11.8. Repositioning Objects and Keeping Those Positions

This next user customization is for more modern applications that apply Web 2.0 functionality in them. Some of this functionality is in moving objects around on a page. When the application in question truly is living up to its name as an application, any changes the user makes should be saved for the next time that same user uses the application. One of these user-customizable options is the position of objects on a page.

For now, let's assume the user can move objects on the screen without going into detail regarding how this is accomplished. We need to keep track of the final *x* and *y* coordinates of the object once it has moved, as well as what object was moved. The easiest way to store this information is in a simple multidimensional array, where the first index stores the object being moved and the second index stores an array containing the object's *x* and *y* coordinates. When the page initially loads, there should be a call to the database to move any objects that were moved from their original position when the user last used the application. It is that simple to save position!

11.8.1. Dragging Objects Around

OK, seriously, first we need to allow the object to be moved or dragged around in the application. In Chapter 10, I introduced you to the `Draggable` script.aculo.us object. Using this object is the easiest way to move an object to a new position, and by using the `snap` object, you can exert the tiniest bit of control over where the object is to be placed.

Here is the code to create a `Draggable` object that stores the final *x* and *y* coordinates to a variable when the object stops moving:

```
new Draggable('objectContainer', {
    handle: 'objectHandle',
    snap: 20,
    starteffect: false,
    endeffect: function(p_element) {
        window.status = Position.cumulativeOffset(p_element);
    }
});
```

With this code, the final coordinates are set in the `window.status` property, but it is easy to imagine a more useful way to deal with them.

11.8.2. Storing Information in a Database

We could save the *x* and *y* coordinates of a draggable object in a cookie, like all the other customization options we have seen so far, but we are going to aim for something a little more permanent here. A database is a logical place to store this information, but what information are we to store for our draggable object?

First, we need to assume that there is a way to uniquely identify the user that is manipulating the application. So, we will assume that the user has a login ID stored somewhere, such as in a `Session` variable. Then we must store the element information: the element `id`, *x* coordinate, and *y* coordinate.

Here is a simple table to store this information.

Column	Type
loginID	`INTEGER`
elementID	`VARCHAR(25)`
xCoord	`SMALLINT`
yCoord	`SMALLINT`

We can create this table with the following SQL:

```
CREATE TABLE draggable_position (
    loginID          INTEGER             NOT NULL,
    elementID        VARCHAR(25)         NOT NULL,
    xCoord           SMALLINT            NOT NULL,
    yCoord           SMALLINT            NOT NULL,
    PRIMARY KEY (loginID),
    KEY (elementID)
);
```

Our client is going to need the server to perform two SQL commands. The first is to send the elements in the table for the user with the matching `loginID` and their *x* and *y* coordinates. The second is to save (or update) the table with the element and its coordinates for the user's `loginID`.

Retrieving the information is as simple as the following SQL:

```
SELECT
    d.elementID,
    d.xCoord,
    d.yCoord
FROM
    draggable_position d
WHERE
    d.loginID = $loginID;
```

Inserting information would require the following SQL:

```
INSERT INTO draggable_position (
    loginID,
    elementID,
    xCoord,
    yCoord
) VALUES (
    $loginID,
    $elementID,
    $xCoord,
    $yCoord);
```

Updating an existing row would require the following SQL:

```
UPDATE
    draggable_position
SET
    xCoord = $xCoord,
    yCoord = $yCoordWHERE
    loginID = $loginID AND
    elementID = $elementID;
```

This is the basic idea for storing information in a database. Your mileage may vary.

11.8.3. Sending Changes with Ajax

Sending the changes in the position of an object in the application to the server to store in a database is a snap. Here is where we can actually utilize the object's final coordinates. The inline function in the `endeffect` option will now contain an Ajax call to the database, passing it the information it needs:

```
endeffect: function(p_element) {
    var coorinates = Position.cumulativeOffset(p_element);
```

```
    Ajax.Request('savePosition.php',
        method: 'POST',
        parameters: 'id=' + p_element.id + '&x=' + coordinates[0] + '&y=' +
            coordinates[1]
    );
}
```

The server must take the passed parameters and either insert or update the database with this data. Example 11-10 shows what this would look like.

Example 11-10. The server-side code to store element position in a database

Code View:

```
<?php
/**
 * Example 11-10.  The server-side code to store element position in a database.
 */
/**
 * The Zend Framework Db.php library is required for this example.
 */
require_once('Zend/Db.php');
/**
 * The generic db.php library, containing database connection information such as
 * username, password, server, etc., is required for this example.
 */
require('db.inc');

/* Set up the parameters to connect to the database */
$params = array ('host' => $host,
                 'username' => $username,
                 'password' => $password,
                 'dbname' => $db);

try {
    /* Were the parameters passed that needed to be? */
    if ($_POST['id'] && $_POST['x'] && $_POST['y']) {
        $login = $_SESSION['login_id'];
        $id = $_POST['id'];
        $x = $_POST['x'];
        $y = $_POST['y'];

        /* Connect to the database */
        $db = Zend_Db::factory('PDO_MYSQL', $params);
        /* Create the SQL string */
        $sql = "SELECT loginID FROM draggable_position WHERE loginID = $login "
            ."AND elementID = \"$id\"";
        /* Get the results of the query */
        $result = $db->query($sql);
        $new = -1;
        /* Was a row returned? */
        if ($row = $result->fetchRow( ))
            $new = $row['loginID'];
        /* Should this record be updated? */
        if ($new != -1) {
            $set = array (
                'xCoord' => $db->quote($x),
                'yCoord' => $db->quote($y)
            );
            $table = 'draggable_position';
            $where = "loginID = $login AND elementID = \"$id\"";

            /* Update the record */
            $rows_affected = $db->update($table, $set, $where);
        } else {
            $row = array (
```

```
                'loginID' => $login,
                'elementID' => $id,
                'xCoord' => $x,
                'yCoord' => $y
            );
            $table = 'draggable_position';

            /* Insert the record */
            $rows_affected = $db->insert($table, $row);
        }
    }
} catch (Exception $e) {}
?>
```

The server does not need to send anything back to the client with this example, because the client really can't do anything, even if there is a problem with the database or server-side script.