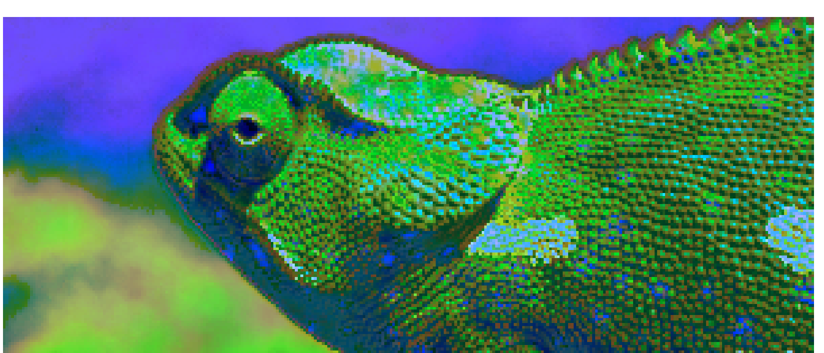


Blue Chameleon CMS
(Packages)

For links outside this document,
download the relevant chapter or the

Blue Chameleon Content Management System full documentation.

May 8, 2012

Contents

1 Configuring and administration **13**
1.1 System interface 13
1.2 Users and usergroups 14
1.2.1 Users 14
1.2.1.1 User groups that a user belongs to 15
1.2.1.2 User management through time 16
1.2.2 User groups 16
1.2.3 User groups access rights 16
1.3 Blue Chameleon Content Management System user rights 17
1.3.1 Rights on headlines 17
1.3.2 Rights on Articles 18
1.3.3 Rights on page objects 18
1.3.4 Rights on Images 18
1.3.5 Rights on Search 18
1.3.6 Rights on URLs 18
1.3.7 Rights on columns 19
1.3.8 Rights on publishing 19
1.3.9 Rights on Data Base Publishing 20
1.3.10 Rights on Editions 20
1.3.11 Rights on Background Frames 20
1.3.12 Rights on Downloads 20
1.3.13 Rights on Site management 21
1.3.14 Rights on Article Models 21
1.3.15 Rights on Headline Models 21
1.3.16 Rights on Notice boards 21
1.3.17 Rights on System log 22
1.3.18 Rights on Scripts 22
1.3.19 Administrator rights 22
1.3.20 Rights on Hermetic groups 22
1.3.21 Information level 23
1.3.22 Rights on Books 23
1.3.23 Rights on Book Models 23
1.3.24 Java option menu 23
1.3.25 Connection speed 23
1.3.26 Rights on User management 24
1.3.27 Rights on Menus, Menu Templates 24

1.3.28 Rights on Packages . . . 24
1.3.29 Rights on Forms . . . 24
1.4 Columns . . . 25
1.4.1 Publisher and objects . . . 25
1.4.2 Creating columns . . . 25
1.4.3 Managing columns . . . 26
1.4.3.1 Changing columns . . . 26
1.4.3.2 Updating columns . . . 26
1.4.3.3 Backups . . . 27
1.4.4 Blue Chameleon directory structure . . . 28
1.4.4.1 Virtual columns . . . 28
1.5 Publisher settings . . . 28
1.5.1 File types . . . 29
1.6 FTP profiles . . . 30

2 Composing content **33**
2.1 Principles . . . 33
2.2 Headlines models . . . 33
2.2.1 Creating a new headline model . . . 34
2.2.2 Updating a headline model . . . 35
2.2.3 Contents of a headline model . . . 36
2.2.3.1 Enriching a headline model with frames . . . 37
2.2.3.1.1 Frame templates used by a headline model . . . 39
2.2.3.1.2 Updating frame number . . . 40
2.2.3.2 Meta Tags . . . 40
2.3 Frame templates . . . 41
2.3.1 Creating and updating frame templates . . . 41
2.3.2 "List" frames . . . 42
2.3.3 "Leaf" frames . . . 43
2.3.4 Frame style . . . 44
2.4 Headlines : creation and editing . . . 45
2.4.1 Editing a headline : a basic example . . . 45
2.4.2 Editing a leaf frame . . . 46
2.4.3 Editing a list frame . . . 47
2.4.3.1 Example for a leaf frame as used by a list . . . 48
2.5 Layouts . . . 50
2.6 Articles and article models . . . 50
2.7 Managing pages . . . 50
2.7.1 Publishing pages . . . 50
2.7.1.1 Multiple publish . . . 51
2.7.1.2 Scheduled publishing . . . 52
2.7.2 Properties of headlines . . . 53
2.7.3 Other management options for pages . . . 54

