Enhancing WebGen5 with Access Control, AJAX Support, and Editable-and-Insertable Select Form.

by
Mariko Imaeda

Submitted to

Oregon State University

In partial fulfillment of the requirements for the degree of

Master of Science

Presented December 4, 2007
Commencement June 2008

Abstract

WebGen is a software tool for generating Web scripts automatically for a Web-based database application. In this project, access control, AJAX support, and editable-and-insertable table mechanisms were added to WebGen. With our access control mechanism, an access-control level can be specified for each table. In access control level 1, for example, a user can read any records, and a logged-in user can insert records and update and delete the records inserted by her. There are five access control levels. WebGen now can generate an AJAX server-side PHP script that retrieves, based on a given value, one or multiple records from the database. The given value may be selected from a dropdown list in a form, and the retrieved value or values can be set in an `input` element or in a `select` element as options, respectively. With an *editable*-and-*insertable select* form, a user can now read, insert, update, and delete multiple records in a table at one time.

Table of Contents:

1. Introduction

WebGen 5 is a software tool generating scripts for Web forms that are used for managing records stored in a database. Five Web scripts namely, *search*, *select*, *edit*, *info*, and *action* scripts, can be generated for each table. A user can provide search parameters with the *search* form, and the retrieved records are displayed in the *select* form. She can view the detailed information relate to one record with the *edit* or *info* form. The *edit* form allows her also to insert, update, or delete a record. The *action* script is activated for inserting, updating, and deleting records in the database. These web scripts can be generated from a *configuration* file that specifies how the fields in the forms should be generated. The configuration file is produced from the metadata of the database.

Four Web script generators precede WebGen 5. WebSiteGen 1 was the first attempt. It generated Web scripts from an *ER diagram*. However, this approach was not effective, because an ER diagram may not accurately reflect the real structure of a database. Starting with WebSiteGen 2, relational database schemas were used to generate Web scripts. WebSiteGen 2 was a Windows application written in Java that generated ASP Web scripts. About one year later, WebSiteGen 3, which generated more complex ASP.NET Web scripts supporting *one-to-many* and *many-to-one* relationships between tables, was developed. WebSiteGen 3 was written in C#, and it was actually used to generate Web scripts for real projects.

When WebSiteGen 3 was partially completed, PHP Web scripts had to be generated, and the work on WebSiteGen 4 was started. However, WebSiteGen 4 was not very successful. The code became long and hard to understand. A change in one part often caused ripple effects throughout the entire Web script generator, and hence the generator was difficult to maintain.

While trying to overcome the problems in WebSiteGen 4, we came across a new idea of using *templates* for generating Web scripts. Because a template resembles the generated Web scripts, creating a set of templates is easier than writing a generator in a

conventional programming language. Moreover, as one template only generates one type of web scripts, changes in one template do not affect other templates, unless the changes are related to parameters passed between scripts.

In this project, we added mechanisms for *access control*, AJAX support, and *editable*-and-*insertable* table to WebGen 5.

In order to restrict access to records stored in the database by a user, we implemented access control mechanism. We provide five access control levels 0 – 4, one of which can be specified in the configuration file for a table. We also categorize each user in one of the `public`, `owner`, or `admin` group. If the access control level for the table is 0, no access restrictions are applied, and a user can read, insert, update, and delete any records. When the access control level for the table is one of the levels 1 – 4, accessing a record in the table by a user in the `public` or `owner` group is restricted. At any access control level, an `admin` user can read, insert, update, and delete any records.

When form scripts for a table are generated by `webgen`, one or more AJAX server-side PHP scripts can be generated. Each AJAX server-side script retrieves one or multiple records based on a given value and returns the values computed from them. In the *search* or *edit* form, one dropdown list and another `select` or `input` element are associated with the server-side script as a source field and as a target field, respectively. When the value of the source field is modified, the script is activated and returns the result to the Web form. The returned values are handled by a common AJAX client-side JavaScript code and set in the target element.

Previously, a *select* form displayed multiple records, each as a row in a table, and a user could only view and delete those records. In order to allow a user to insert and update records in addition to viewing and deleting them, we can now generate an *editable*-and-*insertable select* form. An editable-and-insertable table is called a data-grid.

Each table cell is converted to an `input` or `select` element so that a user can modify its value. Furthermore, a new row can be added at the end of the table.

In Section 2, we explain the details of our access control mechanism. Section 3 describes the details of the AJAX support mechanism. An editable-and-insertable *select* form is discussed in Section 4. In Section 5, conclusions are provided, and possible future work is discussed.

2. Access Control Mechanism

It is often required to allow a user to access only certain rows in tables. Our security mechanism is organized as follows.

1. Users are categorized into three groups, `public`, `admin`, and `owner`. Each user in the `admin` group and the `owner` group need to have an account and log-in. No access restriction is applied to the `admin` users at any level. The users in `public` group are the users who have not logged-in.
2. One of the five access control levels 0 – 4 can be applied to the forms of each table. The access control level can be defined in the configuration file for the table. The access control levels are applicable to the users in the `owner` and `public` groups.
3. The ID of a user is stored in every record owned by that user.

Information on the users is maintained in table `login_user`. The details about this table are discussed in Section 2.1.

Access to a table by users in the `owner` and `public` groups is restricted by the access control level defined for the table.

Level 0. No access restriction is applied. Every user can insert/read/update/delete any record in the table.

Level 1. An `owner` user can read any record and insert a new record, but she can update and delete only the records owned by her. A `public` user can read any record.

Level 2. An `owner` user can insert a record, and she can read/delete/update only the records owned by her. No permission is given to a `public` user.

Level 3. A `public` user and an `owner` user can read any record. However, they cannot insert, update, or delete a record. All the records of this access control level need be owned by `admin` users.

Level 4. No permission is given to a `public` user or an `owner` user.

We explain about these access control levels more in Section 2.3.

2.1 User Information

In our security control mechanism, table `login_user` maintains information on all the `admin` and `owner` users. If any table has access control level other than level 0, a user registration table and table `d_login_user_role` need be created. Figure 2.1 gives the `CREATE` statement for a sample user registration table. Any table can be used as the user registration table as long as it contains columns `login_name`, `password`, and `d_login_user_role_id`.

```
CREATE TABLE login_user{
  login_user_id integer,
  login_name varchar,
  password varchar,
  name varchar,
  address varchar,
  city varchar,
  state varchar,
  zip_code varchar,
  phone varchar,
  fax varchar,
  email varchar,
  row_owner_id integer,
  d_login_user_role_id integer,
};
```

Figure 2.1: `CREATE` statement for table `login_user`

```
CREATE TABLE d_login_user_role {
  d_login_user_role_id integer,
  user_role_name varchar,
};
```

d_login_user_role_id	user_role_name
1	admin
2	owner

Figure 2.2: Table `d_login_user_role` and the two records.

Table d_login_user_role stores the possible user roles, in our case, admin and owner as shown in Figure 2.2. The IDs of the records can be stored in column d_login_user_role_id of table login_user.

Each user in the admin or owner group must have a record in table login_user. Important columns in table login_user are the followings:

login_user_id

The primary key column.

login_name

The value is used as the login name for log-in.

password

The value is used as the password for log-in.

d_login_user_role_id

The foreign-key column linked to column d_login_user_role_id in table d_login_user_role. The value is 1 for an admin user or 2 for an owner user.

row_owner_id

The same value in column login_user_id.

