Client-Side Components: Support Classes

Java Client-Side Components

Introduction

Programs written for the World Wide Web implicitly acknowledge a well-defined and somewhat limiting separation between the client (browser) and the server. Data can pass from server to the browser and back again, but only under rigidly controlled conditions. This is considered true of object-oriented programs as well as procedural programs. For example, you can have objects on the server side of a connection and objects on the browser side that have been downloaded from the server. The server can pass initial values to the downloaded objects, but once they're on the page, these objects tend to be pretty much on their own.

A feature named *Java client-side components* bridges this "gulf." It enables Java applets on the client side to synchronize their states continuously with objects on the server side. Java client-side components also allow applets to trigger action methods on the server.

There are several advantages to using client-side components in a WebObjects application. The most obvious benefit is that Java applets have almost none of the limitations of forms. HTML forms are fixed and limited as to what they can do or how they look. You have fields and buttons, and not much else. Applets, on the other hand, are true programs with a graphical user interface, set in the context of an HTML browser. Applets can be just about any imaginable control on a page: a dynamic calendar, a spreadsheet, a graphing tool. If a control that you need doesn't exist, you can create it, with a little work. And with a little extra work, you can get the applet to work with the objects on the server side of a WebObjects application.

Another advantage of Java client-side components results from the way these components—that is, the applet controls—communicate with the server. Previously, the initial state of applets could be downloaded from the server, but that ended the communication. There was no way the applets could pass data back to the server or receive any subsequent updated state from the server.

Client-side components enable applet controls to synchronize any change in their state with the server and to receive back any further change in that state made by the server-side component. This state synchronization occurs in a request-response cycle in which the response-generation phase is used primarily to return state. As a result, state is synchronized without the page being reloaded.

Furthermore, client-side components allow users to activate applet controls to trigger action methods in the server component, and elicit responses that can result in the re-synchronization of state *or* the return of a new page.

It's easy to think of potential uses of client-side components. For example, you could have a tax-preparation application in which an assortment of applets represent a tax form. The aggregate applet itself performs calculations on the entered data, but it downloads timely information like tax tables from the server. When a form is complete, the user could trigger an action method to store the current tax record in a database.

If Client-Side Components Doesn't Work

After you install WebObjects, the Java client-side components feature should "just work" (assuming you have a Java-enabled browser). Unfortunately, files are inadvertently moved or deleted sometimes. The following checklist will assist you in situations like these. For Java client-side components to work, the following parts must be in place.

- You must have a Java-enabled browser.
- The following packages and classes must be installed in *DOCUMENT_ROOT*/

WebObjects/Java:
next.util.PropertyListUtilities.class
next.util.KeyValueCoding.class
next.wo.client.Association.class
next.wo.client.SimpleAssociation.class
next.wo.client.SimpleAssociationDestination.class

In addition, the next.wo.client.controls package, which contains the applet classes (TextFieldApplet, ButtonApplet, and so on), must be installed in the same location. See "Client-Side Applet Controls" for descriptions of each supported applet.

When NeXT's Java package is installed, it is written to *NEXT_ROOT*/NextLibrary/Java as well as to *DOCUMENT_ROOT*/WebObjects/Java. If your *DOCUMENT_ROOT* has been deleted or corrupted, you can copy the packages in *NEXT_ROOT*/NextLibrary/Java to *DOCUMENT_ROOT*/WebObjects/Java.

Of course, if you intend to create your own applets or Association subclasses, or do other Java development, you must obtain Sun's JDK (Java Development Kit) or a similar product.

If you install the development version of WebObjects, you get examples of applications that use client-side components in *NEXT_ROOT*/NextDeveloper/Examples/WebObjects and example applet controls in the next.wo.client.examples package (installed in both *NEXT_ROOT*/NextLibrary/Java and *DOCUMENT_ROOT*/WebObjects/Java). The TimeOffJava example application contains the source code for the controls in next.wo.client.examples. You are strongly encouraged to read the application's ReadMe file before attempting to compile this source code.