3 Object management **57**
3.1 Images 57
3.1.1 Uploading an image 57
3.1.2 Image folders 59
3.1.3 Managing images 59
3.1.4 Integrating images 59
3.2 Links 60
3.2.1 Creating a URL link 61
3.2.2 Managing URL links 62
3.2.3 Integrating links 62
3.3 Scripts 64
3.3.1 Creating a script - Management 64
3.3.2 Integrating a frame script 65
3.3.3 Integrating a link style script 66
3.4 Downloads 67
3.4.1 Creating a download 67
3.4.2 Managing downloads 68
3.4.3 Integrating downloads 68
3.5 Menus 69
3.5.1 Menu scripts 69
3.5.2 Menu models 71
3.5.3 Creation of a menu 71
3.5.3.1 Filling of a menu 72
3.5.3.2 Menu result and integration 74
3.5.4 More complex menus 75
3.5.4.1 A menu script that handles children menu items 75
3.5.4.2 A menu with children menu items 75
3.6 Forms 78
3.6.1 Form models 78
3.6.1.1 Form model management - parameters 79
3.6.2 Defining forms 79
3.6.2.1 Updating a form 80
3.6.3 Composing a form 80
3.6.3.1 Editing a form page 81
3.6.3.2 Form page element types 82
3.6.3.3 Editing a form page element 83
3.6.3.4 After form is submitted 84
3.6.3.5 A form result 84
3.6.3.6 A form with several pages 84
3.6.4 Integrating a form 85
3.7 Variable files 86
3.7.1 Example of varfile contents 86
3.7.2 Varfiles as variables 87

4 Other elements **89**
4.1 Site management . . . 89
4.1.1 Uploading files . . . 89
4.1.2 Creating directories . . . 90
4.1.3 Managing files and directories . . . 91
4.1.4 Executing `SQL` . . . 91
4.1.5 Export and Import . . . 93
4.1.5.1 Generating export files - importing them . . . 93
4.1.5.2 Export file contents . . . 94
4.2 Model defaults . . . 96
4.2.1 Model defaults and headlines . . . 96
4.3 Books . . . 97
4.3.1 Book models . . . 97
4.3.2 Creating books . . . 97
4.3.3 Modifying a book . . . 99
4.4 Notice boards . . . 100
4.4.1 Creating and managing notice boards . . . 100
4.4.2 Association with a column . . . 100
4.4.3 Notice board entries . . . 101
4.5 Editions . . . 102
4.5.1 General edition management . . . 103
4.5.2 Publishing editions . . . 104
4.6 Options . . . 104
4.6.1 Version . . . 104
4.6.2 System Logs . . . 105
4.6.3 Expiration report . . . 106
4.7 Searching for content . . . 106
4.8 Management of objects . . . 108

5 Packages **109**
5.1 Mandatory scripts . . . 109
5.1.1 `PackageInstall.phs` . . . 109
5.1.2 `PackageGroupModify.phs`, `PackageGroupModify1.phs` . . . 110
5.1.2.1 Result : package group rights . . . 110
5.1.3 `PackageMain.phs` . . . 113
5.1.3.1 Result : package main page . . . 114
5.1.4 `PackageUninstall.phs` . . . 116
5.2 Package general management . . . 117
5.2.1 Installing a package . . . 117
5.2.2 Updating a package . . . 117
5.2.3 Uninstalling a package . . . 118
5.3 Using object types and objects to compose headlines . . . 118
5.3.1 Enabling a leaf frame to handle packages and an object . . . 119
5.3.2 `PackageObjTypeList.phs` . . . 120
5.3.3 `PackageObjectList.phs` . . . 122
5.3.4 `PackageObjectName.phs` . . . 122

5.3.5 PackageObjectView.phs . . . 124
5.4 Using attributes . . . 125
5.4.1 Enabling a frame to handle attributes . . . 125
5.4.2 Scripts for attribute . . . 126
5.5 Featuring multiple objects on the same headline . . . 127
5.5.1 A leaf frame that handles multiple objects . . . 128
5.5.2 Packages : defining more of them . . . 130
5.5.3 Object and attribute scripts for this case (guidelines) . . . 130
5.5.3.1 The PackageObjectView.phs script for this example . . . 132

