The friendly web framework

Programmers manual

Daniel Böhringer

Contents

1. Introduction

Desktop databases are perfect for rapid GUI prototyping. A major advantage is that database rows are synced with the GUI automatically. However, most desktop database systems perform poorly under heavy load. Others are not suitable for programming web applications at all.

<dbweb> is an Apache2-based web framework that provides rapid prototyping as well as deployment under heavy load. You can write a fully CRUD[1]-functional web application with complex master detail semantics in (purely declarative) HTML-like syntax! Additionally, <dbweb> introduces specialized input elements such as ComboBox or DataGrid. These allow for handling very large data-sets as they fetch the data on demand and do not require data pagination.

All application code (e.g. invoked from buttons, context menus) runs server-side[2]. <dbweb> applications are nevertheless fully interactive and provide confirm-dialogs or GUI-animation effects for immediate feedback. This is possible through Ajax technology. The Ajax-layer is hidden to the application developer by default. An object oriented API provides access to the database through a thin object-relational mapper.

2. Overview

In <dbweb>, the GUI is written in HTML/CSS. <dbweb> introduces only a few additional tags (the most common are `<foreach:..>...  </foreach>`, `<form:..>...  </form>` and `<var:..>`). These tags always refer to *DisplayGroup*s which connect the GUI to your databases. We discuss DisplayGroups in chapter 5.

Briefly, a DisplayGroup basically represents a single table/view in the database but additionally keeps track of row selection (a single row is always selected). The selection can be changed interactively or programmatically. DisplayGroups can be connected to other DisplayGroups in master-detail configurations [3]. Commonly, a DisplayGroup feeds the data row-wise into a HTML table or a more specialized DataGrid view.

Data is edited within tables or through dedicated forms. By default, all edits are immediately committed when the input element loses focus. All changes are reflected throughout the GUI without page reloading. DisplayGroups can also be enriched with context menus. Context-menus commonly trigger specific application code (chapter 7).

3. Quickstart example

We herein demonstrate how to write a simple data browser for a minimalistic two-table relational database. This database stores author-names together with their book titles (the SQL-schema is in listing 3). A screenshot of our finished application is in figure 1.

[1] Acronym for the basic operations operations create, update and delete

[2] Currently, only Perl is supported, future releases may also support (server-side) Javascript

[3] `http://en.wikipedia.org/wiki/Master-detail`

The user can change the selection of the authors table by left-clicking. The appropriate book titles automatically show up next-by. Changes are immediately committed upon editing the form. The GUI elements update without page reloading.

Figure 1: Screenshot of our data browser for the authors/books database 1. The first author is selected (dotted border). The second column lists only the books of that author. A particular book is selected and can be edited in the form to the right.

The <dbweb> code for figure 1 is in listing 1. The data tables and the form are laid out column-wise (lines 1-4). Lines 5-16 comprise the author-table. This table is generated through the concept of *foreach repetition* (discussed in detail in section 4.1). Briefly, a template (lines 11-14) is concatenated for each data-row in the DisplayGroup `authors` (line 10). The two `var:...` tags are substituted by the appropriate content in each row. The same concept applies to the book-title table (lines 19-28). This table automatically displays only the appropriate rows, depending on the selection of the first table. The rightmost column comprises of a form (lines 31-33). This form allows for editing the current book title (line 33). See chapter 4.3 for a full discussion of forms.

But how does <dbweb> know which are the appropriate book-records are to be listed in the books-table? Where is the information the database connection? These informations are bundled in the DisplayGroups, that also keep track of the selected rows. Listing 4 gives the DisplayGroup code for our example. This code is commonly auto-generated from the SQL-schema, so you should not worry about the complexity at this stage.

The <dbweb>-Apache2-module transparently transforms the 37 lines from listing 1 into plain HLML and Javascript that works with all current web-browsers. All tables are click-responsive and the detail form is automatically connected to the master table. This is interesting, because we did not write any code for event processing or database access. Obviously, <dbweb> factors away all tedious event processing code. A similar AJAX application written e.g. directly in Javascript and a server-side scripting language such as perl/PHP would span at least 1000 lines of code even for our simple example scenario. The complexity comes from cross-browser compatibility and the edge cases such as changing selection during editing.