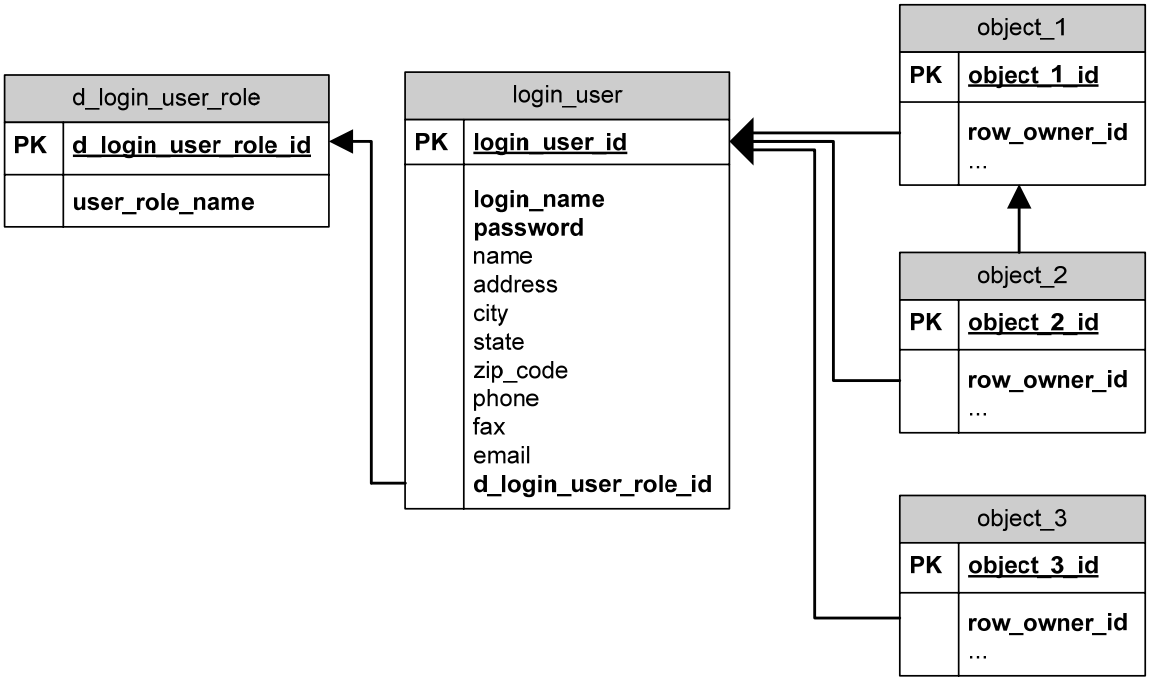

Figure 2.3: ER schema diagram for table `login_user`.

In order to implement access control, the owner is defined for each record. For this purpose, column `row_owner_id` is added to each table that requires access restriction, as shown in Figure 2.3. When a new record is inserted in the table, the ID of the user, which is the `login_user_id` of that user, is set as the value of `row_owner_id`.

2.2 Registration and Log-In

If a user wants to access tables protected with one of access control levels 1 – 4, she must create an account from a registration page as shown in Figure 2.4. With this registration page, `d_login_user_role_id` for the user is automatically set to 2, which indicates `owner`. The value entered for login name is checked if it is unique. After creating an account, she can log-in from the login page shown in Figure 2.5.

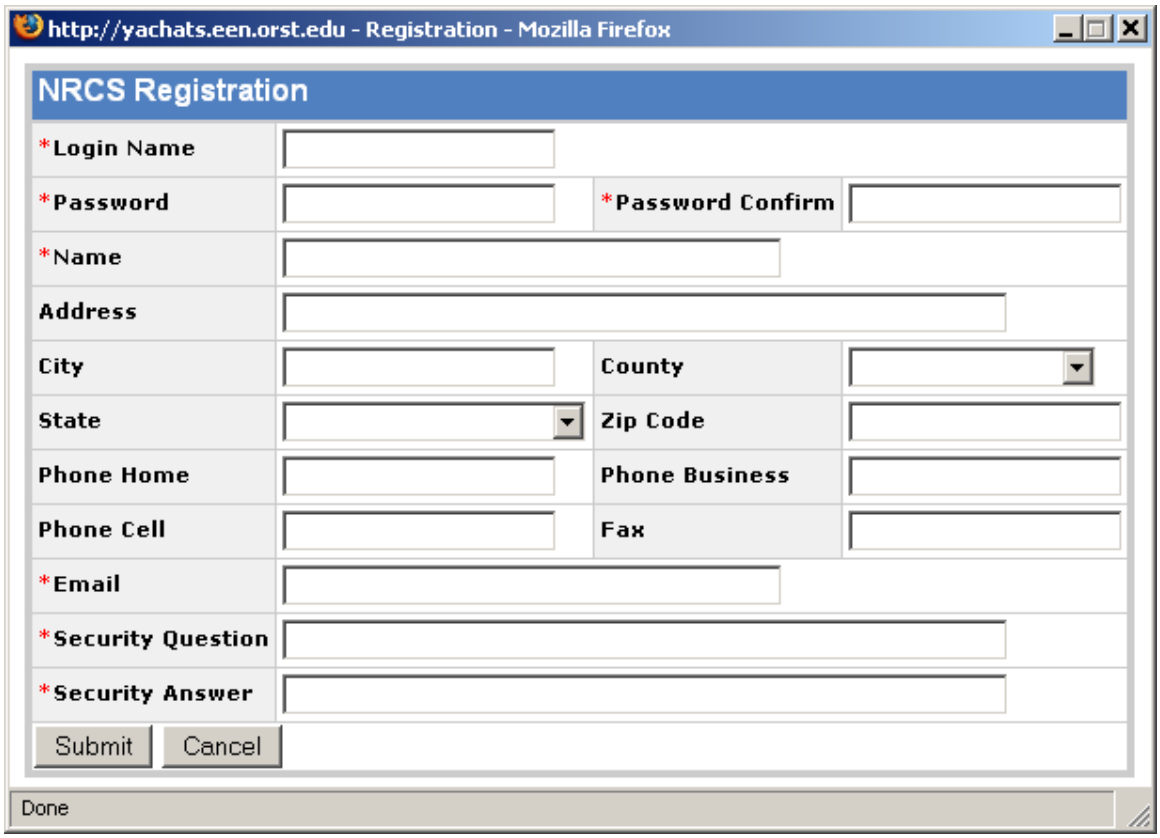

Figure2.4: The registration page.

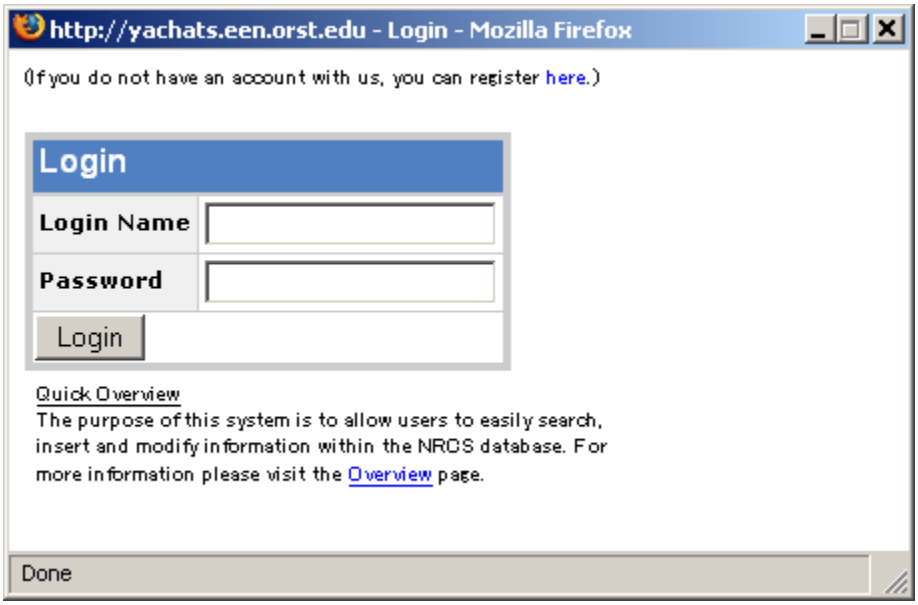

Figure2.5: The login page.

At the login page, a user enters her login name and password, and a session data is initialized. With the login name and the password, the record of the user is searched from table `login_user`. If the user is authorized, then the ID and the role of the user are stored in the session as shown in the code of Figure 2.6.

`$_SESSION['UID']` – the ID of the user.