6 Annex **135**
6.1 Leaf frame variables . . . 135
6.1.1 "Table" variables . . . 137
6.1.2 "Select" variables . . . 137
6.1.3 "SQL select" variables . . . 138
6.2 System script glossary . . . 138
6.2.1 Headline model scripts . . . 138
6.2.2 Leaf frame model scripts . . . 139
6.3 Icon glossary . . . 141
6.4 Form page elements : detailed . . . 143
6.4.1 "Active" elements . . . 143
6.4.1.1 Checkboxes . . . 143
6.4.1.2 Radiobuttons . . . 144
6.4.1.3 Listboxes . . . 144
6.4.1.4 Input field and text area . . . 145
6.4.1.5 Form label - 'with check box' . . . 146
6.4.2 "Passive" elements . . . 146
6.4.2.1 Hidden field . . . 146
6.4.2.2 Form image . . . 146
6.4.2.3 Form label - 'normal' . . . 146
6.4.2.4 Form separator . . . 147

List of Figures

1.1 The Publisher's main page. 13
1.2 Defining a new user. 15
1.3 Groups a user belongs to. 15
1.4 Defining a user group. 16
1.5 The most important access rights as defined for a user group. 17
1.6 Creating a new column. 26
1.7 Updating a column. 27
1.8 The properties of your Publisher. 29
1.9 Defining a FTP profile. 31

2.1 Creating a headline model. 34
2.2 Here, a headline model can be updated and its children frame templates be chosen. 36
2.3 Thanks to this, the pages created on this model will have meta-tags called Author, LastUpdate and a Page Title. 41
2.4 Creating a new headline. 45
2.5 First edit ; as both frames use only one template ("Basic text frame" and "Side Menu List"), so clicking on the icon shows them immediately. . . . 46
2.6 Filling in with information the leaf's various variables. 47
2.7 A preview of the page. 48
2.8 Populating a list frame. 49
2.9 Publishing several headlines in one single time. 51
2.10 Configuring and viewing data related to headline "Home Page". 53
2.11 Managing headlines in a file system way. 55

3.1 Uploading an image. 58
3.2 Creating a folder for images. 59
3.3 From here, all images and folders can be globally managed. 60
3.4 Integrating an uploaded image. 61
3.5 Creating a link. 62
3.6 Putting a previously-defined URL link. 63
3.7 Creating a script to be used in a leaf frame. 65
3.8 Uploading a file that will be used for downloading purposes. 67
3.9 Integrating a download, with a 'download' variable. 69
3.10 Creating a menu model. 71
3.11 Creating a menu "Basic top menu" based upon model 'Top Menu'. . . . 72
3.12 Creating a form model. 78

3.13 Adding parameters to a form model. 79
3.14 Creating a form based on the example model. 79

4.1 Uploading a file to be used in headline contents. 90
4.2 Creating a new directory, directly under the «#SQLWWWHOME#» root. . . . 91
4.3 Editing a text-based file. 92
4.4 Outputting the contents of a table. 93
4.5 Exporting the current column. 94
4.6 An example of export file contents. 95
4.7 Creating a model default for a headline model. 96
4.8 Creating a book model. 98
4.9 Creating a book. 98
4.10 From here, book elements can be managed. 100
4.11 Creating a general notice board. 101
4.12 Manually adding two entries to this notice board. 102
4.13 Checking one's own activity for last week. 106
4.14 Performing a search on objects which publishing name begin by 'Josh'. . 107
4.15 From here, any object can be managed. 108

5.1 Modifying this package's rights for a particular user group. 110
5.2 The main page for package Festivals. 114
5.3 The first install of the package "Festivals". 117
5.4 The process of selecting an object type and an object for a leaf frame that is package-enabled. 119
5.5 A selection of object types. 121
5.6 Finally choosing the object. 123
5.7 A page design that calls for multiple objects to be featured. 128
5.8 The 3-object frame after all relevant choices of objects and attributes have been made. 132

6.1 Composing a checkbox selection for a form. 144

List of Tables

5.1 Scripts, functions and variables as used for attribute setting 126

6.1 Leaf frame variables 136
6.2 Leaf frame variables (continued) 137
6.3 System scripts for headline, list and container models 139
6.4 System scripts for leaves 140
6.5 System scripts for leaves (continued) 141
6.6 Blue Chameleon Content Management icons 142
6.7 Blue Chameleon Content Management icons (continued) 143

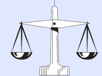
Chapter 5

Packages

Through use of Blue Chameleon Content Management System, the need for extended functions might arise sooner or later, requiring data to be uploaded, processed and managed. This implies the enabling of database capabilities as well as more complex script features.