Listing 1: Complete source code of our authors and book example. <dbweb>-keywords are in boldface.

```
1 <html>
2 <table>
3 <tr>
```

```
<td valign=top>
  <table class="datatable">
    <tr>
    <th>id</th>
    <th>author</th>
    </tr>
    <foreach:authors>
    <tr>
      <td><var:id></td>
      <td><var:name></td>
    </tr>
  </foreach>
  </table>
</td>
<td valign=top>
  <table class="datatable">
    <tr>
    <th>booktitle</th>
    </tr>
    <foreach:books>
    <tr>
      <td><var:title></td>
    </tr>
    </foreach>
  </table>
</td>
<td valign=top>
<form:books format:label="Edit book title">
  <var:title edittype=text>
</form>
</td>
</table>
</body>
</html>
```

4. GUI programming

4.1. Foreach repetition

The most basic way to expose database content tabularly is *foreach repetition*. E.g. the template of a HTML table row is concatenated for each data-row in the DisplayGroup from the enclosing `foreach`. Chapter 3 gives a basic example.
The generalized syntax is: `<foreach:`*DG[.filterName][options]*`>` ... `</foreach>` where *DG* is the name of any DisplayGroup, *options* an optional set of parameters (table 1). The enclosed template block is commonly the template for a single table row. Herein, database values can be exposed through `<var:..>` tags (see next section). The filterName option is discussed in chapter 5.3

Table 1: Parameters of the `foreach` tag.

Parameter	Description
plain=YES	Performs template processing only without making the rendered table interactive. This option is only useful in very advanced setups. With this option set, automatic user interaction and standard master-detail set-ups will not work as advertised.
classnameVar=*col*	This option will assign a CSS classname by textually concatenating `class`, *col* and the acutal value of *col* for the current row. You can use this feature to automatically highlight rows on the basis of a certain value in the current row. You can define the highlighting style in your CSS (see chapter 6 for where to placing the CSS).
perl=*perlfuncName*	*perlfuncName* is invoked in each iteration and can modify the template processing.

Apart from rendering <dbweb> tables to pure HTML, *foreach repetition* additionally makes the resulting HTML table interactive by default: the user can change the selected row by clicking on the table. <dbweb> ensures that the elements of the selected row are automatically assigned to the CSS class `selectedRow` in the web browser. The default <dbweb> stylesheet defines a slightly darker background colour and a dotted border. The visual results are best when the table is assigned the CSS-class `datatable`.

4.2. Exposing database content

Database content is exposed through `<var:`*column name*`>` tags. These tags are only meaningful within `<foreach:>` or `<form:...>` environments that connect the column names to the appropriate database tables through a DisplayGroup. The generalized syntax of the `<var:>` tag is: `<var:`*column name* [*options*]> where *column name* is the name of a valid column in the DisplayGroup referred to by the surrounding `<foreach>` or `<form>` environments. *options* an optional set of parameters discussed in table 2.

4.3. Editing Data through forms

Forms group related data for editing. The form data is automatically bound to — and kept in sync with the selected row of the form's DisplayGroup. The selected row can e.g. be changed by clicking at a table that is bound to the same DisplayGroup. Listing 1 and figure 1 give a basic example of a simple form that also demonstrates some of the

Table 2: Common parameters for the `var:` tag. Additional options are discussed in table 3 together with the respective edittypes.