`$_SESSION['UROLE']` – the role of the user, `owner` or `admin`.

`$_SESSION['UNAME']` – the login name of the user.

```
session_start();
...
if (($role = get_role($form['login_name'], $form['password'])) != '') {
  $_SESSION['UID'] = get_user_id($form['login_name'], $form['password']);
  $_SESSION['UNAME'] = $form['login_name'];

  switch($role) {
   case "1":
    $_SESSION['UROLE'] = "admin";
    break;
   case "2":
    $_SESSION['UROLE'] = "owner";
    break;
  }
  ...
}

function get_role($login_name, $password) {
$sql_select = "SELECT role FROM login_user
               WHERE login_name = '$login_name'
                 AND password = '$password';";

  $db->query($sql_select);
  if ($db->num_rows() == 1) {
    $db->next_record();
  return $db->f('role');
} else
  return null;
}

function get_user_id($login_name, $password) {
$sql_select = "SELECT login_user_id FROM login_user
               WHERE login_name = '$login_name'
                 AND password = '$password';";

  $db->query($sql_select);
  if ($db->num_rows() == 1) {
    $db->next_record();
  return $db->f('login_user_id');
} else
  return null;
}
```

Figure 2.6: `login.phtml`

2.3 Access Control Levels

In order to implement access control, one of the five access control levels `0 – 4` need be specified with variable `$access_control_level` in the configuration file for a table. Also, each user need be classified as `admin`, `owner`, or `public`. Figure 2.7 shows the access control applied under this condition.

Level	User Group		
	`admin`	`owner`	`public`
0	Any actions	Any actions	Any actions
1	Any actions	Read any records Insert new records Update/Delete owned records	Read any records No Insert actions No Update/Delete actions
2	Any actions	Read owned record Insert new records Update/Delete owned records	No actions
3	Any actions	Read any records No Insert action No Update/Delete actions	Read any records No Insert actions No Update/Delete actions
4	Any actions	No actions	No actions

Figure 2.7: Possible user actions at each level.

According to the access control level defined in the configuration file, access restrictions are enforced by the web scripts generated by `webgen`. In the following, we describe how the *search*, *select*, *edit*, *info*, and *action* scripts for each table implement access control. Since any action is allowed for an `admin` user, possible actions for an `admin` user are not described.

Access control level 0 (Default)

No restriction is applied. Anyone can insert/read/update/delete records in the table. This is the default access control level.

Access control level 1

At this level, a user in any group can read any records in the table. However, only a logged-in user can insert records, and the records inserted are owned by that user. An `owner` user can update and delete only records owned by her.

Search script

The *search* form can be used by all users.

Select script

1. Records selected can be listed for any user.
2. For an `owner` user and an `admin` user, the *Insert New* button is shown.
3. For an `admin` user, the *Delete* button is shown.

Edit script

1. For a `public` user, the *edit* form is not accessible. When a primary key value for a record is passed to the *edit* script, the *info* form is loaded.
2. When the script is activated for updating an existing record by an `owner` user, the *info* form is loaded if the ID of the `owner` user does not match the value of `row_owner_id` of the record.
3. For a record to be inserted or updated by an `owner` user, the ID of the user is stored as the value of `row_owner_id` and `modified_by` of the record, and the current date is stored as the value of `modified_date`.
4. For a record to be inserted or updated by an `admin` user, the values of form parameters `row_owner_id`, `modified_by`, and `modified_date` are used as the values of the record.
5. Deletion of a record by an `owner` user can be performed only when the value of `row_owner_id` of the record matches the ID of the `owner` user.

Info script

No access control is required for any user.

Action script

1. For a `public` user, the *action* form is not accessible.
2. When the *Delete* button in the *select* form is clicked by an `admin` user, each of the selected records is deleted by this *action* script.

Access control level 2

At this level, a `public` user cannot take any action. The login-in page is loaded when a `public` user tries to access a form. An `owner` user can insert records and access only those records that are owned by her. An `owner` user cannot read records owned by others.

Search script

For a `public` user, the *search* form is not accessible. When a `public` user accesses it, the login-in page is loaded.

Select script

1. For a `public` user, the *select* form is not accessible.
2. For an `owner` user, in addition to the parameters passed from the *search* form, the ID of the user is set as the search parameter value of `row_owner_id`, and hence only the records owned by that user are retrieved.
3. With the *Delete* button, an `admin` user can delete any selected records, and an `owner` user can delete selected records owned by her.
4. If the *select* form is *editable*, a user can update and delete multiple records from the form. Furthermore, if it is *editable* and *insertable*, a new record can be inserted with the *select* form. These actions are performed when the *Apply* button is clicked. The *Apply* button is shown for an editable *select* form. The details of an *editable* and *insertable select* form are discussed in Section 4.

Edit script

The *edit* script works like the one whose access control level is `1`, except for the following differences.

1. For a `public` user, the *edit* form is not accessible.
2. When the script is activated for updating an existing record by an `owner` user, the error message is given if the ID of the `owner` user does not match the value of `row_owner_id` of the record.

Info script

1. For a `public` user, the *info* form is not accessible.
2. For an `owner` user, the error message is given if the ID of the `owner` user does not match the value of `row_owner_id` of the record.

Action script

1. For a `public` user, the *action* form is not accessible, so the error message is given.
2. When deletion of records is requested from the *select* form, each of the selected records is deleted in this *action* script.
3. When the *Apply* button in the *select* form is clicked, the applicable action for each record is executed in this *action* script. The details are described in Section 4.

Access control level 3

At this level, a user can read any records. Only an `admin` user can insert a new record or update and delete existing records.

Search script

The *search* form can be used by any user.

Select script

1. Records selected can be listed for any user.
2. For an `admin` user, the *Insert New* button and the *Delete* button are shown.

Edit script

For a user in the `public` or `owner` group, the *edit* form is not accessible. When the script is activated for updating by a user in the `public` or `owner` group, the *info* form is loaded for display only. When the script is activated for inserting a record, an error message is given.

Info script

No access control is required for any user.

Action script

For a user in the `owner` or `public` group, the *action* script is not accessible. When a user in the `public` or `owner` group accesses it, the error message is given.

Access control level 4

At this level, only `admin` users can access records. A user in the `public` or `owner` group cannot even read records.

All the *search*, *select*, *edit*, *info*, and *action* forms are accessible for only `admin` users. When the script is activated by a user in the `public` or `owner` group, the error message is given, or the login-in page is loaded.

2.4 Columns Readable only by an Admin User

Fields for some columns can be hidden from a user in the `public` or `owner` group, while those fields are displayed for an `admin` user. For example, the field for column `row_owner_id` need not be shown for an `owner` user or should not be edited by her. However, the value of `row_owner_id` should be readable and editable by an `admin` user, since she might need to know who owns the record and change the owner. Sample forms accessible by an `owner` user and an `admin` user are shown in Figure 2.8.a and 2.8.b, respectively.

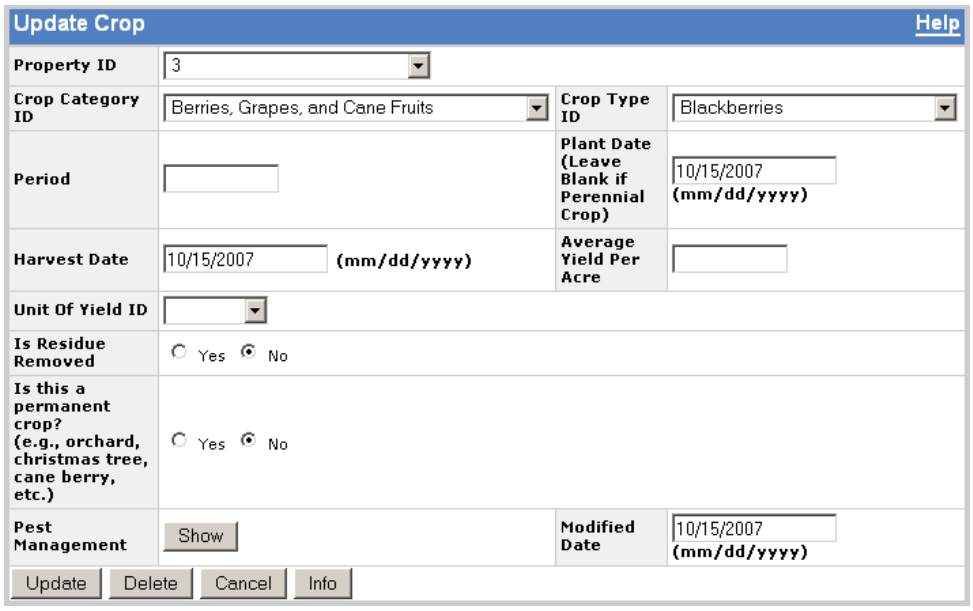

Figure 2.8.a: The *edit* form for an `owner` user.