A package is composed of several `.phs` scripts, which make extended use of Blue Chameleon's own language, `OSL` ; if you are unfamiliar with it, please check `OSL`'s User Guide.

Amongst the scripts that constitute a package, most of them aim at managing the package itself and its objects inside the system. They have to be integrated for the package to properly function.


User group rights for managing Packages are detailed at 1.3.28.

5.1 Mandatory scripts

What follows describes the scripts that **must** be included into your package for it to function properly. A package is, as an end result, a library compiled from those scripts, as similarly described in **Developing Your Blue Chameleon Add-On**.

This library will then be uploaded in your system and its features will be available for page composing.

5.1.1 `PackageInstall.phs`

This script contains the necessary instructions to register and update the package inside the system, according to a version variable. It contains the `SQL` table creation instructions to execute the first time and, afterwards, table updates.

An example is given at Code 6. There :

- the _PkgeId variable as given at the beginning identifies your package and must be unique for each package ;
- things are done according to variable piPackageVersion (which value is known whenever this script is called) and, according to their success (as given by variable _Ok, which can be set through evaluations of «#SQLSTATUS#»es), the value of piPackageVersion is incremented. See "Updating a package" (5.2.2) for more details.

5.1.2 PackageGroupModify.phs, PackageGroupModify1.phs

These scripts aim at respectively configuring the group access rights (1.3) for the package and modifying them. They are featured at Code 7 and 8.

In the first, a FORM is generated : one of the CGI variables, _Script, is set to YourPackageName:PackageGroupModify1.phs and a group right is selected through a menu (which options can be expanded if needed). In the second, the update is propagated to the OPS table that manages user groups.

5.1.2.1 Result : package group rights

By accessing the group user rights for a package, what is shown in Fig.5.1 will be then available : a list of uploaded packages with for each a possibility to **Modify rights**, in the layout as defined by script PackageGroupModify.phs.

Figure 5.1: Modifying this package's rights for a particular user group.

Code 6 A template for `PackageInstall.phs`.

```
« * Parameters:
 * piPackageVersion, currently installed version of the package
 * iPackagePublisherId, Id of publisher for which package is to be installed
 *»

«LET _PkgeId=150»
«VAR NEW _CptError=0»
«VAR NEW _Ok=1»

«IF «piPackageVersion»<1»   «* Create tables+indexes *»
  «SQLSTATUS TRANSACTION BEGIN»

    «SQL CREATE TABLE OPS_FESTIVALS(
      idFestival int, azName varchar(128),
      azCity varchar(64), iDate int, idArtist int,
    iOrder int)»

    «SQL CREATE TABLE OPS_ARTISTS(
      idArtist int, azName varchar(128), idMusicalStyle int, azCountry
varchar(64))»

    (Create some other tables...)

    «SQLSTATUS TRANSACTION END»
«ENDIF»

«DIRECTORY CREATE images/artists»
(Create some other directories...)

«IF «piPackageVersion»<2»
  «* Upgrade package to version 1*»
«ENDIF»

«IF _Ok==1»
  «LET _PackageVersion=1»
«ENDIF»
```