Parameter	Description
edittype=*aType*	This option makes the data editable and also determines the input element. *aType* is one of `text`, `password`, `plain`, `combo`, `popup`, `boolean`, `textarea`, `button` and `upload`. See table 3 for a detailed discussion of these edittypes.
class="*aCSSclass*"	Adds *aCSSclass* to the input element int the DOM.
style="*aStyle*"	The CSS-style of the input element is set to *aStyle*.
id="*aDOMid*"	Assigns the *aDOMid* to the input element. You can refer to the input element in your custom JavaScript. You are responsible to ensure that *aDOMid* is unique in the DOM.
autocomplete="off"	Turns off the autocompletion feature built into the modern browsers.
format:label="*aLabel*"	*aLabel* is translated to a HTML label tag assigned to the resulting input element.
format:date="*format-string*"	Translates calendarial database content to UI representation and vice versa through a POSIX date format string. E.g. in Germany, dates are commonly formatted `%d.%m.%Y` whereas the format-string of the ISO standard is `%Y-%m-%d`. Please refer to the `date` manpage of your web server system for all date-component tokens that are available on your system.
format:typeahead= "*DG.col*"	Pulls down a pick-list below the input control upon typing just like the autocomplete feature of modern web-browsers. However, this option limits the autocompletion list to the data from column *col* of DisplayGroup *DG*.
format:lookup= "*DG.col*"	Substitutes the value of the column specified by <var:...> (current column) with the value of *col* in DisplayGroup *DG*. The value is taken from the row in DG whose primaryKey equals the value of the current column value (just like in SQL joins).

Table 3: Discussion of all edittypes for the `var:` tag.

Parameter	Description
text, password	Single line text input (masked for password). This width of the input field can be modified with the option `size="`*countColNums*`"` with *countColNums* a numeric value that has to be optimized empirically.
textarea	Multiline text input. Width and height of the input area are specified through the `width="`*countColNums*`"` and `height="`*countRowNums*`"` options, respectively.
plain	Substituted by the data without any transformations.
popup, combo	Edit a foreign key on the basis of a (more user friendly) surrogate column. Popup provides a *select* input element, combo renders to a single line text input element with an Ajax-pick-list attached below. Use popup if the list is rather small. Use combo for larger lists as the options are constantly narrowed during typing. Large pick-lists may slow down the browser significantly after focussing the combo element. Here, `format:pulldown="off"` can prevent the complete pick-list to pull down initially. The *surrogate column* from DisplayGroup *DG* is determined through `format:data="`*DG.surrogateColumn[.filterName]*`"` for both popup and combo. The primaryKey of *DG* rather than the surrogate column is bound to the associated `var` tag. The option `format:notNull="YES"` ensures that the first entry from the data-source is preselected (instead of the NULL value) and that no empty (NULL) option is available in the popup/pick list.
boolean	Checkbox for boolean data. Note that NULL and FALSE cannot be distinguished by the end-user.
button	Button input element. Server-side code can be attached through `perlfunc="`*aPerlFuncName*`"`. See chapter xx for where to put the server-side code.
upload	Provides an Ajax-powered upload button to directly place the data from client-side files into the database.

`var` tags discussed in the previous section. The full syntax is: `<form:`*DG* `[`*options*`]> ...</form>` where *DG* is the name of any DisplayGroup and *options* an optional set of parameters (table 4). The enclosed part typically comprises of `<var:..>` tags with the edittype attribute set appropriately (see previous section).

Table 4: Parameters of the `foreach` tag.

Parameter	Description
label="someLabel"	Adds someLabel as heading above this form.
style="someCSS"	Adds someCSS the the CSS style of this form.
id="someID"	Assigns DOM-id *someID* to this form.
plain=YES	Transforms to pure HTML form without <dbweb> mechanics.
perl=*perlfuncName*	*perlfuncName* is invoked upon data editing for validation. See section xx.

4.4. DataGrid Tables

The major drawback of *foreach repetition* is that the page may get very large when the DisplayGroup holds more than 50 rows. This would hamper usability and speed in the Web browser. DataGrid Tables provide a scroll bar to a fixed height table and fetches only the currently visible subset of rows from the database. This allows for displaying arbitrary long tables without pagination. The generalized syntax is in listing 2.

Listing 2: Syntax-block for DataGrids (see chapter 5.2 for a discussion of the optional filterName attribute)

```
<table:someDG[.filterName] rows="CountOfVisibleRows">
  <head>
    <col:WidthOfColumn> SomeHeaderName </col>
    <col:WidthOfOhterColumn> OtherHeaderName </col>
  </head>
  <foreach>
    <cell><var:columnName></cell>
    <cell><var:otherColumnName></cell>
  </foreach>
</table:someDG>
```

where *columnName* and *otherColumnName* are valid columns in DisplayGroup *someDG*. The visible height of the table is set through the integer *CountOfVisibleRows*. The table appearance is identical to foreach repetition If the row count is lower or equal to this value. Otherwise, a native scrollbar is attached to the right of the table. The column with is fixed. Oversized cells are clipped. The column.width is pre-set in the header section through *WidthOfColumn* and *WidthOfOhterColumn*. These values have to specified

as integer postfixed by unit, such as `110px` for 110 pixels width in the browser layout.

