# Data Binding - how does it work?

## Soft binding versus hard binding

DomUI data binding uses a form of data binding we call "soft binding". When you bind some model property to a component it is the *component* which maintains (remembers) the binding. The actual binding of the data is triggered by two global events:

- When a request *enters* the servers we move data from *control* to *model*
- When all user logic has finished, and the response is to be rendered back we move data from *model* to control. This means the control will show the value from the model when rendered back in the browser.

Because the binding itself is maintained by the control bindings are only active while the control itself is attached to a page. If a control gets removed from a page it's bindings will not update. But as soon as a control is added to a page (again) binding will be updated.

The alternative to "soft binding" is "hard binding". In hard binding all property setters of classes are "Observable": they are coded in such a way that they call listeners as soon as a setXxx() call changes the value of the property. Instrumenting all setters to do this properly adds yet more boilerplate to Java's already very verbose "properties" which is a disadvantage. In addition having setters propagate changes means that it is very hard to control *when* bindings execute. And finally, since objects might live quite a while it becomes important to be able to clean up listeners on properties, or the system leaks memory or CPU cycles.

Using soft binding prevents all this: the update mechanism is bound to the request/response cycle, and bindings are automatically removed as soon as components go out of (visible) scope.

## Binding to the "value" of a component: bidirectional bindings

We usually bind a value in the model to the value of a Control. This is what we do when we specify something like:

```
control.bind().to(editModel, "firstName");
```

This binding *implicitly* binds to the value of the control, and it is *bidirectional*: if the control changes the data moves from control to model, but if the model changes it moves to the control. This bidirectionality is important for the value because that is what is used in the model and presented on the screen.

### Binding errors: the bindValue property

There is one small lie in the above. The code above usually binds to a control property called *bindValue* instead of the property *value*. The reason for this is a kind of design issue in DomUI: the getValue() call of a component triggers conversion and validation of the control's contents, and if any of these fail the getValue() call will set the control in error state immediately, and send an error to the form so that it is displayed on-screen. This behavior is reasonable if you do not use binding but with binding it has the effect that every roundtrip to the server will immediately validate all controls. For instance when a form has mandatory fields the first field filled in will cause all other fields to become red, and for all of them the screen shows the "xxx is required" message.

As we cannot change the above behavior for getValue() (as existing code depends on it) we need an alternative, and that is the bindValue() property. The getBindValue() method does exactly the same as getValue() *except* propagating an error to the error UI. Calling getBindValue() will do conversion and validation as usual, but any error is just thrown as a ValidationException without it propagating to the display.

The binding code will catch the validation exception and keep it inside the binding so that the fact that the control contains invalid data is not lost. The value inside the model is not changed.

To actually show binding errors (and to know they are there) you, as a developer, should execute the following code *before* using model data (for instance inside your save() method):

```
if(bindErrors())
  return;
```

This call will walk all DomUI nodes starting at the "this" node (so the entire subtree) and check for binding errors. All errors will then be propagated to the control UI (using setMessage()) on the control) which in turn will cause the messages to be shown. If any binding contains errors the bindErrors()) call returns true, and in that case we should cancel the save. We do not need to show a message as the errors are already shown.

Binding to bindValue is automatic as long as you use the xxx.bind() syntax of binding. This call will check that the control actually has a bindValue property and use that, if not it will use the value property instead.

## Binding to other Control properties: unidirectional bindings

Binding to the bindValue/value property is bidirectional. But you can bind datamodel values to other properties of a control. This allows the model to influence the behavior or display of a control. To bind to some other property than the value of a control you must specify the control property to bind to in the bind() call. Example: to bind to the "disabled" property we code:

```
bind("disabled").to(model, "readOnly");
```

As soon as the model's readOnly property changes this change is propagated to the control. This makes it interesting and simple to move business logic to separate classes: just expose properties that define the wishes for the UI and bind those to the appropriate component properties.

There is one thing special about these bindings: they are *unidirectional*. This means that the binding only moves data in one direction: always from model to control, never the inverse.

There was a definite reason why I needed these to be unidirectional but I do not remember at this time.

In fact, all bindings to control properties except value are unidirectional. This also means you cannot make a control that returns multiple values on multiple properties as you cannot bind to the others.

## How components register bindings

Bindings are made with the bind() call on the component that you want to bind to the model. The bind() call creates a builder and this builder requires you to call one of the "to" methods on it to complete the builder. As soon as the builder gets finished the builder creates a ComponentPropertyBindingBidi or ComponentPropertyBindingUni instance which represents the binding and stores it *inside* the DomUI node for the control.