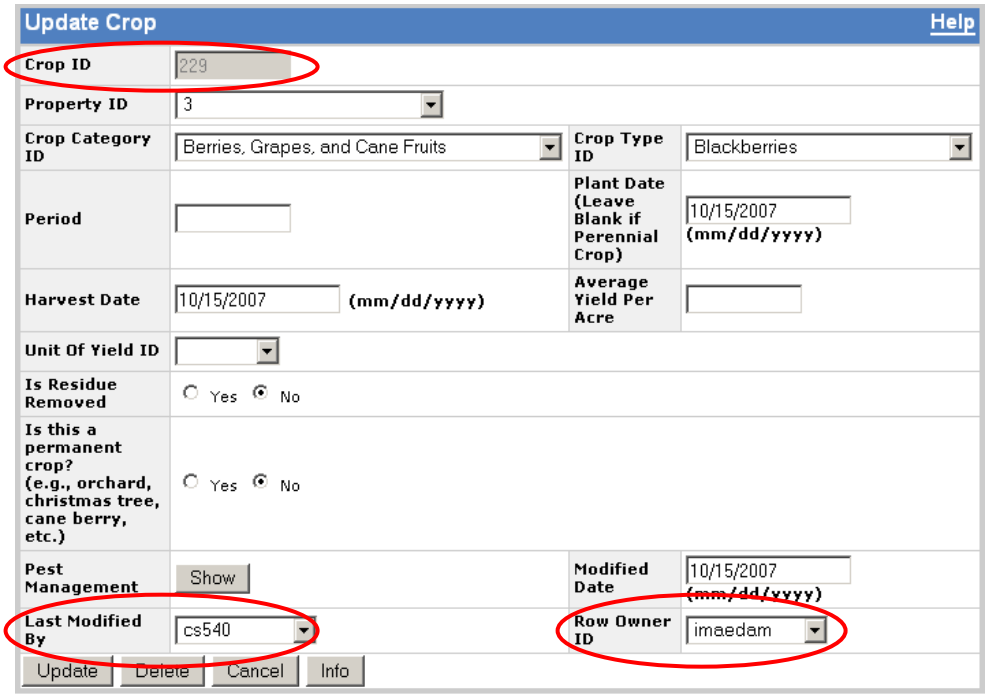

Figure 2.8.b: The *edit* form for an `admin` user.

In order to hide fields from `public` and `owner` users, attribute `admin_only` need be set for each of those columns. When a user in the `public` or `owner` user activates a *search*, *select*, *edit*, or *info* script, the fields for the columns whose `admin_only` attributes are set to `true` are not generated by the script. Also, when an SQL query for inserting and updating a record is formulated, those columns are not included in it.

Furthermore, the values of columns `row_owner_id`, `modified_by`, and `modified_date` need be automatically set when an `owner` user insert or update a record.

When an `owner` user updates a record with an *edit* form, she might try to update a record owned by another user by providing the primary key value in the URL. In order to prevent such an action, the value of `row_owner_id` of the record to be updated is retrieved from the database and checked before the SQL query is executed. Although this check does not prevent the user from updating another record owned by her, a record owned by another user cannot be updated.

3. AJAX Support

We often have to provide a set of possible options for a dropdown list in a form according to the selected value in another dropdown list. For example, after a state is selected with the form shown in Figure 3.1, we have to provide for selection only the counties in that state.

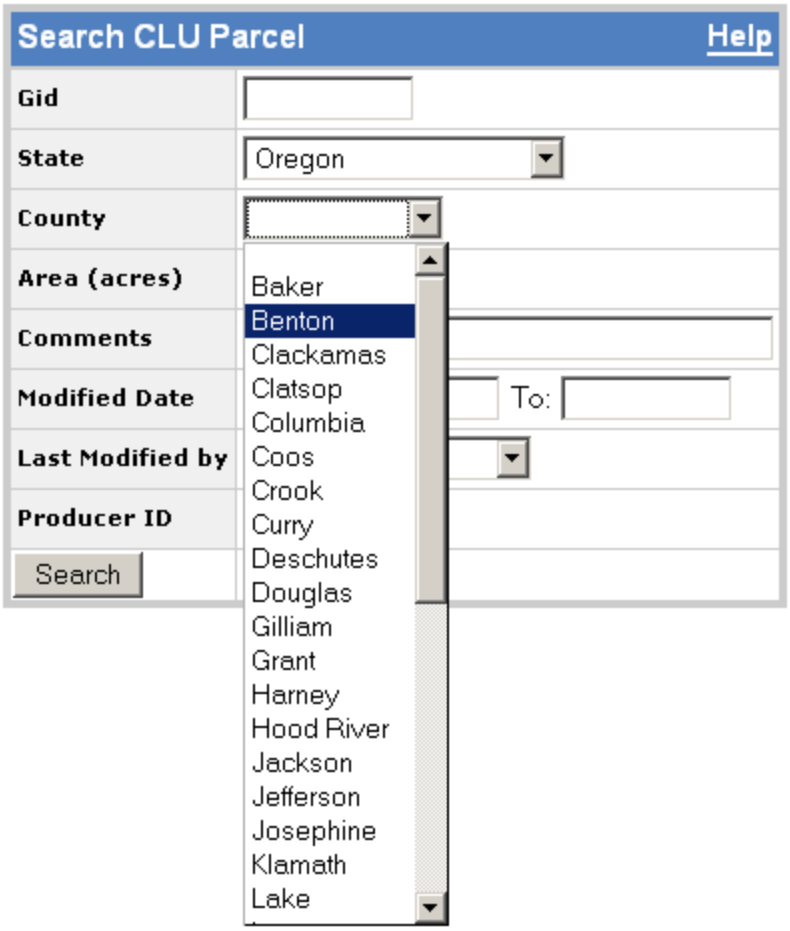

Figure 3.1: List of the counties in the state selected.

We implemented this mechanism by using AJAX as shown in Figure 3.2. When the user selects a state from a dropdown list in the form, the ID of the state is sent as an AJAX request to PHP script `clu_parcel_ajax_d_county.php`. The form script contains JavaScript `ajax_client.js` to issue the AJAX request. The PHP script then

retrieves the list of the counties in the state from the database and returns it to the form. Then, the counties returned are set in the `county` dropdown list.