4.5. Conditions in the User Interface

Parts of the UI can be conditionally hidden away on the basis of database-values. Typical uses for conditional parts of the UI is to hiding elements that does not make sense in a particular context. Another usage is to implementing tabs that structure large forms into smaller aspects of semantically related fields.

In <dbweb>, conditional parts of the UI are placed within the `<cond var:...>` and `<condDG:...>` environments. The former construct is valid only within a `<foreach:...>` or `<form:...>` environment that provides the link to a DisplayGroup. Here, the formal syntax is: `<cond var:`*columnName* =*condition*> ... `</cond>`
where *column name* is the name of a valid column in the DisplayGroup referred to by the surrounding `<foreach:...>` or `<form:...>` environments. *condition* is discussed in table 5.

`<condDG:`, the other incarnation of conditionals can be placed anywhere as the DisplayGroup is specified with the tag. The syntax is:
`<condDG:`*aDG* `var:`*columnName* =*condition*> ... `</condDG:`*aDG*> with aDG the DisplayGroup whose columnName of the selected row is queried against the condition.

Table 5: Syntax for condition statements. The enclosed block is deleted during rendering unless the condition evaluates true. The selection and count keywords are valid only for `<condDG:...` environments.

var:columnName=	True if data...
const:*someLiteral*	...equals to *someLiteral*
eq:const:*someLiteral*	...equals to *someLiteral*
ne:const:*someLiteral*	...is not equal to *someLiteral*
gt:const:*someLiteral*	...is greater than *someLiteral*
lt:const:*someLiteral*	...is less than *someLiteral*
eqnull	...is a NULL value (in database sense)
nenull	...is not a NULL value (in database sense)
selection=	**True if selection...**
true	...is true in boolean context
false	...is false in boolean context
visible	...is visible
invisible	...is invisible
empty	...is empty
count= gt:const:*someNumber*	True if the DisplayGroup holds more rows than *someNumber*.

5. Database interface

5.1. What is a DisplayGroup?

DisplayGroups encapsulate database tables/ views for <dbweb>. A DisplayGroup represents a single database table/view one-by-one. However, it additionally keeps track of a selected row. The first row of the DisplayGroup (in the database order, unless you supply ordering specifications, see section **??**) becomes selected during initialization. The selection can be changed interactively by the user through clicking at any row, or programmatically through the API.

By default, DisplayGroups hold the full dataset of the underlying database table. These DisplayGroups are called *master* DisplayGroups and need a connection-property to make the link to the database (see table 6). DisplayGroups can also be connected to other DisplayGroups by means of the `bindToDG` property. These connected DisplayGroups inherit the database connection from their master DisplayGroup. They can additionally be configured to become a *detail* DisplayGroup through providing the `bindFromColumn` property. *Detail* DisplayGroups hold only the subset of all rows with the foreign key from the bindFromColumn property equalling the primary key of the selected row in their master DisplayGroup. In our author-book example application (figure 1) the authors DisplayGroup is the master DisplayGroup and the books DisplayGroup is configured as a detail DisplayGroup.

DisplayGroups are declared within the `<DisplayGroup` *[options]*... `</DisplayGroup>` environment. *[options]* specifies the syntax of the enclosing display group definitions. Multiple <DisplayGroup> environments can be declared within a single file or included from library files, see section 6. <dbweb> supports JSON and classical Apple PropertyList syntax. The latter is the default, JSON has to be marked out by `format="JSON"`. All examples in this document are in Apple PropertyList syntax as this markup language is much easier to read.

5.1.1. DisplayGroups for database tables

Our minimalistic relational database from the introduction example is shown in SQL syntax in listing 3. The corresponding DisplayGroup definition is given in listing 4. Table 7 and 6 list all properties of a DisplayGroup.