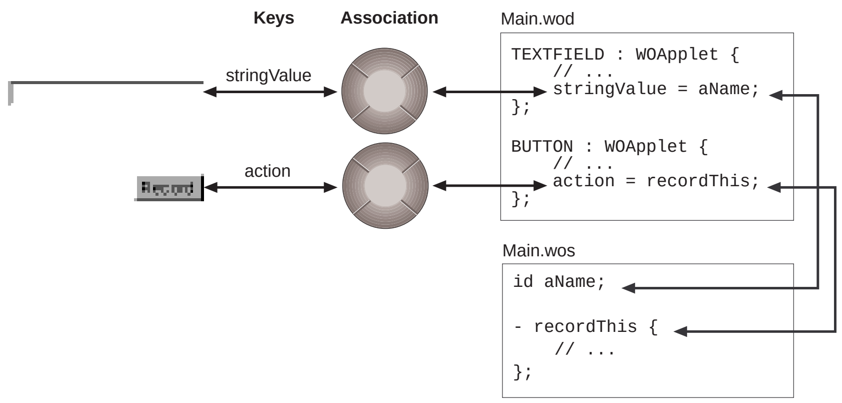
Note: If Java client-side components don't seem to work on your system, do not set the CLASSPATH environment variable to point to the next.* packages in *DOCUMENT_ROOT.* Doing so might seem to resolve the problem, but this is true only in those situations where the browser and server reside on the same machine.

How Client-Side Components Work

Client-side components have a duality in that they are represented by objects both on the client browser and in the server component. On the client side they are specially modified or interfaced applets. On the server side, they are represented by WOApplet dynamic elements.

A WOApplet dynamic element is a kind of "proxy" on the server for the applet. A WOApplet permits the specification of applet-specific parameters, such as the dimensions of the applet and the location of the ".class" file to download to the browser. It also allows you to initialize parameters to be downloaded to the applet and to bind an applet's keys to variables and methods in the server-side component. These bindings associate state in the applet with state in the server and events in the applet with the invocation of methods in the server. What makes these bindings possible is another parameter specified in a WOApplet declaration: an association class. By providing an association class, you endow an applet with the capabilities of state synchronization and action invocation.

Before delving further into the subject of Associations, first consider the keys defined by an applet and how state synchronization happens. An application determines the keys of an applet through the class of the applet (specified in the code attribute) and the applet's association class (specified in associationClass). Keys fall into two groups: state bindings and action bindings. State bindings form the basis for state synchronization by associating state in the applets with state in the server. Action bindings associate particular events in the client applet (such as button being clicked) with the invocation of methods in the server.



```
TEXTFIELD : WOApplet {
    // ...
    stringValue = aName;
};

BUTTON : WOApplet {
    // ...
    action = recordThis;
};
```

```
id aName;

- recordThis {
    // ...
};
```

Figure 1: Bindings Between Client and Server Components

State is synchronized between client and server in three phases:

1. When a page is first generated the server sends all state for which there are bindings to the client.
2. Before an action is invoked in the server, the client sends any of its state that has changed back to the server.
3. After the action is completed, the server sends of its state that has changed back to the client.

This last synchronization occurs only if no new page is returned to the browser. When a method invoked remotely through an applet action binding returns nil, it signals that, instead of returning a new page, the server should re-synchronized its state with the applets on the page. WebObjects "snapshots" the changes in state in the server so that only the state that has changed is sent back to the client.

Note: The last two phases of synchronization cycle can only be initiated on the browser side. That is, except for the first "initialization" phase, the server component can only react to an action triggered in an applet. The component cannot unilaterally update the state of an applet when its own state changes.