Code 7 A template for PackageGroupModify.phs.

```
«SQLEXEC STRING _GName=select azGroupName from «#SQLWPA#»OPSP_GROUPS where
iGroupId=«iPackageGroupId»»
<BR><STRONG>«_GName»:</STRONG> Group Access rights for package
<STRONG>«azPackageName»</STRONG>:

<FORM METHOD="post" ACTION="/Scripts/sql.exe">
 <INPUT TYPE="hidden" NAME="SqlDB" VALUE="«SqlDB»">
 <INPUT TYPE="hidden" NAME="Sql" VALUE="opsPackage.phs">
 <INPUT TYPE="hidden" NAME="_Script" VALUE="Festivals:PackageGroupModify1.phs">
 <INPUT TYPE="hidden" NAME="xid" VALUE="«xid»">
 <INPUT TYPE="hidden" NAME="iPackageGroupId" VALUE="«iPackageGroupId»">
 <INPUT TYPE="hidden" NAME="iPackageId" VALUE="«iPackageId»">
 <INPUT TYPE="hidden" NAME="_OPSMenu" VALUE="«_OPSMenu»">

 «VAR NEW _GroupRights»
 «SQLEXEC STRING _GroupRights=SELECT azGroupRights FROM «#SQLWPA#»OPSP_GROUPS
WHERE iPublisherId=«iPackagePublisherId» AND iGroupId=«iPackageGroupId» AND
iPackageId=«iPackageId»»

 <TABLE>
  <TR>
   <TD>Festivals :</TD>
   <TD>
    <SELECT NAME="pos0">
     <OPTION VALUE="48"«IF _GroupRights[0]==48»SELECTED«ENDIF»>No rights</OPTION>
     <OPTION VALUE="49"«IF _GroupRights[0]==49»SELECTED«ENDIF»>View rights</OPTION>
     <OPTION VALUE="50"«IF _GroupRights[0]==50»SELECTED«ENDIF»>Modify rights</OPTION>
     <OPTION VALUE="51"«IF _GroupRights[0]>=51»SELECTED«ENDIF»>Validate
rights</OPTION>
    </SELECT>
   </TD>
  </TR>
 </TABLE>

<BR><INPUT TYPE="submit" VALUE="Modify">
</FORM>
```

Code 8 The script `PackageGroupModify1.phs`.

```
«VAR NEW _GroupRights="00000000000000000000"»

«SQLEXEC STRING _GroupRights=SELECT azGroupRights FROM «#SQLWPA#»OPSP_GROUPS
WHERE iPublisherId=«iPackagePublisherId» AND iGroupId=«iPackageGroupId»»
«SQLEXEC STRING _GName=SELECT azGroupName FROM «#SQLWPA#»OPSP_GROUPS WHERE
iGroupId=«iPackageGroupId»»
«LET _GroupRights[0]=@int(pos0)»

«SQL UPDATE «#SQLWPA#»OPSP_GROUPS SET azGroupRights='«_GroupRights»' WHERE
iPackageId=«iPackageId» AND iPublisherId=«iPackagePublisherId» AND
iGroupId=«iPackageGroupId»»
«IF #SQLSTATUS#<>1»
  <BR>ERROR: Command 'UPDATE «#SQLWPA#»OPSP_GROUPS SET
  azGroupRights='«_GroupRights»' WHERE iPackageId=«iPackageId» AND
  iPublisherId=«iPackagePublisherId» AND iGroupId=«iPackageGroupId»' failed.<BR>
«ELSE»
  <BR>«_GName» Group rights for package <STRONG>«azPackageName»</STRONG>
  have been successfully updated.
«ENDIF»
```

5.1.3 `PackageMain.phs`

When, from the main page, the package name is clicked, a content as generated by the `PackageMain.phs` script is featured. It is used to manage the main data as used in the package. Therefore, it is mostly free-form apart from the loading of group rights which can be used to allow certain actions, such as adding a new element, deleting another...

Codes 9 and 10 feature an example of what can be done. There :

- the content of one the main tables (as defined in the `PackageInstall.phs` script, 5.1.1) is shown on the screen ;
- for each line, a icon "launches" an `EditFestival.phs` script aimed at for instance modifying the data related to that festival ;
- a icon "launches" yet another script (`AddAnArtist.phs`) aimed at for instance adding a new artist to the festival ;
- at last, at the bottom of the page, a icon could be added, doing the same for a script aimed at adding a new element to the table.

It is to note that, while the bulk content of that package's main page is totally free, there are several things to comply to :

- `INPUT`s of the form as featured in Code 9 in black must be present ;

- a form always redirects to `OPS`'s `opsPackage.phs` script. The script that has to be executed is stored in the CGI variable `_Script`, always in the `YourPackageName:ScriptToExecute.phs` fashion.

5.1.3.1 Result : package main page

By accessing a package from the main page, the output of script `PackageMain.phs` is shown. The current example is featured at Fig.5.2.

Festivals :

Festival List

Name	Date	City	Artists	
Lapin Kulta Heavy Metal Fest	22/6/2011	Rovaniemi	5	
Mittelalter und Goth Fest	13/8/2011	Munich	4	
Rock and Stuff	3/10/2011	Budapest	6	

+

Figure 5.2: The main page for package Festivals.