Listing 3: SQL schema of a minimalistic relational example database. (PostgreSQL syntax)

```
CREATE TABLE authors (
    id    serial PRIMARY KEY,
    name  varchar
);

CREATE TABLE books (
    id          serial PRIMARY KEY,
    author_id   int4 references authors(id),
```

Table 6: Basic DisplayGroup properties. You may want to use ModelMaker.pl to generate the markup directly from SQL-syntax.

Property	Explanation
table	Name of the database table.
columns	List of all column names that are made accessible from the table.
primaryKey	Name of the primary key of table. **Note:** Only primaryKeys with values exclusively made of digits, a-z (case insensitive), whitespace, dots and underscore characters are currently supported by <dbweb>.
types	Key-Value dictionary specifying the database types (values) of all non-text columns (keys). Valid types are bool, int (also valid for floats) and date (also valid for timestamps).
write_table	Optional name of a database table for writing, e.g. when table points to a read-only view. Newer versions of PostgreSQL require the type specification. May be optional in other settings.
suppress_insert	Optional list of columns that are not valid within write_table. Otherwise database errors will occur upon inserts and updates.
connection	Dbi-DSN specifying the database connection. When omitted, the DisplayGroup chain (see boundToDG property) is iterated for the connection.
connectionEnv	Name of an environment variable holding the connection string (useful when switching from a test- to a production without modifying the source code).
connectionEnvAuto	When set to YES, the connectionstring is taken from the environment variable with name `dbweb_connection_string_`*nameOfYourApp*.
user, password	Database credentials for the connection (optional, see chapter **??**).
encoding	Encoding of the database table (defaults to latin1).

Table 7: Advanced DisplayGroup properties.

Property	Explanation
boundToDG	Name of the parent DisplayGroup
bindFromKey	Name of the foreign key column in this DisplayGroup
targetColumn	Name of the binding key in the master DisplayGroup. Defaults to the primaryKey in the master DG if omitted.
data	See section 5.1.2
cache	Copies *all* rows from table into the session. Subsequent reads come from the cache that is synced with updates from <dbweb>. Useful for working with long-loading complex views. Do not use with long-tables as this would slow down the current session. See also section 5.1.4
DataInSession	Hold global variables. See section 5.1.3.
contextmenu	Define a context menu. See section 5.4.
autoSort, sortColumns	Specify sorting of the data. See section 5.2.
filters	Specify filters that can be accessed from the UI layer. See section 5.3.

```
    title       varchar
);
```

Listing 4: DisplayGroup markup of the SQL from listing 3. Please refer to table xx for a discussion on secure ways to provide the connection string and database credentials.

```
  authors = {
    table="authors";
    columns = ( id, name );
    types = { id = int; };
    primaryKey = id;
    connection="dbi:Pg:dbname=test;user=root;host=localhost";
  };
  books = {
    table="books";
    columns = ( id, author_id, title );
    types = {id = int; author_id = int; };
    primaryKey = id;
    bindToDG="authors"; bindFromColumn="author_id";
  };
```

5.1.2. DisplayGroups for static data

Data can also directly placed in the DisplayGroup. This is useful for feeding Tab-controls. See Listing 5.

Listing 5: The data is statically attached in this basic example

```
<DisplayGroups>
{ myDG = {
    columns = ( id, name );
    types = { id = int; };
    primaryKey = id;
    data = ((1,"Pictures"),   (2, "Pictures Long"),
            (3, "Features" ), (4, "Relatives" ) );
  };
}
</DisplayGroups>
```

5.1.3. DisplayGroups for global variables

Global variables of the current session is best kept in a DisplayGroup with the DataInSession=YES attribute. This DG has only a single row that is always selected. The data is stored in the current session.

5.1.4. DisplayGroups for in-memory data

Data not from databases is handled in display groups with the Cache=YES attribute and no table attribute. The data has to be pumped in through the API and is stored in the current session.

5.1.5. DisplayGroups for Cookies/ HTML5-storage

This is not yet implemented but may be supported in future releases if there is user-demand arises.