An Association object—specifically an instance of an Association subclass—provides the "glue" that secures the bindings of state and action between client and server. Associations know how to get and set the state of their applets at run time. They are also responsible for knowing when to fire their applets' supported actions. To enable this, Associations for particular applets maintain a list of "keys" (state attributes) that the applet manages and a list of actions that the applet can trigger. The value for a "key" must be a property-list type of object (either singly or in combination, such as an array of string objects). The corresponding property-list type of objects for Objective-C and Java are:

Table 1:

Objective-C	Java
NSString	String
NSArray	Vector
NSDictionary	Hashtable
NSData	byte[]

Associations, however, don't act alone. They mediate the exchange of state and action information between their applets and a hidden applet downloaded along with the applets that appear on the page. This hidden applet, AppletGroupController, controls the visible applets and handles communication back to the server. The AppletGroupController uses an Association to access each of the applets on the page. It is through these Associations that the data or state which each applet manages is passed to the AppletGroupController and, through it, to the server. When an Association fires its applet's action, the AppletGroupController does what is necessary to ensure that the bound method in the server is invoked. An AppletGroupController, once downloaded, knows what class of Association to use and what the destination applets are by inspecting the visible applets on the page and looking for some special parameters.

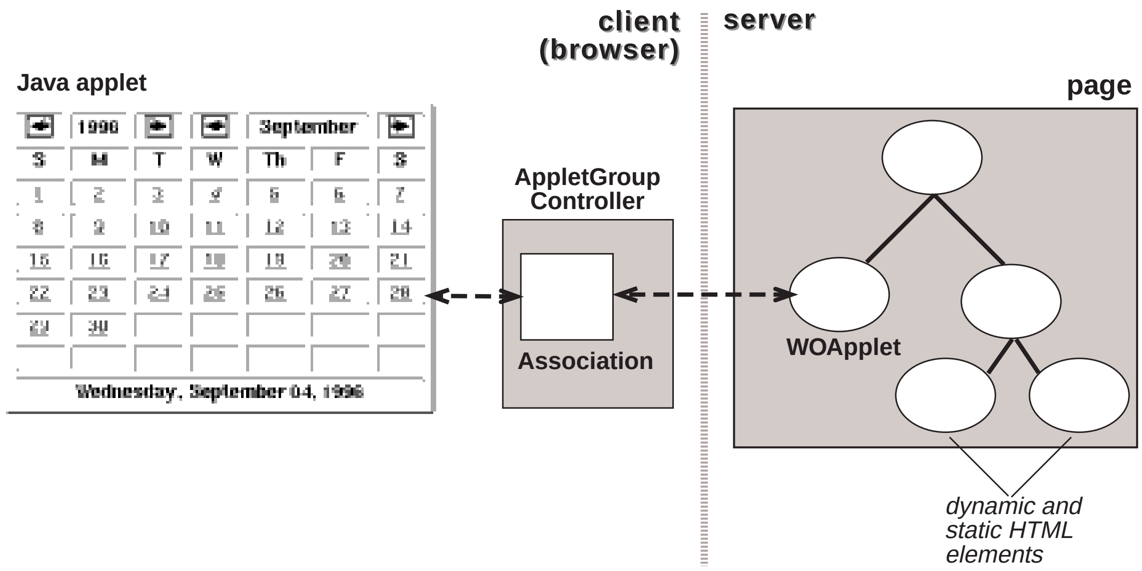


Figure 2: The Principal Objects Involved in Client-Side Components

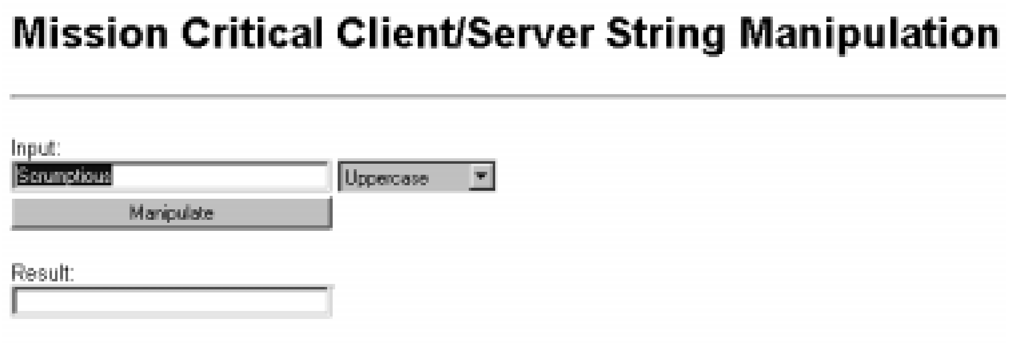
Sometimes an Association subclass can be tied to a particular "family" of applets instead a particular applet. Such is the case with applets provided by NeXT. These applets use the SimpleAssociation class, but get and set the key values themselves. To do this, they assume most of the duties of the Association by implementing the SimpleAssociationDestination interface. You can adopt this strategy with applets that you create. See "Making Your Own Java Client-Side Components" on page 9 for more information on this subject.

Integrating Client-Side Components

How do you get custom applet controls to work in a WebObjects application? This section describes the procedure. Before you begin, however, make sure that the **next.*** package and all other ingredients needed for client-side components are in place (see "If Client-Side Components Doesn't Work" on page 1 for specifics).

Note: Most applet controls provided by NeXT are available on a WebObjects Builder palette. However, the instructions below are geared toward hand-crafted HTML and declarations made in a ".wod" file. See "Loading and Unloading Palettes" in "Advanced WebObjects Builder Tasks" for instructions on loading palettes.