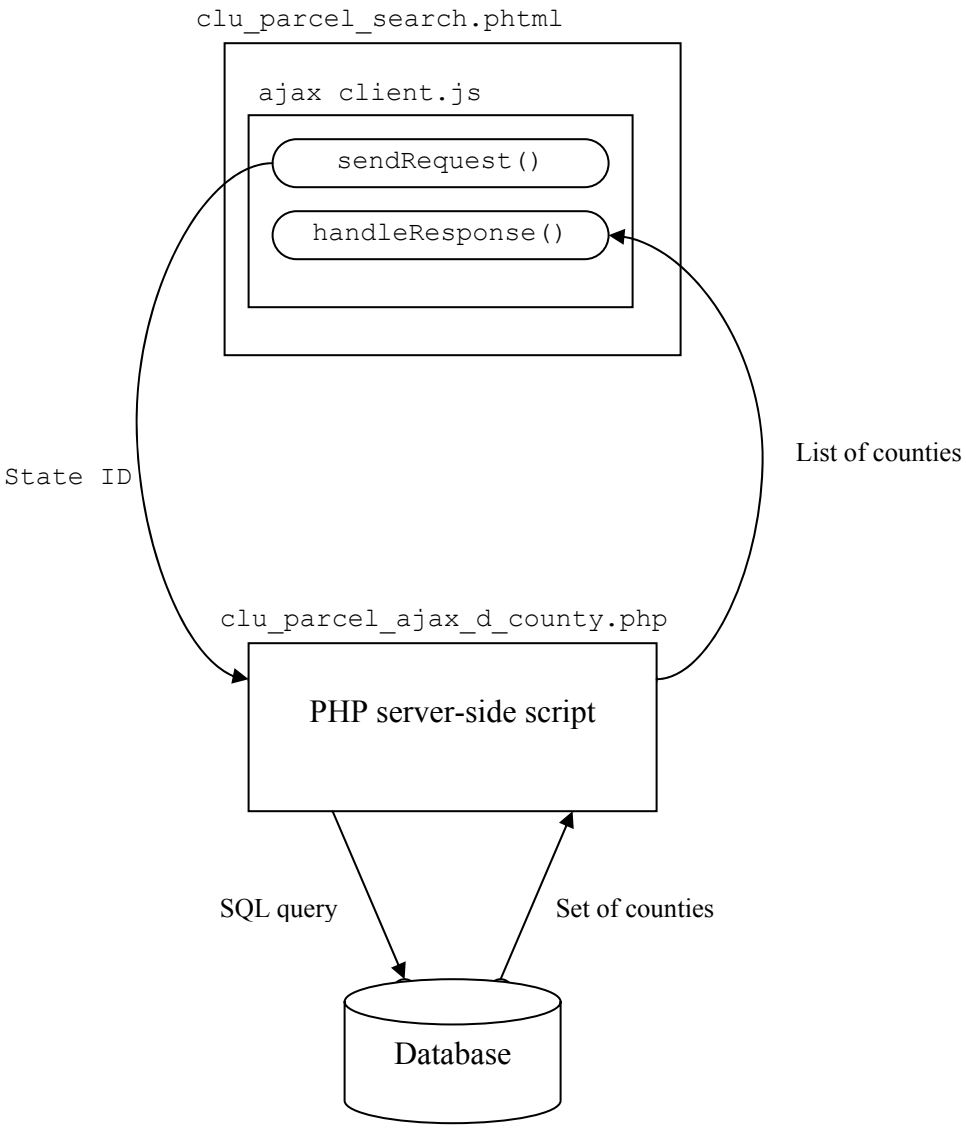

Figure 3.2: AJAX request processing.

AJAX client-side JavaScript file `ajax_client.js`

A Web page can include this JavaScript file to issue an AJAX request. The AJAX request is sent with function `sendRequest()`, and its response is received with function `handleResponse()`.

AJAX server-side PHP file *xxx*`_ajax_`*yyy*`.php`

This PHP script is activated by an AJAX request. SQL queries are formed with the parameters passed in the request, and those queries are executed to retrieve records from the database. The response formulated from the retrieved records is sent back to `handleResponse()`.

WebGen is a software tool for automatically generating Web scripts that display Web forms and operate on data stored in the database. The previous version of WebGen can generate five types of Web scripts: *search*, *select*, *edit*, *information*, and *action* scripts shown in Figure 3.3 for each table from a configuration file. A template written in PHP is provided for each type of Web scripts. The generated scripts are executed on the Web server by a PHP interpreter. Each script, except for an *action* script, generates a Web form that is displayed on a client computer by a Web browser.

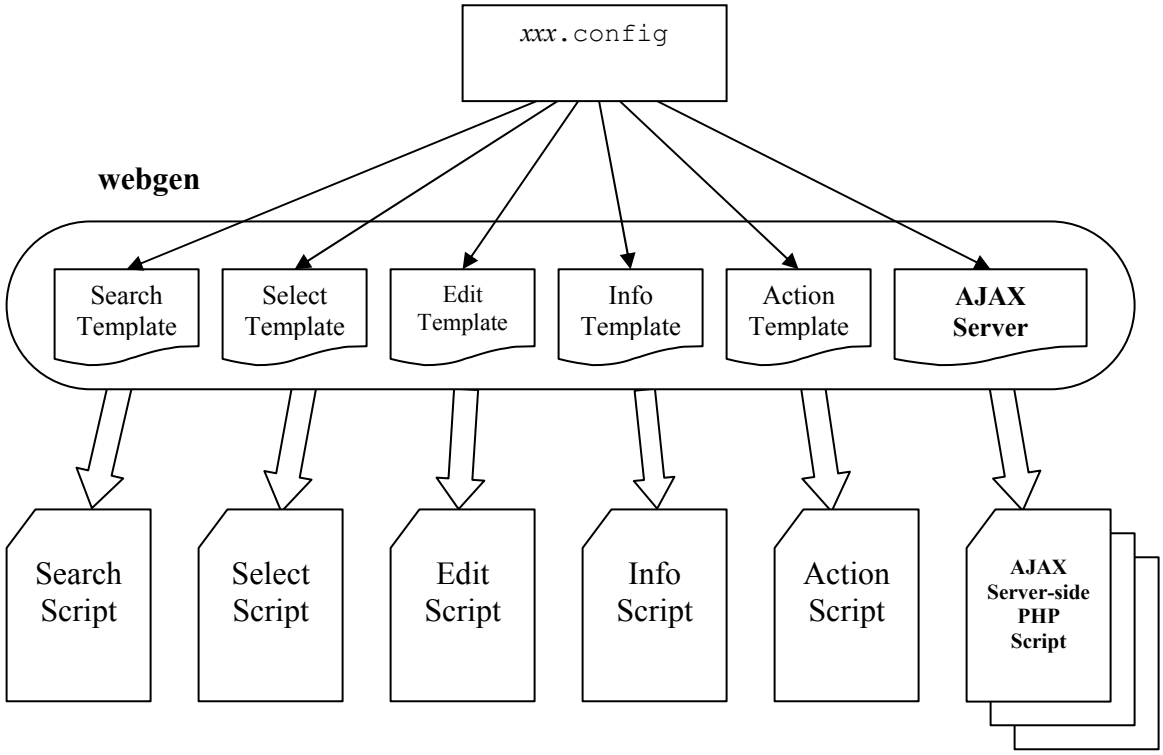

Figure 3.3: Generating Web-scripts by WebGen templates.

In addition to the scripts previously generated, WebGen can now support AJAX requests by parameterizing `url`, `target_element`, and `response_type` in `ajax_client.js`.

1. Parameter `url` indicates the URL consisting of the server-side PHP script and the HTML parameters.
2. Parameter `target_element` indicates the ID of the element where the response is stored.
3. Parameter `response_type` can be `value` or `options`, where `value` indicates a scalar value, and `options` indicates the options for an HTML `select` element.

Also, server-side script *xxx*`_ajax_`*yyy*`.php` is automatically generated, and for this purpose, `$ajax_fields` is added to configuration file *xxx*`.config`.

For each type of AJAX requests, one server-side AJAX script in PHP is needed. When UNIX command `webgen` is issued with table name *xxx*, AJAX server-side scripts in PHP as well as five form scripts are generated as shown in Figure 3.3.

