The D2W Rule System

Related articles

What is Direct to Web?
Changing the Rules with Direct to Web

The D2W Rule System

by Ramsey Gurley

As you probably know, WebObjects is built using the Model-View-Controller (MVC) design pattern. The rule system is in the control layer of a WebObjects D2W app. With it, you can define everything from component bindings to application behavior. To a D2W novice, the rule system can seem really intimidating. It isn't well documented and the location of rule system keys that control application behavior aren't immediately obvious either. Hopefully, by the end of this document, you'll have a better idea of where to look and what to do with the WebObjects rule system.

Rule system techniques

Rule system basics

To start, let's look at the structure of a rule. At the most basic level, a rule has three fundamental parts: A priority, a qualifier(also known as the 'left hand side' or LHS), and an assignment(also known as the 'right hand side' or RHS). The idea is that rules are 'fired' when the rule's qualifier matches the current state of the D2WContext. Once a rule is fired, evaluation for that particular key path stops. The priority decides the order in which rule are evaluated, and the Assignment defines the logic used to assign a value to the key path the D2WContext is trying to infer.

That's quite a mouthful, so let's take a look at a very basic example:

```
100: *true* => pageWrapperName = "PageWrapper" [WO:com.webobjects.directtoweb.Assignment]
```

The parts of this rule are:

```
Priority: qualifier => keyPath = value [WO:Assignment class]
```

In this case, the qualifier is always true, so long as no other rule priority is greater than 100, the pageWrapperName key in the D2WContext will always equal "PageWrapper". The assignment being used is the simple Assignment class, so the string in 'value' is assigned directly to 'keypath.' And since the D2WContext implements NSKeyValueCodingAdditions, you can reference this keyPath in a d2w component using a wod binding like:

```
PageWrapper : WOSwitchComponent {
        componentName = d2wContext.pageWrapperName;
}
```

By using a switch component, the rule system decides what component is used to generate the page wrapper. If you do this in all your d2w page components, you can easily switch out a page wrapper using the rule system. Indeed, this is what IS done in every D2W page component I've seen. By defining component bindings like this, you create a clean separation of application controller logic from the application's view layer. Without the rule system, your logic would end up scattered across the view layer and it would be considerably more difficult to maintain.

And that is really the basic concept behind of the whole rule system. Simple, right? 🙂

It's worth noting, this answers a fundamental question about the rule system: Where are the keys that I can use in the rule system located? The keys are located in the D2W components. If you are trying to change the behavior of a D2W page or component via the rule system, there's currently no better way than to simply look at the component source and bindings for clues as to what those keys happen to be. As such, when you start writing your own custom D2W components, you should use the same keys consistently or your rules file will become an unmanageable mess. You can create new keys whenever they are necessary, but it is best to reuse the existing ones when possible.

Overriding rules

Overriding a rule is a simple task. Continuing with the example above, suppose you want to use a different page wrapper for editing objects than you do throughout the rest of the app. In this case, overriding the rule is quite simple:

```
101: task='edit' => pageWrapperName = "EditPageWrapper" [WO:com.webobjects.directtoweb.Assignment]
```

This rule has a higher priority than the previous rule and the qualifier specifies the 'edit' task specifically. What this means is that when the D2WContext's valueForKey("task").equals("edit") == true, then valueForKey("pageWrapperName").equals("EditPageWrapper") will also be true. Instead of using "PageWrapper" defined by the previous rule, the component used when task='edit' will use the "EditPageWrapper" instead. If the task is not edit, then this rule will not fire and the rule with the next highest priority will be evaluated.

Rule chaining

Rules can also be chained. In other words, one rule may depend on the outcome of another rule. Consider the following pair of rules:

```
100: *true* => editPageTemplateName = "ProfessionalEditPage" [WO:com.webobjects.directtoweb.Assignment]
100: *true* => pageName = "editPageTemplateName" [WO:er.directtoweb.assignments.ERDKeyValueAssignment]
```

In this case, the key pageName is assigned the string value "ProfessionalEditPage". Notice the two different assignment classes. If com.webobjects.directtoweb.Assignment had been used on both rules, pageName would equal the string "editPageTemplateName" instead of "ProfessionalEditPage." The custom wonder assignment in this case tells the rule system to assign the value of the key editPageTemplateName to the key pageName. Extending this example, you might have

```
100: look = 'pro' => editPageTemplateName = "ProfessionalEditPage" [WO:com.webobjects.directtoweb.Assignment]
100: look = 'fun' => editPageTemplateName = "FunEditPage" [WO:com.webobjects.directtoweb.Assignment]
100: *true* => pageName = "editPageTemplateName" [WO:er.directtoweb.assignments.ERDKeyValueAssignment]
```

If you repeat these rules for your query, inspect, list, etc etc etc pages, you can now switch the entire layout of your app with a single key named 'look.'

Rule caching

Suppose you now want override your 'look' key.

```
100: *true* => look = "fun" [WO:com.webobjects.directtoweb.Assignment]
105: session.user.isAdmin => look = "pro" [WO:com.webobjects.directtoweb.Assignment]
```

For the sake of this discussion, let us assume our D2W app is only using Apple's D2W framework (Despite the fact that we are referencing a custom wonder assignment in our chaining). You first visit your 'edit' page for the entity 'Foo' and get the 'fun' template. Now you log in and return to the 'edit' page for the entity 'Foo' and discover the look is stuck on 'fun'. You cannot get the edit foo page to switch to your 'pro' look! This happens because the rule system caches key values for performance. Once a key is evaluated, the rule system may cache it and reuse that value. To resolve this issue, and others like it, you need to understand rule system caching.