The following instructions use the CapitalizeString example application to illustrate each of the steps.



1. Place dynamic-element markers (WEBOBJECT) in the HTML file

You position and identify applet controls as dynamic elements in the HTML file of the component (page) just as you do any other dynamic element.

```
<!-- Other HTML goes here -->

   Enter a string in the field, select a function from the list and press the Do It but-
ton:

    <br>

    <WEBOBJECT name=INPUTFIELD></WEBOBJECT>

    <WEBOBJECT name=FUNCTION></WEBOBJECT>

    <br>

    <WEBOBJECT name=BUTTON></WEBOBJECT>

    <br>

    Result:

    <br>

    <WEBOBJECT name=OUTPUTFIELD></WEBOBJECT>

    <br>

<!-- Other HTML goes here -->
```

2. Make the appropriate WOApplet bindings

In the component’s “.wod” file, associate each identifier with a WOApplet dynamic element and initialize the standard WOApplet attributes as well as those specific to the applet class.

```
INPUTFIELD : WOApplet {

    code = "next.wo.client.controls.TextFieldApplet.class";

    codebase = "/WebObjects/Java";

    width = "200";
    height = "20";

    associationClass = "next.wo.client.SimpleAssociation";

    stringValue = inputString

};


FUNCTION : WOApplet {

    code = "next.wo.client.controls.ChoiceApplet.class";

    codebase = "/WebObjects/Java";

    width = "100";

    height = "20";

    associationClass = "next.wo.client.SimpleAssociation";

    itemList = functionItemList;

    selectedItem = functionSelectedItem;

    backgroundColor = "white";

};


BUTTON : WOApplet {

    code = "next.wo.client.controls.ButtonApplet.class";

    codebase = "/WebObjects/Java";

    width = "200";

    height = "20";

    associationClass = "next.wo.client.SimpleAssociation";

    title = "DoIt";

    action = "capitalizeString"

};


OUTPUTFIELD : WOApplet {

    code = "next.wo.client.controls.TextFieldApplet.class";

    codebase = "/WebObjects/Java";

    width = "200";
    height = "20";

    associationClass = "next.wo.client.SimpleAssociation";

    stringValue = outputString;

    enabled = "NO";

};
```

The following WOApplet attributes are standard and require assigned values. Although you could bind these attributes to variables, they are assigned constants in most cases. Constant values should be quoted.

code
: The class (including complete package prefix) of the applet.

codebase
: The *DOCUMENT_ROOT* subdirectory containing the package specified in the code attribute.

width
: The width of the applet, in pixels.

height
: The height of the applet, in pixels.

associationClass
: The subclass of Association for objects that get and set the state of applets and cause methods to be invoked in the server when actions are triggered in the applet. For the applet controls provided by NeXT, the Association subclass is SimpleAssociation. As the example shows, you must specify the package as well as the class. This attribute is optional. If you do not specify an Association subclass, however, then the applet is largely ornamental (as in WebObjects 2.0), since it is only able to accept initial values.