3.1 AJAX Client JavaScript

In order to support an AJAX request, JavaScript file `ajax_client.js` need be included in a form script. Two functions `sendRequest()`, which is invoked when a value is selected from a dropdown list in a form, and `handleResponse()`, which is a callback function for a response produced by an AJAX request, are implemented in this file.

```
function sendRequest(url, target_element, response_type) {

    var http_request = false;
    if (window.XMLHttpRequest) { // Mozilla, Safari,...
      http_request = new XMLHttpRequest();
      if (http_request.overrideMimeType) {
        http_request.overrideMimeType('text/xml');
      }
      http_request.target_element = target_element;
      http_request.response_type = response_type;
    } else if (window.ActiveXObject) { // IE
      try {
        http_request = new ActiveXObject("Msxml2.XMLHTTP");
      } catch (e) {
        try {
          http_request = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (e) {}
      }

      global_target_element = target_element;
      global_response_type = response_type;
    }

    if (!http_request) {
      alert('Giving up :( Cannot create an XMLHTTP instance');
      return false;
    }

    http_request.onreadystatechange =
      function() { handleResponse(http_request); };
    http_request.open('GET', url, true);
    http_request.send(null);

}
```

Figure 3.4: AJAX JavaScript function `sendRequest()`.

`sendRequest(`*`url`*`, `*`target_element`*`, `*`response_type`*`)`

Arguments

`url`

URL of the server-side PHP script.

`target_element`

The ID of the HTML element where the response data is set.

`response_type`

Type of the HTML element for the response, `value` which indicates a scalar value or `options` which indicates a set of options for an HTML `select` element.

Returns

`false` if an instance of `XMLHttpRequest` or `ActiveXObject` is not created. This method returns nothing if it is created successfully.

Description

First, object `http_request` that handles AJAX requests and responses on the client-side is created. For IE5 and IE6, `http_request` is an instance of `ActiveXObject`, and for Mozilla, Firefox, Safari, and IE7, it is an instance of `XMLHttpRequest`. Custom properties `target_element` and `response_type` are added to this instance. Function `handleResponse()` is set in the property `onreadystatechange` as the callback function for a response. By `open()` function, `url` and the HTTP method, which is `GET`, are set. Finally, `http_request` is sent by `send()`.

```
function handleResponse(http_request) {
  if (http_request.readyState == 4) {
    if (http_request.status == 200) {
      if (http_request.response_type) {
        response_type  = http_request.response_type;
        target_element = http_request.target_element;
      } else {
        response_type  = global_response_type;
        target_element = global_target_element;
      }

      switch (response_type) {
        case "options":
          responses = http_request.responseText.split('|');
          select = document.getElementById(target_element);

          select.options.length = 0;
          select.options[0] =  new Option("", "", false, false);
          for (var i = 0; i < responses.length; i += 2){
            select.options[1 + i/2] =
              new Option(responses[i+1], responses[i],
                         false, false);
          }
          break;
        case "value":
          response = http_request.responseText;
          document.getElementById(target_element).value = response;
          break;
        default:
          break;
      }
    } else {
      alert('Response error code: ' + http_request.status);
    }
  }
}
```

Figure 3.5: AJAX JavaScript function `handleResponse()`.

`handleResponse(`*`http_request`*`)`

Arguments

`http_request`

An instance of `XMLHttpRequest`

Returns

Nothing.

Description

When the client-side script receives a response, this function is activated.

1. If `http_request.response_type` is `value`, then the returned value is set in the text box of the `input` element specified by `http_request.target_element`.
2. If `http_request.response_type` is `options`, then the response data is set in the dropdown list of the `select` element specified by `http_request.target_element`. The response data is a sequence of values separated by a character `|`. For example, the options of a dropdown list for a list of Oregon counties are encoded as,

   `001|Baker|003|Benton|005|Clackamas|007|Clatsop`.

   Each pair of values is set as one option of the `select` element.

3.2 AJAX Server-Side PHP Scripts and the Template for them

The server-side PHP script for each type of AJAX requests can be generated automatically by `webgen` from template script `ajax_server.tmpl`. If variable `$ajax_fields` is defined in *xxx*`.config` file, for each element in `$ajax_fields[]`, the template activated from `webgen` generates PHP script *xxx*`_ajax_`*yyy*`.php`, where *yyy* is the name of the table whose records are retrieved by an AJAX request. The following properties are defined for each element of `$ajax_fields[]`:

`source_column` (Required)

The foreign-key column in table *xxx*.

`sqlFrom` (Required)

The name of the table whose records are retrieved by an AJAX request. This name is also used as *yyy* in *xxx*`_ajax_`*yyy*`.php`.

`linked_column` (Optional)

The foreign-key column in the table whose records are retrieved. If this value is same as the value of `source_column`, this need not to be defined.

`sqlSelect` (Required)

Two columns in the table specified by `sqlFrom`. The values in these columns are used for the options of the `select` element.

`response_type` (Required)

The type of the target element, `options` or `value`. Type `options` indicates that an AJAX request returns a list of values to a dropdown list, and type `value` indicates that an AJAX request returns one value.

`whereAdd` (Optional)

An additional condition for the `where` clause of the SQL statement.

`orderBy` (Optional)

The column for sorting the retrieved records. This property is applied to the `order by` clause of the SQL statement.

Consider a form where a state and a county need be selected. With this form, when a state is selected, the list of the counties in the state is returned and displayed in a dropdown list. The relationship among the form and table `d_state` is shown in Figure 3.6. Table `d_state` contains information on the states, and table `d_county` information on the counties.

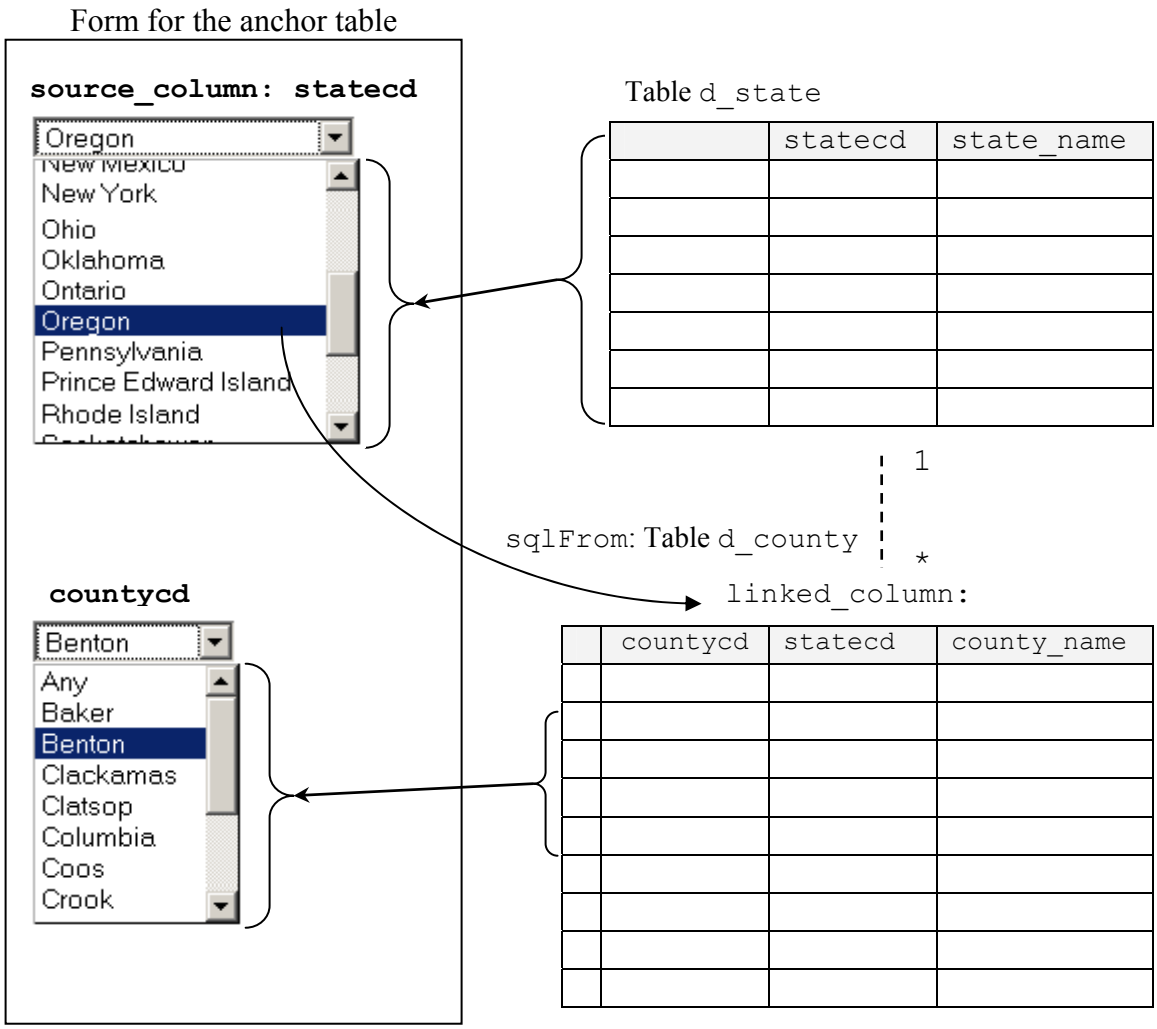

Figure 3.6: Relationship among the form and table `d_state` and table `d_county`.