5.2. Sorting

By default, dbweb does not sort data but preserves the order returned from the database. This conservative behaviour can be overridden through the `autoSort` and `sortColumns` properties of a DisplayGroup. The sortColumns property specifies a hash of column names as illustrated in listing 6. You can depose further sorting policies under different names but retrieving them is currently unsupported.

Listing 6: Syntax for automatic sorting. This DisplayGroup will contain that are primarly sorted for Column1 and secondarily by Column2

```
someDG={
```

```
  ...
  autoSort="mySortingSpec";
  sortColumns=
  { mySortingSpec =("Column1", "Column1");
    otherSortingSpec =("Column2", "Column1");
  };
};
```

5.3. Filtering

Filters specify different subsets of the full DisplayGroup dataset. You can select filters in the `<foreach>` (chapter 4.1) or `<table>` (chapter 4.4) environments, and also in datasource attributes of popup/combobox GUI elements (table 3). Filters are static by default. You can only specify constant attributes. However, you can modify filters at runtime, e.g. in the _earlyauto_ event handler. Here, you can e.g. overwrite the constant val attribute of a certain filter-element (listing xx). This is e.g. useful for only displaying the records, that the current user is allowed to see. The property name for filters is `filters`. The basic syntax for filters is very similar to sorting (chapter 5.2). The filters hash holds condition arrays. A single condition has this form: `{col="someCol"; op="someOperator"; val="someValue"}`. The condition is only true if all condition elements are true (logical AND). someOperator can be eq, ne, eqnull, nenull or literal. The val property can be omitted for eqnull and nenull, that test for NULL or non-NULL (in database speaking), respectively. A valid SQL fragment is required in the val property for the literal operator, a constant value otherwise.

5.4. Context menus

Context menus provide specific actions that may be tailored the the data in a specific DisplayGroup. For this reason, each DisplayGroup can have its own context menu. The context menu opens upon right clicking any row that belongs to this DisplayGroup. The context menu is configured through the `contextmenu` property of a DisplayGroup. Like a filter, this hash contains arrays of entry elements. A single entry element has this form:

Listing 7: Context menus are defined in the DisplayGroup. You can add separators or invoke perl code or perform simple DOM manipulations such as unhiding a GUI element

```
someDG={
  ...
  contextmenu=
  ( { name="Selektieren";
      action="select";
    }, { separator="true"; },
    { name="Bestaetigen";
      raiseDOMId="dom_opdat";
    },
```

```
    { name="OP geht nicht";
      perlfunc="enter_blockdate";
    }
  );
};
```

6. Application partitioning and navigation

Web applications are commonly split into (full-screen) sub-applications that cover specific sub-domains of the application. E.g. one page could cover customer administration whereas an other page could handle user-admin issues. Commonly, a fixed navigation element visually groups these sub-application to a single 'application-cluster'. <dbweb>-application-clusters usually reside in a cluster-specific subdirectory below dbwebressources (as defined in your httpd.conf, Listing 10, lines 3 and 4).

When an URL requests a path below your dbweb URL-location (in your httpd.conf, see Listing 10, line 5) such as `http://localhost/dbweb/myapp`, <dbweb> automatically jumps to the page called login.dgwapp inside of directory `myapp`. This file has to always exist. This behavior can be overriden through a t=targetApp in the URL. Simple authentication-free applications can be written within login.dgwapp. However, this page typically redirects to the main application page of the application cluster upon sucessful authentication through the API. Each dgwapp can have it's own CSS with the same name but css extension instead of dgwapp. A common CSS is automatically read from style.css below your dbwebressources.

6.1. Jumping between dgwapps

This is accoplished through <link:someTarget> tags.

6.2. Navigation tabs

You should include a common header that provides...

7. Application programming

Commonly, applications comprise of functionality beyond the basic CRUD operations and database navigation. E.g. reporting tools may download a PDF or an email should be sent when a button is pressed. Such application specific code is is organized in functional units named perlfuncs for historical reasons. A perlfunc can be semantically regarded a subroutine containing arbitrary code. This code is executed by the same version of perl as in the the mod_perl2 of your web server. Perlfuncs reside inside the dgwapp inside the `<perlfunc..>` environment. The syntax definition is given in Listing 8.