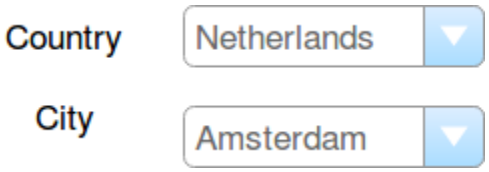
Because the actual bindings are stored inside the DomUI tree we have automatic lifecycle management: as soon as a DomUI node is removed from the display tree the bindings become inactive.

## Binding order

### Why is binding order important?

Components on the screen are ordered naturally, by their DOM location as created by the developer. But to handle binding properly it is important that binding is done in proper order.

Let's give an example. We have a screen with two combo boxes:




The "Country" combo contains a few countries, the "City" combo contains cities in all of these countries - not just of the selected country.

Now let's see what happens if we order binding in "dom order"... Say the user changes "Netherlands" to "Great Britain":

- The request enters the server, and binding executes *moveControlToModel*.
- "Country" changed to "Great Britain":
    - The model discovers that city "Amsterdam" is not in Great Britain, so it changes the "city" property to "London".
- But now we bind "city" which is "Amsterdam" from the UI code.
    - The binder overwrites "London" with "Amsterdam", and updates "Country" back to Netherlands.

Clearly this is not really what we want. We need to have some "order" imposed on the bindings, so that it behaves as expected.

### How DomUI orders binding

When a request comes in the binder (SimpleBinder) will run moveControlToModel, after all components have obtained their value(s) from the request. It does that as follows:

- Walk the tree, and find all bindings.
- Check each binding for a changed value, i.e. where the model value differs from the control value.
    - For items that have the same value: ignore
- We now have a list of bindings whose value changed. Order these bindings as follows:
    - A "deeper" binding comes before a "higher" binding
    - Bindings at the same "level" execute in order of dom traversal.
- Now bind all values as per the above ordering.

By ordering like this most binding issues should resolve themselves automatically.

# Issues / pitfalls when using binding

## Binding performance

Bindings are visited twice every roundtrip: once to move controls to the model (at request entry) and once to move the model back to the controls (when the response leaves the server). Since bindings are handled so often we need special measures to ensure that this does not get slow.

The core of binding handling is walking the node tree which is quite fast, so currently nothing is done to change that.

But moving data is optimized, and most of the optimizations are actually done by the components themselves.

All DomUI components check, whenever some setter is called, whether the value that is set actually changed. If it did not then the setter exits immediately. This ensures that controls are only really updated when some value actually changes on them.

Once data actually changes most DomUI components simply decide to call forceRebuild(). This call destroys their current presentation and causes them to be rebuild at render time. If multiple values change on a component forceRebuild() gets called every time- but this call is very cheap. The actual (expensive) rendering only happens when it is actually time to present the thing, and at that time no changes will be made anymore.

## When has a value changed?

Most properties on a control are primitives or simple values (like dates and strings), so checking whether they change in the setter is quite cheap and well-defined.

But the value property can also be some complex data structure like a class instance with multiple properties, a list or a map. For this there is an issue with "what is equal and what is not".

All controls handling simple values use equals() to detect whether a new value has been set or not. But calling equals on things like collections and maps can be very expensive. What is worse is that the most common case (nothing changed) is the most expensive: every object in the list/map needs to be checked. Most DomUI components that have these kinds of properties will use == (reference equality) instead of equals (structural equality) for these properties (including the value property) to prevent binding to become glacially slow.

This means that to make DomUI aware of a change in these properties you need to create a new instance of the list, map or whatnot.

## Complex values (values with thingies in them)

In fact, all values that are compound (consist of multiple thingies) are something to be aware of. Collections and Maps are obvious examples but it also goes for values that are classes with multiple values in them. Take for instance a class like this:

```
public class Person {
  public property String firstName;
  public property String lastName;
}
```

We can make some IControl<Person> which will edit/display a Person by binding it to some datamodel property of the same type. This works fine if Person is immutable: in this case the only way to change the data of a person is to actually create a new instance, and doing a setValue() of that instance. This will be seen as a change by the control (as the instance changes) and consequently the control's presentation is redone, showing the new person data.

But if Person is mutable another way to change it is to do something like p.setFirstName("Frots"). This will change the firstName property inside the instance, but the control cannot notice that change because the instance it holds is actually changed, so comparing its instance (either by reference or even by equals()) never shows the chanage. The net effect is that data has changed- but the control shows an old value.