For this purpose, `$ajax_fields` need to be defined in the configuration file as shown in Figure 3.7.

```
$ajax_fields = array(
  array(
    "source_column" => "statecd",
    "sqlFrom"       => "d_county",
    "linked_column" => "statecd",
    "sqlSelect"     => array("countycd", "county_name"),
    "response_type" => "options",
    "orderBy"       => "county_name",
  ),
);
```

Figure 3.7: `$ajax_fields` in *xxx*`.config`.

1. `source_column`, which is the foreign-key column in the anchor table for the form, is set to `statecd`. `statecd` is an alternate key in table `d_state`.
2. `sqlFrom` is table `d_county`, from which county records are retrieved.
3. `linked_column`, which is the foreign-key column in table `d_county`, is set to `statecd`. This column is linked to column `statecd` in table `d_state` and to column `statecd` in the anchor table.
4. `sqlSelect` is a pair of columns `countycd` and `county_name` in table `d_county`. The values of these columns are retrieved for the dropdown list of the counties in the state selected. The values in columns `countycd` and `county_name` are used by the options of the `select` element.
5. `response_type` is `options`, since multiple records are retrieved from table `d_county`.
6. `orderBy` is `county_name` so that the counties names retrieved are sorted according to their names.

When `webgen` is activated for table *xxx*, *xxx*`_ajax_d_county.php` shown in Figure 3.8 is generated from `$ajax_fields` defined in *xxx*`.config`. This script is used as the server-side PHP script for the *search* and *edit* forms for table *xxx*.

```
<?
  include("../datasource.php");
  include("../../../framework_v3/common.phtml");

  $statecd = get_param('statecd');

  $sql =
    "select countycd, county_name
     from   d_county ";

  if (!empty($statecd)) {
    $sql .= " where statecd = '$statecd'";
  }

    $sql .= " order by county_name";

   $db->query($sql);

   $select_options = array();
   $nrows = $db->num_rows();
   for ($i = 0; $i < $nrows; $i++) {
     $db->next_record();
     $select_options[] = $db->f('countycd');
     $select_options[] = $db->f('county_name');
   }
   $select_options_string = implode('|', $select_options);
   return $select_options_string;
?>
```

Figure 3.8: *xxx*`_ajax_d_county.php`.

Based on the definition of `$ajax_fields`, the following SQL statement is constructed:

```
SELECT countycd, county_name
       FROM d_county
 WHERE statecd = '$statecd'
    ORDER BY county_name
```

After the county records for the selected state are retrieved from table `d_county`, the values of `countycd` and `county_name` in each record are first stored in array `$select_options[]`. Then all the elements in `$select_options[]` are joined into `$select_options_string` where adjacent values are separated by character `|`. Finally, `$select_options_string` is returned.

When a value is selected from the dropdown list of the states, `sendRequest()` need be called by the `onChange` event. For this purpose, the `add_attribute` option need be defined in `$edit_fields` and `$search_fields` for the `statecd` field as

```
'add_attribute' => 'onChange =
   "sendRequest(\'./xxx_ajax_d_county.php?statecd=\'+this.value,
                \'countycd\',
                \'options\');
   return false;"'
```

As we discussed in Section 3.2, `sendRequest()` requires three parameters: `url`, `target_element`, and `response_type`.

1. `url` designates the server-side AJAX script generated by `ajax_server.tmpl` and the HTML parameter for the `statecd` field.
2. `target_element` indicates the name of the `select` element in which the list of the counties selected for the state specified are displayed.
3. `respose_type` is `options` since a list of counties for the options of the `select` element is returned from the sever-side PHP script.

With the `add_attribute` option defined in `$edit_fields`, in the *edit* script, the `onChange` attribute is added to the `select` element for the `statecd` field as shown in Figure 3.9.

```
<select name=statecd ID=statecd onChange="sendRequest(
  './clu_parcel_ajax_d_county.php?statecd=' + this.value,
  'countycd',
  'options');
 return false;">
    <option value="" selected> Any</option>
    <option value=00> Aguascalientes </option>
    <option value=01> Alabama </option>
    <option value=02> Alaska </option>
    <option value=03> Alberta </option>
    <option value=04> Arizona </option>
</select>
```

Figure 3.9: Code of the `select` element of a state

4. Editable and Insertable Select Form

Using an *edit* form generated by `webgen`, a user can perform an insert, delete, and update action for a single record, but she cannot manipulate multiple records at a time. With an ordinary *select* form, a user can view a list of records and delete records selected from that list. For inserting a new record or updating an existing record, an *edit* form need be open from the *select* form as shown in Figure 4.1.