Listing 8: Syntac for perlfunc definition. SomeFile is commonly a perl library relative to dbwebressources.

```
<perlfunc name=someName" include="SomeFile">
{ ....
}</perlfunc>
```

Perlfuncs may be hooked to buttons and context menus through the `perl=` and `action=` keywords, respectively. Perlfuncs can also be automatically invoked at various places within the event loop or upon data updates or inserts. The code is executed inside a specialized environment, with all a perl-object instantiated for each DisplayGroup below the `$DG::` namespace. See section 7.2 for a discussion of these objects' methods and other functions to manipulate the application state, UI and database.

7.1. Refactoring

A central element for factoring out your common code-components is the include statement. The syntax for including is:
`<include src="path/local/to/dbwebressources.dgw">`
the extension can be chosen freely, dgw is only a convention. <html> and <DisplayGroup> environments are merged into the current application as well as all <perlfuncs>. The code is concatenated in case a perlfunc of the same names is already defined in the current application. This is useful e.g. for refacturing authentification checks out of application-specific _auto_ or _earlyauto_ handlers.

7.2. API

Explain basic stuff like DG namespace, undef and NULL, encoding issues

8. Security issues

Discuss PK filtering here, explain, why client side edit requests are limited to rows that were selected serverside. Explain why secured against SQL injections. Warn about context menues, cache issues... Other worthy points include where to specify database passwords (we support both user-supplied as well as static ones).

8.1. Automatically executed perlfuncs

explain the _onload_ _bootstrap_ _earlyauto_ and _auto_ hooks. Also explain the perl="xx" options from <form> and <foreach>.

Table 8: DisplayGroup methods for basic database operations.

$DG::someDG->...	Explanation
insertDict($aDictRef);	Inserts $aDict into the database table referenced by someDG. Returns the value of the primaryKey for the newly inserted row. Keys of $aDict not corresponding to valid column names are ignored.
insertDictUsingFilter($aDictRef, $aFilterName);	Same as insertDict, but extends $aDictRef with the key-value properties from a filter named $aFilterName in someDG.
updatePKWithDict($aPK, $aDictRef, $anOptionsDictRef);	Updates the database row referenced by the primary key $aPK with $aDict. Only keys that correspond to valid column names are updated.
updateSelectionWithDict($aDictRef, $anOoptionsDictRef);	Simply calls updatePKWithDict(...) with the primary key of the current selection of someDG.
deletePK($aPK);	Deletes the database row referenced by $aPK from the database table referenced by someDG.
deleteForWhereClauseDictRaw($aDictRef);	Low level method for deleting all rows whose column-values equal the key-value properties of $aDictRef. Ignores master-detail configurations and works on the full table.

Table 9: DisplayGroup methods for working with selections.

$DG::someDG->...	Explanation
selectedPK();	Returns the primary key of the selected row in someDG.
selectedDict();	Returns the the data of the selected row in someDG as (unblessed) hash reference. The keys of this hash correspond to the column names.
invalidateSelection();	Invalidates the current selection. As a result, the first row in someDG is selected upon the next page refresh.
makeSelectionVisible();	Scrolls a livegrid to make the selected row visible. Note: currently not working properly.

Table 10: DisplayGroup methods for working with the UI.

$DG::someDG->...	Explanation
addUserscript();	...
DG::disableButton('*someName*');	*someName* is the same as in the <button:'*someName*> tag. This is not a method but a function.
disableUIElement('*someName*');	not a method but a function
removeUIElement('*someName*');	not a method but a function

Table 11: DisplayGroup methods/functions for validating and modifying database operations within displaygroup handlers.

$DG::someDG->...	Explanation
addUserscript();	...
DG::mutablePendingInsertionDict();	This is not a method but a function.
pendingUpdateDict();	gives the dictionary that <dbweb> is about to insert. containts only user supplied data. master detail data is added later on.
mutablePendingUpdateDict();	gives the dictionary that <dbweb> is about to insert. containts only user supplied data. master detail data is added later on. can be used either as function or method.
mutablePendingUpdateDict();	gives the dictionary that <dbweb> is about to insert. containts only user supplied data. master detail data is added later on.
abortPendingUpdate();	stops the.

