Development-WO Component-Code Template and WODs

Overview

WebObjects was created using several well-understood and powerful software design patterns. The most obvious pattern used by WebObjects is the Model-View-Controller architectural pattern. This means that code and resources in a WebObjects application are separated into two fundamental roles: *Model* and *View*. The *Controller* role may be fulfilled by Model, View or in a separate set of Controller classes or, more commonly, a little of all three.
The Model portion of a WebObjects application includes:

- **EOModels** - several `.plist` files bundled together into a `.eomodeld` package that defines an Entity-Relationship Model, including Entities, Attributes and Relationships. Originally Apple provided a diagramming tool called *EOModeler* that allowed graphical design of a Model including drag-and-drop creation of relationships, but it was deprecated along with the rest of the Apple-authored development tools when WOLips and became the recommended development environment.
- **Entity Classes** - These classes provide the core getters and setters for attributes and relationships. Custom business logic is also often added to these classes.

The View portion of a WebObjects application includes:

- **Components (`.wo` packages)** - these are packages made up of several additional files
  - **`.wod` file** - this file abstracts the bindings to the `.java` classes out of the HTML/XML/etc. definition to further separate the layout portions of the View from the code portion. Modern WebObjects applications often use in-line bindings which can replace the use of `.wod` files, but at the cost of more tightly tying the code to the layout.
  - **`.html` file** - this file can be a standard `.html` simply with the addition of `<webobjects>` tags that bind the Component to the `.java` class.
  - **`.api` file** - this file can define the API of a component that is reusable by other components, defining the available and required bindings.
  - **`.woo` file**
- **Component Java Classes** - These classes provide the logic to provide values for the bindings defined in the `.wod` or `.html` files.
- **Localizable.strings files** - these files define the localized strings that are available to a WebObjects application that implements Localization to provide a different presentation to users that speak different languages.

Not Just HTML!
One of the things interesting aspects of WebObjects is that the component architecture does not presume to produce HTML. You can use the same component system to create HTML, XML, CSS, XSLT, emails, or any other type of dynamic data.

Code

Whether you extend WODynamicElement or WOComponent, you always have a Java class that implements your component's logic, and optionally stores its state. By default, the name of your Java class determines how you will refer to your Component throughout your application, which you will see several examples of later. There are a few key differences between implementing WODynamicElements and WOComponents.

WODynamicElements

If you are building a WODynamicElement, your class will not have a Template or WOD file. Any output is generated in code directly. WODynamicElements also can not assume that there will be an instance per usage, rather a single instance of your class may be concurrently serving multiple Components. As a result, your WODynamicElement must be completely thread-safe. These attributes make WODynamicElements ideal for producing relatively small output or output that is "computed." If you find yourself writing large amounts of code to handle outputting data, you may want to consider the benefits of templates that come from building WOComponents.

There are three primary methods that you may implement in your WODynamicElement depending on the type of interaction your element provides or requires:

- takeValuesFromRequest
- invokeAction
- appendToResponse

For more information on the execution order of these methods at runtime, read the Request-Response Loop section.

To receive values from the request (for instance, form data), implement the takeValuesFromRequest method. If your WODynamicElement responds to user actions, you must implement the invokeAction method. And if your WODynamicElement generates output, you must implement the appendToResponse method. At runtime, your WODynamicElement will be wired up to a WOComponent that you can access via the WOContext that is passed into each of the three methods above, which can be used to resolve bindings in a particular appearance on a page. For example, to retrieve the value of a binding, it might look like:

```
private WOAssociation myNameAssociation;
...
public void appendToResponse(WOResponse _response, WOContext _context) {
  WOComponent component = _context.component();
  String name = (String) myNameAssociation.valueInComponent(component);
  super.appendToResponse(_response, _context);
}
```

This same pattern is used for Stateless WOComponents. WODynamicElement can provide a .api file to describe its bindings. For more information on .api files, see the .API File section below.

WOComponents

WOComponents add several major capabilities above and beyond WODynamicElements. The primary additional capability is support for templating. When you create WOComponent, you can also create a ".wo" folder, which contains three files: a .html or .xml template, a .wod binding declaration, and a .woo file. Like WODynamicElements, WOComponents can also provide an optional .api file (as described below). The code for a WOComponent can be a simple as simply declaring that your class extends WOComponent along with its constructor. For example:

```
public class MyComponent extends WOComponent {
  public MyComponent(WOContext _context) {
    super(_context);
  }
}
```

The same three core methods described above for WODynamicElements also exist for a WOComponent, however because WOComponents can be stateful, you can also declare instance variables (ivars) in your component that you can bind to. Each time your WOComponent is used on a page, a new instance is created, which is another large distinction as compared to a WODynamicElement. It is not necessary for your WOComponents to be thread-safe.

The WO Folder

Templates

By default, WOComponent templates can be either .html or .xml files. WebObjects templates separate all binding declarations into a separate file called a .WOD file, which is different than many other web frameworks. As a result, the templates are generally much easier to read and can often be more easily given to a designer to work on without the risk of code-related bindings being modified by accident. WOComponents are declared in the template using a "webobject tag", which looks like an HTML tag:

```
<webobject name = "PersonName"></webobject>
```

The name declared in the WebObject tag will be used as a key to lookup the corresponding binding definition in your WOD file.