Code 9 An example for PackageMain.phs (I).

```
«IF azPackageRights[0] < 49»<B>You haven't got the right to use this
package</B>«ELSE»

<B>Festival List</B><P>

<FORM NAME="category" ACTION="/Scripts/sql.exe" METHOD="post">
 <INPUT TYPE="hidden" NAME="SqlDB" VALUE="«SqlDB»">
 <INPUT TYPE="hidden" NAME="xid" VALUE="«xid»">
 <INPUT TYPE="hidden" NAME="Sql" VALUE="opsPackage.phs">
 <INPUT TYPE="hidden" NAME="_Script" VALUE="">
 <INPUT TYPE="hidden" NAME="_Context" VALUE="«_Context»">
 <INPUT TYPE="hidden" NAME="iPackageId" VALUE="«iPackageId»">
 <INPUT TYPE="hidden" NAME="_Back" VALUE="">
 <INPUT TYPE="hidden" NAME="iPContext" VALUE="1">

 <INPUT TYPE="hidden" NAME="_FestivalId" VALUE="0">

 <TABLE CELLPADDING="2" CELLSPACING="0" BORDER="1">
  <TR>
   <TH BGCOLOR="#CCCCCC" nowrap>Name</TH>
   <TH BGCOLOR="#CCCCCC" nowrap>Date</TH>
   <TH BGCOLOR="#CCCCCC" nowrap>City</TH>
   <TH BGCOLOR="#CCCCCC" nowrap>Artists</TH>
   <TH BGCOLOR="#CCCCCC" nowrap> </TH>
  </TR>
  «SQLOUTPUT»
   <TR>
    <TD>«azName»</TD>
    <TD>«=@daytimeday(iDate)»/«=@daytimemonth(iMonth)»/«=@daytimeyear(iDate)»</TD>
    <TD>«azCity»</TD>
    «SQLEXEC INT nbArtists = SELECT COUNT(*) FROM OPS_FESTIVALS WHERE
    idFestival=«idFestival»»
    <TD align="center">«nbArtists»</TD>
    <TD>
     «IF azPackageRights[0]>49»
      <IMG SRC="«#SQLWWWHOME#»/img/data.gif"
      STYLE="cursor:pointer" onClick="onEditFestival(«idFestival»)"/>
       <IMG SRC="«#SQLWWWHOME#»/img/data_ext.gif" STYLE="cursor:pointer"
      onClick="onAddArtist(«idFestival»)"/>
     «ENDIF»
    </TD>
   </TR>
  «/SQLOUTPUT SELECT DISTINCT(azName), idFestival, iDate, azCity FROM
OPS_FESTIVALS ORDER BY iDate»
 </TABLE>

(...)

</FORM>

«ENDIF»
```

Code 10 An example for `PackageMain.phs` (II : `javascript` part).

```
<SCRIPT LANGUAGE="javascript">

 function onEditFestival(i)
  {
   document.category._FestivalId.value=i;
   document.category._Script.value="Festivals:EditFestival.phs";
   document.category.submit();
  }

 function onAddArtist(i)
  {
   document.category._FestivalId.value=i;
   document.category._Script.value="Festivals:AddAnArtist.phs";
   document.category.submit();
  }

(...)

</SCRIPT>
```

5.1.4 PackageUninstall.phs

This script is launched when an uninstall of a package (5.2.3) is done. It can be used, as featured in Code 11, to erase all table data used by the package.

Code 11 An example for `PackageUninstall.phs`.

```
«SQL DROP TABLE OSL_FESTIVALS»
...
(Drop all package's tables)

«DIRECTORY DELETE images/artists»
...
(Delete all created folders)
```

5.2 Package general management

5.2.1 Installing a package

Once at least all mandatory files as described in 5.1 have been written and compiled with the `sqllib` compiler and the compiled library has been uploaded to the Publisher's `SysLibHome`, the package is ready to be installed, as featured on Fig.5.3.

Package / Install :

Package Install

Select the package to install:

Package Library: Festivals Please enter the root of the package library.

Package Name: Festivals

Administrator Group: Administrator

Install

Package Install

The package version 1 has been installed.

Figure 5.3: The first install of the package "Festivals".