Caching is fairly straightforward, but it differs depending on whether you are using Project Wonder's D2W framework or just Apple's. Apple's D2W framework has a set of global keys called significant keys. The default significant keys are task, entity, propertyKey, and configuration. You can add more using D2W.factory().newSignificantKey() method. In the above example, you would only need to add the 'look' and 'session.user.isAdmin' key paths to your global significant keys for everything to work properly. However, you must be careful since the keys are global. If you have too many significant keys, then you will basically have no caching and your application's performance will suffer. You must also be careful which keys you select. For example, including 'session' as a significant key would practically disable caching.

Project Wonder's rule system

Project Wonder has an overhauled caching system that works in a more granular way. For the remainder of this article I will be discussing the Wonder rule system unless otherwise noted. Wonder has two basic Assignment subclasses you will want to use when creating your own custom assignments. ERDAssignment and ERDDelayedAssignment. These classes differ in the way they handle caching. Values assigned by an ERDAssignment are cached using dependent keys. Values assigned by ERDDelayedAssignment simply are not cached.

ERDAssignment

ERDAssignment caches values using the ERDComputingAssignmentInterface. That interface defines one method: dependentKeys(). The dependentKeys method takes one string argument representing a keyPath. The return value is an array of keyPaths that will change the outcome of the Assignment value to keyPath. In this way, the Wonder rule system allows for 'significant' keys to be assigned on a per keyPath basis. So, for example, if you create a custom assignment for the 'look' key and the value needs to be re-evaluated based on the current user, then your dependentKeys() method might look something like:

```
public NSArray dependentKeys(String keyPath) {
    if("look".equals(keyPath)) {
        return new NSArray(new String[] {"session.user"});
    }
    return NSArray.emptyArray();
}
```

ERDAssignment also does a bit of magic for you in the fire() method. If you were to simply descend from Assignment, you would typically override the fire() method and execute your custom logic there. In many cases, you will want to execute different logic based on the keyPath being assigned. So when a ERDAssignment fires, it instead calls a method named by keyForMethodLookup(). This method simply returns keyPath() unless overriden. So, instead of overriding fire() in a ERDAssignment subclass, you create a method by the same name as your keypath. For example, given the following rule that uses a your custom ERDAssignment:

```
100: *true* => myKeyPath = "someValue" [WO:com.whatever.MyGreatERDAssignment]
```

Your method that handles the assignment would look like:

```
public Object myKeyPath(D2WContext d2wContext) {
    // Logic goes here to apply a value to the key named by keyPath()
}
```

In this way, you can define the logic for individual keys separately. This allows you to use a single assignment to resolve values for multiple keyPaths without resorting to a large if/else block. You can extend this behavior further by overriding the keyForMethodLookup() method. If you override this method and return value() for instance, it would then assign a value to your key with a method by the same name as your value.

```
public Object someValue(D2WContext d2wContext) {
    // Logic goes here to apply a value to the key named by keyPath()
}
```

This gives you multiple assignment methods per key if you need them without spreading the logic over different assignment classes. It is a very clever solution.

ERDDelayedAssignment

ERDDelayedAssignment differs from ERDAssignment in that delayed assignments are not cached. The rhs key is evaluated each time the D2WContext needs it. However, there is a catch. The value of a delayed assignment is NOT reevaluated if it is used on the LHS of a rule. Suppose you have the following rule:

```
100: *true* => calendarView = 'session.userPreferences.calendarView' [WO:er.directtoweb.assignments.
ERDDelayedKeyValueAssignment]
```

Now, anytime you call d2wContext.calendarView in your components, you'll get the value contained in session.userPreferences.calendarView. You then try to set a componentName using the 'calendarView' key in rules like this:

```
100: calendarView = 'DAY' => calendarComponentName = "DayView" [WO:er.directtoweb.assignments.
ERDDelayedKeyValueAssignment]
100: calendarView = 'MONTH' => calendarComponentName = "MonthView" [WO:er.directtoweb.assignments.
ERDDelayedKeyValueAssignment]
100: calendarView = 'WEEK' => calendarComponentName = "WeekView" [WO:er.directtoweb.assignments.
ERDDelayedKeyValueAssignment]
100: calendarView = 'YEAR' => calendarComponentName = "YearView" [WO:er.directtoweb.assignments.
ERDDelayedKeyValueAssignment]
```

If you have not called the key from the d2wContext before this rule fires, it won't matter if session.userPreferences.calendarView has changed, and the last value inferred by the rule system will be used. The key is effectively cached for LHS usage. Fortunately, we can simultaneously solve this problem and reduce the number of rules required using another custom Wonder assignment. ERDDelayedSwitchAssignment gives you the ability to move your qualifier to the RHS:

```
100: *true* => calendarComponentName = {"qualifierFormat" = "calendarView = '@@'"; "switch" = {"DAY" = "DayView";
"MONTH" = "MonthView"; "WEEK" = "WeekView"; "YEAR" = "YearView"; }; } [WO:ERDDelayedSwitchAssignment]
```

Tips and tricks

Additional rule model files

By default, the rule system checks for two rule files in your projects and frameworks. Those are user.d2wmodel and d2w.d2wmodel. If you are using Project Wonder, you can add more by setting the following property in your properties file:

```
er.directtoweb.ERD2WModel.additionalModelNames = ("myOther.d2wmodel","yetAnother.d2wmodel")
```

Rule tracing

You can turn on rule tracing in the console using `-D2WTraceRuleFiringEnabled YES` in your run/debug configurations. To get more detailed information, you can use Wonder's rule system logging instead.

Related Links

Apple's D2W documentation
rule modeler
Configuring Rule Modeler