The reference chapter “Client-Side Applet Controls” describes the attributes representing the keys and actions of particular applets provided by NeXT.

3. Declare and initialize variables for bound keys

In the implementation file or “.wos” script of the server-side component, declare transaction variables for each of the key bindings made in the “.wod” file. Then you can initialize these variables in the init or awake methods. These initialized values are downloaded to the applets when the page is composed.

```
id inputString;

id outputString;

id functionItemList;

id functionSelectedItem;


- init {

    self = [super init];
    // Set up parameter values

    inputString = @"Scrumptious";

    outputString = @"";

    functionItemList = [NSArray arrayWithObjects:@"Uppercase", @"Lowercase",

        @"Capitalize", @"Flandersize", nil];

    functionSelectedItem = @"0";


    return self;

}
```

In this example, the initial values of both TextFieldApplets are set as well the items of the ListApplet. Setting the functionSelectedItem to zero causes the first item in the functionItemList array ("Uppercase") to be preselected.

4. Implement methods for bound actions

Implement action methods just as you do in other WebObjects scripts or Objective-C implementation methods. Just be sure to return nil instead of self or a response page. This signals to the server-side component that it should synchronize its state with the client applets after completion of the method.

```
- capitalizeString {

    if ([functionSelectedItem isEqualToString:@"1"]) {

        outputString = [inputString lowercaseString];

    } else if ([functionSelectedItem isEqualToString:@"2"]) {

        outputString = [inputString capitalizedString];

    } else if ([functionSelectedItem isEqualToString:@"3"]) {

        if ([inputString length] > 4) {

            outputString = [NSString stringWithFormat:@"%@-diddly-%@",

            [inputString substringToIndex:5], [inputString substringFromIndex:3]];

        } else {

            outputString = [inputString stringByAppendingString:@"-eroonie"];

        }

    } else {

        outputString = [inputString uppercaseString];

    }
    return nil;

}
```

A Note on Backtracking

It is a good idea to have a "welcome" page for applications that use client-side components and, indeed, for all WebObjects applications. Generally the client browser caches URLs and WebObjects caches the corresponding component instances to use as request pages. When the user backtracks through the pages of a session and reaches the first page, a new session will begin and you'll lose all state associated with the lost session. The reason for this is that the first page does not have a URL that includes the session ID, a context ID, and other important information.

In addition to having a "welcome" page as the first page, you can also disable caching by the client, setting the page expiration date to "now." This setting forces the URL request to be sent anew to the WebObjects application, which regenerates the request page. You disable client caching by sending **setPageRefreshOnBacktrackEnabled:** to the WOApplication object with an argument of YES (this happens automatically with pages that have client-side components on them).

Making Your Own Java Client-Side Components

If none of the applets in **next.wo.client.controls** meets your needs, you can write your own applet and implement the associative behavior that is necessary to make the applet a WebObjects client-side component. You can also turn applets that you find on the Internet or buy into client-side components. The strategies you may adopt differ for applets for which you have source code and those for which you don't.

If you have the source code for an applet, you will probably want to modify it so that it implements the SimpleAssociationDestination interface. Applets that implement this interface can use the SimpleAssociation class (a subclass of Assocation) to mediate the exchange of state and action information with the AppletGroupController and thus with the server. You are not required to take this course with applets for which you have the source; if you wish, you can create a subclass of Association and implement the mandatory methods (see "Creating a Subclass of Association" on page 11). However, applets that implement SimpleAssociationDestination do not require an Association subclass tailored for them. Instead they can use the generic SimpleAssociation class.

