to calling `getPerson().getName()` on the given model object. Arrays can be accessed as in "`persons[4].name`". As you can see, PropertyModels are quite a powerful way to give Wicket components access to model objects.

More information on the syntax

More information on the allowed property expressions can be found either in the `wicket.util.lang.PropertyResolver` JavaDoc or on the Property Expression Language wiki page

Compound property models

If you have looked at some examples of Wicket already, you may have wondered why components, which generally must have a model, can be constructed with only an id, omitting any model. There are three reasons for this.

The first, most obvious reason is that not all components actually need a model. Panels and pages, for example, do not use models, though you might find it handy to store a model in these components.

The second reason to construct a `Component` with no model is that it is sometimes impossible to have a model ready for the call to `super(String, IModel)` in the `Component`'s constructor. In this case, the `Component` can be constructed without a model, and the model can be later set with a call to `setModel()`.

The third reason to construct a component with no model is that some components are actually bound to their models at runtime. When an attempt is made to retrieve the model for a Component which does not yet have one, instead of failing and returning null, the `getModel()` method in `Component` calls the method `initModel()`, which gives the `Component` an opportunity to lazy-initialize, returning a model on-the-fly.

This mechanism opens up all kinds of possibilities for model initialization. One particularly convenient and efficient lazy model initialization strategy is employed by the `CompoundPropertyModel` and `BoundCompoundPropertyModel` classes to allow containers to share their models with children. By sharing model objects like this, fewer objects are created, which saves memory, but more importantly, it makes replication of models much cheaper in a clustered environment.

To use a `CompoundPropertyModel`, you simply set one as the model for any container. However, a very typical place to install a `CompoundPropertyModel` is on a `Form` or `Page`. This is natural, because the compound model itself is usually the model that is being displayed by the `Page` or edited by the `For`.

To see a CompoundPropertyModel in action, take a look at the FormInput example . In the InputForm nested class in FormInput.java, the Form is constructed like this:

```
public InputForm(String name)
{
  super(name, new CompoundPropertyModel(new FormInputModel()));
  ...
}
```

If we look at the next few lines of the Form's constructor, we see statements like:

```
add(new RequiredTextField("stringProperty"));
add(new RequiredTextField("integerProperty", Integer.class));
add(new RequiredTextField("doubleProperty", Double.class));
```

these statements are adding FormComponents to the Form. But they are not explicitly attached to any model. Instead, they will inherit the CompoundPropertyModel for the form at runtime. When the framework needs the model for a RequiredTextField, it will call getModelObjectAsString().
This method retrieves the text field's model by calling getModel(), which is implemented like this:

```
public IModel getModel()
{
  if (model == null)
  {
    this.model = initModel();
  }
  return model;
}
```

Since the RequiredTextField has no model, the initModel() method will be called, resulting in a search up the containment hierarchy for a CompoundPropertyModel that can be inherited by the model-less child:

```
protected IModel initModel()
{
  for (Component current = getParent(); current != null; current = current.getParent())
  {
    final IModel model = current.getModel();
    if (model instanceof CompoundPropertyModel)
    {
      return model;
    }
  }
  return null;
}
```

The text field will discover the CompoundPropertyModel for the Form and set that as its own model. So getModel() on the text field ultimately returns the model of the Form.

Next, the nested model object is retrieved from the model by passing in the Component (in this case the RequiredTextField) as a parameter to IModel.getObject(Component). This is the magic that allows the CompoundPropertyModel to retrieve the right value from the Form's model for the Component that is asking for it:

```
final Object modelObject = model.getObject(this);
```

The retrieval is done by using the name of the Component as the property expression to access the model. In the case of this component

```
add(new RequiredTextField("stringProperty"));
```

the `getStringProperty()` method would be called on the `Form`'s model.

The model's object value is available in this way to the component for processing. In the case of the `stringProperty` text field component, it will convert the value to a String. In the case of other `RequiredTextField`'s in the FormInput form, a conversion may occur based on the third parameter to the Component. For example, this component:

```
add(new RequiredTextField("integerProperty", Integer.class));
```

will convert the contents of the text field to and from an Integer value.

BoundCompoundPropertyModel has been removed in Wicket 6!

In the event that you need more flexibility or property expressions power in your compound models, you can use the `BoundCompoundPropertyModel`. This class provides three methods you can call on the container's model object to bind a given child Component to a specific property expression and/or type conversion:

```
public Component bind(final Component component, final String propertyExpression)
public Component bind(final Component component, final Class type)
public Component bind(final Component component, final String propertyExpression, final Class type)
```

To see how this might be used, suppose that the `stringProperty` value needed to be bound to the property expression "`person[0].stringProperty`". In the `FormInput` constructor, we'd say something like this:

```
super(name);
final BoundCompoundPropertyModel formModel = new BoundCompoundPropertyModel(new FormInputModel());
setModel(formModel);
add(formModel.bind(new RequiredTextField("stringProperty"), "person[0].stringProperty"));
```

The `Form` is constructed without a model. We then create the Form's model as a `BoundCompoundPropertyModel`, set it as the Form's model and finally use it to bind the `RequiredTextField` to the desired property expression.

String Resource Models

Localization of resources in Wicket is supported by the `Localizer` class. This provides a very convenient way to retrieve and format application and component specific resources according to the `Locale` that the user's Session is running under. In many cases, it will be sufficient to simply retrieve a localized String in this way, but if a formatted, localized string is desired as a model, a `StringResourceModel` may be used.

`StringResourceModel`-s have a resource key (for looking up the string resource in a property file), a `Component`, an optional model and an optional array of parameters that can be passed to the Java `MessageFormat` facility. The constructors look like this:

```
public StringResourceModel(final String resourceKey, final Component component,
                           final IModel model)

public StringResourceModel(final String resourceKey, final Component component,
                           final IModel model, final Object[] parameters)
```

A very simple example of a `StringResourceModel` might be a `Label` constructed like this:

```
add(new Label("greetings", new StringResourceModel("label.greetings", this, null)));
```

where the resource bundle for the page contains an entry like this:

```
label.greetings=Welcome!
```

By adding a model, we can use the property expressions to interpolate properties from the model into the localized string in the resource bundle. For example, suppose we wanted to add the name from a `User` object to the greeting above. We'd say:

```
User user;

...

add(new Label("greetings", new StringResourceModel("label.greetings", this, user)));
```

and have a resource like:

```
label.greetings=Welcome, ${user.name}!
```

where `User` has a `getName()` method exposing its `"name"` property.

The `Component` class has convenience method which you may find useful:

```
add(new Label("greetings", getString("label.greetings", new Model(user))));
```

Custom models

If we do not want to use the property expressions/introspection, but instead want to to pull a fresh value from the model on each request in a more strongly typed manner, we could provide our own implementation of `IModel`:

```
final Person person = new Person();
person.setName("Fritzl");
Model model = new Model()
{
  public Object getObject()
  {
    return person.getName();
  }
};
add(new Label("name", model));
```

Note that in the above example, we extended from `Model` instead of implementing `IModel` directly, and we did not bother writing `setObject()` (as for the label it is not needed).

By implementing your own models, you can support any kind of model behaviour you can think of, though it is usually a more verbose way of doing things.