A. Installation instructions

<dbweb> is designed to run on Linux based Apache2 installations with mod_perl2 installed. Additional requirements can be resolved by means of listing 9. Copy the content of the github-hosted dbwebressources directory to your local dbwebressources directory. Copy the dbweb.pm, TempFileNames.pm and PropertyList.pm (only if you want to write your DisplayGroups in the old Apple PropertyList syntax) from the github repository to a location that is in the perl-inc path list of your apache2.

Now you only have to create a subdirectory for your application cluster directly below you local dbwebressources directory. You can now start writing your login.dgwapp file and the other application.dgwapp files in that directory.

Listing 9: Command line instruction (root privileges required) to install all required perl modules (tested on SuSE 10.3 64bit)

```
CPATH=/usr/include/apache2/modules/perl perl -MCPAN -e
'for $m ("YAML", "Apache::Session", "DBD::Pg",
"DBI::DBD", "Apache2::Request",
"JSON::XS", "DateTime") { install($m); }';
sudo ln -s /usr/lib/libapreq2.so.3 /usr/lib64/
```

Listing 10: Put this into your http.conf

```
1  LoadModule perl_module libexec/apache2/mod_perl.so
2  LoadModule apreq_module    libexec/apache2/mod_apreq2.so
3  Alias /dbwebressources "/path/to/your/dbwebressources"
4  PerlSetEnv dbwebressourceurl
5    "file:///path/to/your/dbwebressources/" # for performance (optional)
6  <Location /dbweb>
7      SetHandler perl-script
8      PerlHandler dbweb
9      PerlSendHeader On
10 </Location>
```

B. Supported Browsers

The supported browsers are listed in table 12. Browser support basically depends on prototypejs-support `http://www.prototypejs.org`. Opera did not support ContextMenus at time of writing. Others may also work, but are currently untested.

Table 12: Listing of the Web Browsers that are currently fully supported by <dbweb>.

Browser	Vendor	URL	min.
Mozilla Firefox	Mozilla Fnd.	`http://www.mozilla-europe.org/de/`	1.5
Apple Safari	Apple Inc.	`http://www.apple.com/safari/download/`	2
Google Chrome	Google Inc.	`http://www.google.com/chrome/`	5
Internet Explorer	Microsoft Inc.	`http://www.microsoft.com/germany/windows/downloads/ie/getitnow.mspx`	6

C. Example login.dgwapp

The login.dgwapp is accessed by default, when the browser is pointed to your application name below the dbweb URL component. Typically, here

Listing 11: Example for a minimal login.dgwapp

```
<html>
<div id="loginpanel" style="width:550px;">
<form:LOGIN label="My login application" action="redirectToApp">
  <var:user edittype=text editmode=inplace label="name">
  <var:password editmode=inplace edittype=password label="password">
  <foreach:errorDG>
    <tr> <td><var:Message  style="color:red;"></td> </tr>
  </foreach>
</form>
</div>
</html>

<DisplayGroups>{

LOGIN=
{ columns=("user","password");
  DataInSession=YES;
};
errorDG=
{ columns=("Message");
  primaryKey="Message";
  cache=YES;
};

}</DisplayGroups>

<perlfunc name="_earlyauto_">
{ if( DG::getGlobal('authaccept') eq 'failed')
  { $DG::errorDG->appendDictToCache( {Message =>'login failed'} );
    DG::addUserscript('Effect.Shake("loginpanel",{duration:0.2});');
  }
}</perlfunc>

<perlfunc name="redirectToApp">
{ use constant destination=>'mainapp';

  $DG::errorDG->clearCache();
  DG::setGlobal( {'authaccept'=>'failed' } );

  my $login=$DG::LOGIN->selectedDict();
  if($login->{user} eq 'me' && $login->{password} eq 'myS3cretPassw')
  { DG::setGlobal ( {'authaccept'=>'ok' } );
    DG::redirectTo(destination);
```

```
44 |   } else
45 |   { $DG::errorDG->appendDictToCache( {Message =>'login failed'} )
46 |   }
47 |}</perlfunc>
```