Implementing the SimpleAssociationDestination Interface

If you write an applet, or aquire the source code for an applet, you will probably want to follow this procedure to give the applet the associative behavior it needs to be a client-side component.

1. In the class declaration, insert the "implements SimpleAssociationDestination" clause.

```
public class MyApplet extends Applet implements SimpleAssociationDestination {

   // ....

}
```

2. Implement the **keys()** method to return a list (Vector) of state keys managed by the applet.

```
    public Vector keys() {
        Vector keys = new Vector(1);
        keys.addElement("title");
        return keys;
    }
```

3. Implement the **takeValueForKey(**Object, String**)** and **valueForKey(**String**)** methods to set and get the values of keys.

```
    synchronized public Object valueForKey(String key) {
        if (key.equals("title")) {
            return this.getLabel();
        }
    }


    synchronized public void takeValueForKey(Object value, String key) {
        if (key.equals("title")) {
            if ((value != null) && !(value instanceof String)) {
                System.out.println("Object value of wrong type set for key
                        'title'.  Value must be a String.");
            } else {
                self.setLabel(((value == null) ? "" : (String)value));
            }
    }
```

You should be able to access the keys directly or, ideally, through accessor methods ("getLabel()" and "setLabel()" in the above example). It is a good idea to use the **synchronize** modifier with takeValueForKey(Object, String) and valueForKey(String) because these methods can be invoked from other threads to read or set data.

The remaining steps apply only if the applet has an action.

4. Declare an instance variable for the applet's Association object and then, in **setAssociation(**Association**)**, assigned the passed-in object to that variable.

```
    protected Association _assoc;
    // ...
    synchronized public void setAssociation(Association assoc) {

        _assoc = assoc;

    }
```

The Association object must be stored so it can be used later as the receiver of the invokeAction() message. The Association forwards the action to the AppletGroupController, which handles the invocation of the server-side action method.

5. When an action is invoked in the applet, send invokeAction(String) to the applet's Association.

```
    synchronized public boolean action(Event evt, Object what) {

        if (_assoc != null) {

            _assoc.invokeAction("action");

        }

        return true;

    }
```

Creating a Subclass of Association

If you have an applet, but do not have the source code for it, you must follow the following strategy for making the applet a client-side component. You must know the applet's accessor methods for setting and getting state, and, if the applet triggers actions, there must be some way for your Association to detect this. *If the applet doesn't have API for getting and setting state, you cannot make the applet into a client-side component.*

1. Declare an subclass of the Association class.

```
    class MyAssociation extends Association {

        // ...

    }
```

2. Implement the keys() method to return a list (Vector) of keys managed by the applet. See "Implementing the SimpleAssociationDestination Interface" on page 9 for an example.

3. Implement the takeValueForKey(Object, String) and valueForKey(String) methods to set and get the values of keys. Use Association's destination() method to obtain the destination object (that is, the applet).

```
    synchronized public Object valueForKey(String key) {

        Object dest = this.destination();

        if (key.equals("title")) {
            return ((MyApplet)dest).getLabel();
        }
    }

    synchronized public void takeValueForKey(Object value, String key) {
        Object dest = this.destination();
        if (key.equals("title")) {
            if ((value != null) && !(value instanceof String)) {
                System.out.println("Object value of wrong type set for key
                        'title'.  Value must be a String.");
            } else {
                ((MyApplet)dest).setLabel(((value == null) ? "" : (String)value));
            }
    }
```

Note that the class of the destination applet ("MyApplet" in the example) must be cast.

If the applet triggers an action method, it must some mechanism for communicating this event to observers (such as an "observeGadget()" method).

4. The Association responds to the triggering of the applet's action by sending invokeAction(String) to itself.

```
    public void observeGadget(Object sender, String action) { // fictictious
        if ((sender instanceof Gadget) && action.equals("vacuum")) {
            this.invokeAction(action);
        }
    }
```

Note that in this hypothetical example, the Association must first set itself up as an observer.