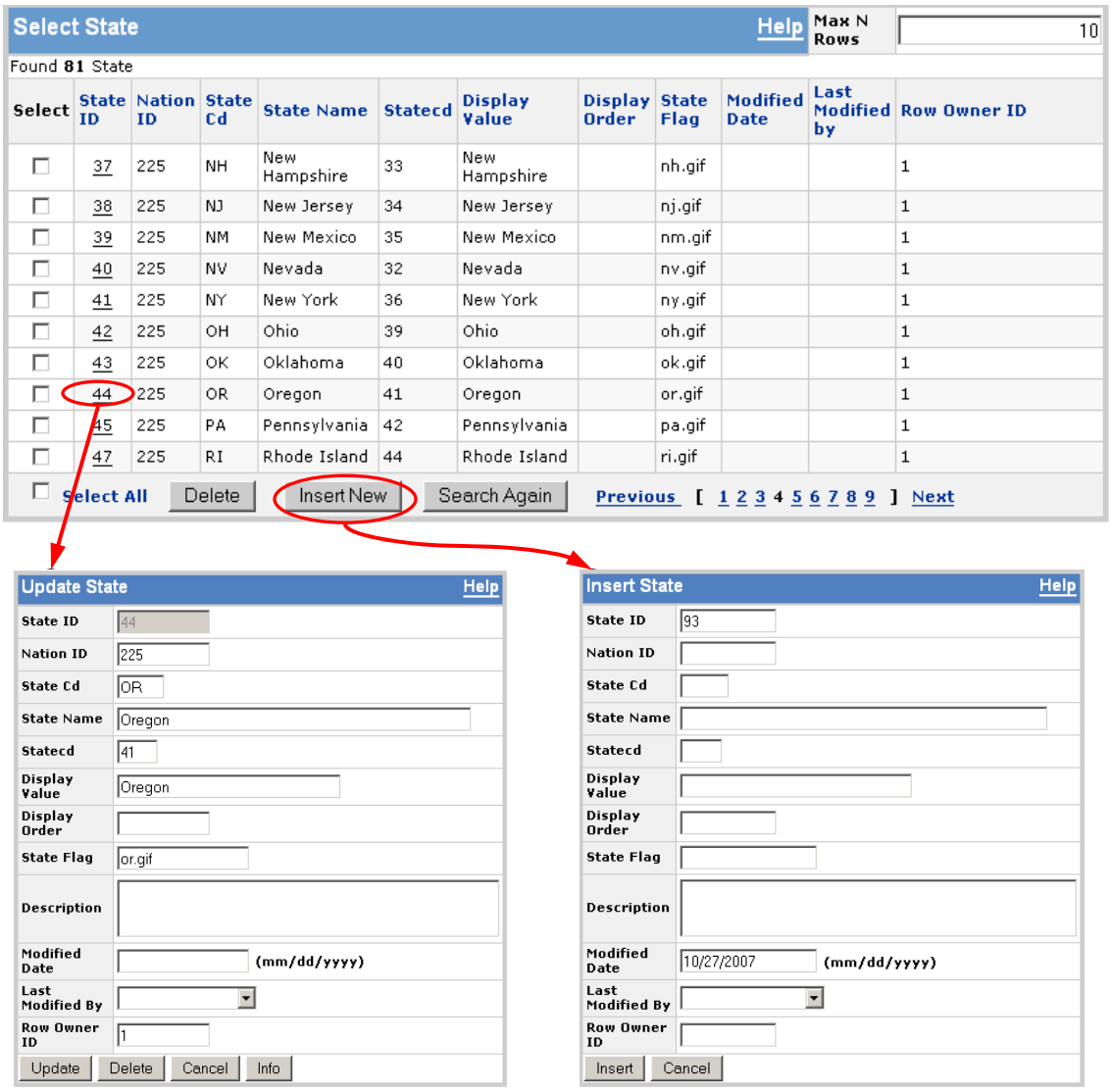

Figure 4.1: Updating and inserting a record from an ordinary *select* form.

However, with an *editable* and *insertable select* form, a user can insert new records and *update* and *delete* existing records. If the *select* form is editable, each form

cell becomes an `input` or `select` element as shown in Figure 4.2 so that a user can modify the value of the element. Furthermore, if the *select* form is insertable in addition to being editable, new rows can be added to the *select* form in order to insert new records.

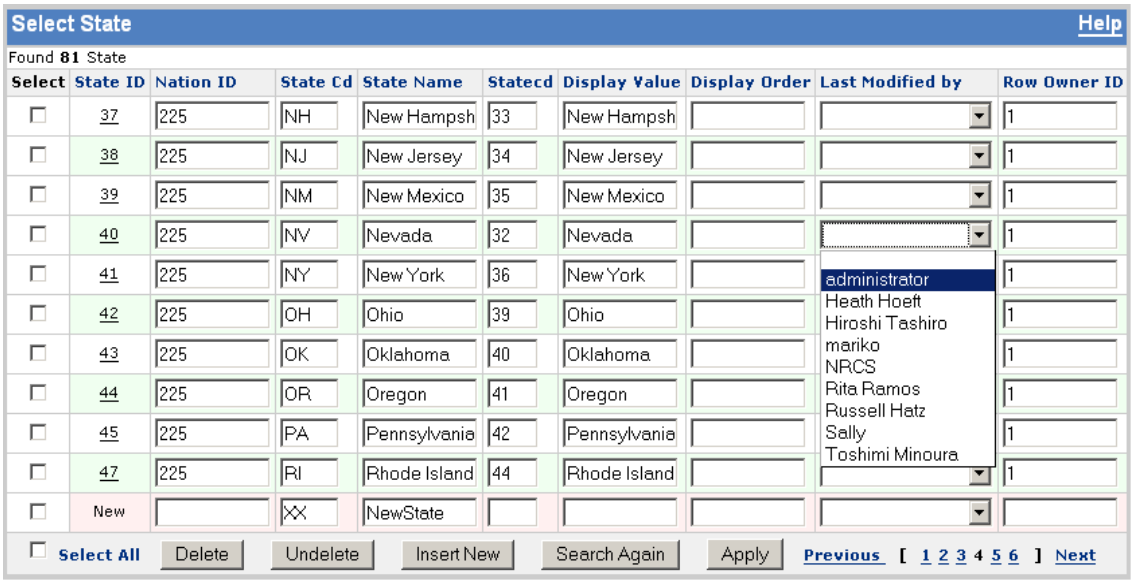

Figure 4.2: An *editable* and *insertable select* form.

If a user wants to update an existing record, she can modify the values in the `input` and `select` elements. If she wants to insert a new record, then she can click the *Insert New* button and provide new values in the `input` and `select` elements. In order to delete existing records, she can select those records and click the *Delete* button. Delete requests can be cancelled with the *Undelete* button. The actual operations on the records in the database are performed when the *Apply* button is clicked.

Two form elements are provided for each field of a record in the *select* form. One is a hidden `input` element that keeps the value retrieved from the database when the *select* form is loaded. The other is an `input` or `select` element maintaining the value that can be updated. The values in these two elements are initially identical. When a

new row is added for inserting a new record, only the form elements for the new values are provided for the row.

In addition to these form elements, form element `subcmd`, which specifies the type of the action applied to each record, is provided:

`'I'` for a record to be inserted,

`'D'` for a record to be deleted, and

`'N'` for the remaining records.

When the *Apply* button is clicked, the form parameters are submitted to the script. Then, an SQL query is constructed based on the value of form parameter `subcmd` provided for each row. If it is `'I'` or `'D'`, an insert or delete SQL query is formulated and executed, respectively. If it is `'N'`, the old and new values of each field of the record are compared. If the old and new values of any field are different, an SQL query for updating the record is constructed and executed.

If variable `$select_editable` is set to `true` in the configuration file of a table, the *select* form of the table can become editable and insertable. The *select* form becomes editable if form parameter `editable=1` is passed to the *select* script and, it becomes insertable if form parameters `editable=1` and `insertable=1` are passed to it.

5. Conclusions and Future Work

We added mechanisms for access control, AJAX support, and *editable*-and-*insertable* table to WebGen. Five access control levels were implemented, and one of them can be specified for each table. With this access control mechanism, we can protect records owned by a user from other non-`admin` users. We provided an AJAX support for the value of an `input` element or the list of the options of a `select` element which is dependent on the value of another element. The AJAX server-side scripts for this purpose can be generated automatically. We extended the template for *select* scripts to support *editable*-and-*insertable select* forms. An *editable*-and-*insertable* form allows a user to insert and update multiple records as well as to view and delete them without opening an *edit* form for each record.

The following features can be added to improve WebGen further.

1. The access control mechanism can be improved if groups of users are introduced.
2. As discussed in Section 2.4, with an *edit* form, an `owner` user can modify another record owned by her by providing the primary key value of the record in the URL. We should prevent her from updating a record in this way.
3. With our AJAX support mechanism, we can use only a single `input` or `select` element as the target element. Sometimes we need to allow multiple target elements.

6. Reference

[ALBA-05] Albader, B. A Configuration File Generator for Web Forms, M.S. project report, School of EECS, Oregon State Univ., 2005.

[CHAL-08] Chalainanont, N., Sano, J. and Minoura, T. Automatic Generation of Web-Based GIS/Database Applications. Open Source Geospatial (OSGeo) Journal, To appear, 2008.

[EUM-03] Eum, D. and Minoura, T. Web-Based Database Application Generator. *IEICE Transactions on Information and Systems, Vol. E86-D*, No. 6. June 2003.

[NAIK-02] Naik, P. WebSiteGen3: A Tool for Generating ASP.NET Forms, M.S. project report, School of EECS, Oregon State Univ., 2002.

[TASH-06] Tashiro, H. WebGen 5 Version 2: Automatic Web-Script Generator for Web-Based GIS/Database Applications, M.S. project report, School of EECS, Oregon State Univ., Sept. 2006.

[WANG-03] Wangmutitakul, P., Li, L., and Minoura, T. User Participatory Web-Based GIS/Database Application. In. *Proc. of Geotec Event Conference*, March 2003.

[WANG-04] Wangmutitakul, Paphun, et al. WebGD: Framework for Web-based GIS/database Applications, *Journal of Object Technology 3*, 4, 209-225, 2004.

[YANG-04] Yang, S. WebGen 5: A PHP Script Generator with Templates, M.S. project report, School of EECS, Oregon State Univ., Sept. 2004.