WOD Files

WOD files (Web Object Declaration?) define the bindings for each of the component references in the template. For instance, if the reference above was connected to a "public String personName()" method on your component, the WOD declaration might look like:

```
PersonName : WOString {
  value = personName;
}
```

There are several pieces to this declaration. The first token is the WebObject tag name which appears in the template. These values much match exactly for WebObjects to resolve the binding information. If you reference a name in your template and do not declare a corresponding WOD entry, WebObjects will throw an exception at runtime.

The second token, after the colon, is the name of the component or element to instantiate in your template. This name is resolved using NSBundle lookup rules, which by default will find any class that has the same name as the token, excluding package name. For instance, WOString might actually be "com.webobjects.appserver._private.WOString", but the name of the class without its package name is "WOString". This is important to note, because it means that if you are using this syntax, you must uniquely name your classes across all packages (an Objective-C heritage, where the concept of a package did not exist). You can, however, choose to fully qualify your class names, in which case the "WOString" in the example above would become "com.webobjects.appserver._private.WOString" and there would be no ambiguity in class resolution. It is possible to override the normal NSBundle lookup process by using a method on NSUtilities:

```
NSUtilities._setClassForName(com.bla.TestComponent.class, "TestComponent");
```

This call would bind the name "TestComponent" appearing in a WOD reference to the class com.bla.TestComponent, avoiding any further class "hunt". This is also useful to replace internal component implements with your own. For instance, you could replace the implementation of WOString with your own class - a capability that Project WOnder makes extensive use of to provide bug fixes and enhancements to the core components.

Moving along in the WOD file, inside of curly braces, you can provide a series of key/value pairs, each ending with a semicolon as a separator. The left hand side of the equals is the binding name. The binding name is resolved on a WOComponent by attempting to find a mutator method or field on your WOComponent class that matches one of the naming conventions: public void setValue(Xxx param), public void _setValue(Xxx param), public Xxx value, public Xxx _value (assuming the example above of a binding named "value"). So if your WOComponent had a setValue method, and automatic binding synchronization was enabled, the setValue method would called passing in the evaluated value of the right hand side of the binding. In the example above, the right hand side is "personName". If personName was actually in quotes in example above, it would be considered a string literal and would be equivalent to calling setValue("personName").

However, personName has no quotes, and is thus interpreted with Key-Value-Coding (KVC). Key-Value-Coding allows you to string together a series of accessor method calls or field references as a string that WebObjects will dynamically resolve using Java reflection. For instance, in the example above, personName would attempt to find any accessor or field named: `public String getPersonName(), public String _getPersonName(), public String personName(), public String _personName(), public String personName, public String _personName`. So if your Java class had a `public String getPersonName()` method, the return value of that method would be passed into the `setValue(..)` method of your WOComponent. Where KVC gets more interesting is that you can construct a series of method calls.

For instance, instead of a person, imagine that your WOSession had a `public Person person()` method, which had a `public Address address()` method, which had a `public String zipCode()` method. Your WOD file binding could look like:

```
value = session.person.address.zipCode;
```

which would bind the zipcode of the address of the person in your sesson to value. Even more interesting are the NSArray operations that Foundation provides. For instance, if your session had a `public NSArray purchaseAmounts()` that returned an array of `BigDecimal`-s, you could refer to the binding `value = session.purchaseAmounts.@sum`, which would return the sum of the values of the array. There are several other Array operations available in the Foundation classes, and Project Wonder provides even more in its ERXArrayUtilities class. For even more advanced KVC capabilities, read the WOOgnl section of Project Wonder.

A single WOD entry can contain several binding declarations, and a WOD file can contain many entries. For example, here is an excerpt from a real WOD file:

```
FilterAction : WOSubmitButton {
  action = filter;
  value = "filter";
}

EditAction : WOHyperlink {
  action = editRequest;
}

NoRequestsConditional : WOConditional {
  condition = requestsDisplayGroup.allObjects.count;
  negate = true;
}
```

WOO Files

At a minimum, a WOO file declares the WO version and the Template's character encoding. As an example, a bare bones WOO file might like look:

```
{
  "WebObjects Release" = "WebObjects 5.0";
  encoding = NSMacOSRomanStringEncoding;
}
```

Additionally, a WOO file can contain definitions of instantiated objects that your WO component can refer to. If you build your components with WOBuilder, you will find the definition of the attributes of your WODisplayGroups in your component's WOO file.

API Files

API files are optional files that appear in the same folder as your .WO, and provides metadata about your component that can be used by an IDE to provide a better experience for your components' users. For instance, API files declare all of the bindings for your component, the binding types (boolean, date, etc), along with a fairly extensive XML language that is used to define binding validation.

As an example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<wodefinitions>
  <wo wocomponentcontent="true" class="AjaxSortableList.java">
    <binding name = "id"/>
    <binding name = "list"/>
    <binding name = "listItemIDKeyPath"/>
    <binding name = "startIndex"/>
    <binding name = "action"/>

    <validation message = "&apos;id&apos; is a required binding">
      <unbound name = "id"/>
    </validation>

    <validation message="&apos;listItemIDKeyPath&apos; must be bound when &apos;list&apos; is bound">
      <and>
        <bound name = "list"/>
        <unbound name = "listItemIDKeyPath"/>
      </and>
    </validation>
  </wo>
</wodefinitions>
```

In this example, the API defines the bindings at the top, along with a series of validations. The validation message is displayed in the IDE when the declarations inside the validation evaluate to "true". For instance, for the first validation, the validation message appears if the "id" value is not bound. In the second case, the message appears if the "list" value is bound but "listItemIDKeyPath" is NOT bound.