5.2.2 Updating a package

Through time, a package's contents may be updated : new scripts can be added, others modified, some `SQL` updates have to be performed... In any case, the library will have to be compiled and uploaded again.

If the case of `SQL` updates and/or actions that necessitate `OSL` code to be executed once (for instance, `DIRECTORY CREATE`), they have to be featured in the `PackageInstall.phs` script, updated as shown at Code 12.

In the case of an update of `PackageInstall.phs`, the package will have to be re-installed as described above ; on success, the screen will confirm that package has been upgraded :

Package Install

The package has been upgraded to version 2.

Code 12 An append to `PackageInstall.phs` to upgrade the package to Version 2.

```
[Instructions for Version 1]

«IF «piPackageVersion»<3»
  «* Upgrade package to version 2*»
  «SQL UPDATE (...)»
  «LET _Ok=#SQLSTATUS#»
«ENDIF»

«IF _Ok==1»
  «LET _PackageVersion=2»
«ENDIF»
```

After tables are created, queries on them can be executed in a simpler, quicker way through Site management / Execute SQL (4.1.4).

The current versions of all installed packages can easily be checked thanks in **Options** (4.6.1).

5.2.3 Uninstalling a package

Through **Packages** / **Uninstall**, a list of all packages is displayed ; clicking on a package will execute the `PackageUninstall.phs` script that had been defined for this package (5.1.4) and will remove it from the main screen. According to this script's contents, data may be erased irreversibly.

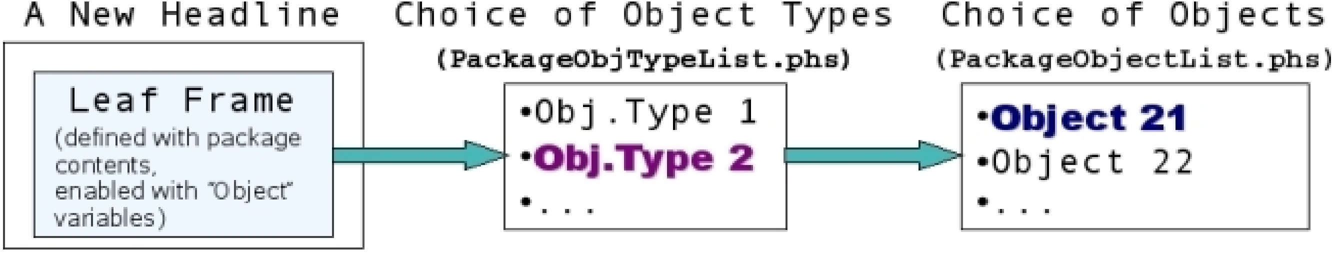
5.3 Using object types and objects to compose headlines

Packages, with their possibility to execute `.phs` scripts, offer vast advantages for headline creation (2.4). More precisely, one single script can be used to produce several different headlines, according to variables that are called object types and objects.

In Blue Chameleon Content Management System, these are managed in the way shown at Fig.5.4.

There :

- as leaf frame (2.4.2) is edited within headline modify context, a script `PackageTypeObjList.phs` displays a choice of several object types, one of which is



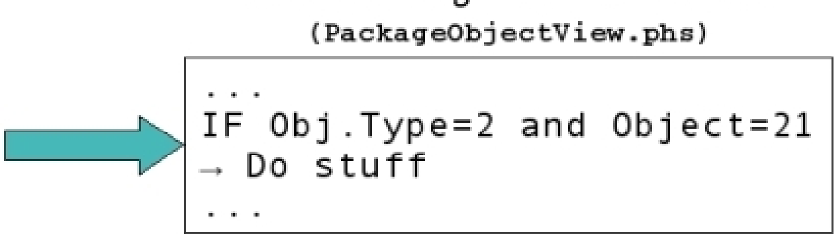


Figure 5.4: The process of selecting an object type and an object for a leaf frame that is package-enabled.

chosen ;

- then, according to the chosen value, a choice of objects is shown through `PackageObjectList.phs` ;
- upon selection of object, the couple of values (Object Type, Object) steers to the right place of `PackageObjectView.phs` in order to generate, upon validation, an information-specific headline that reflects the choices made.

In the current example, in order to make information pages for festivals and artists, a practical application would be :

- make object types as "Festivals" and "Artists" ;
- for each, make objects as individual festivals and individual artists.

The following shows how this practical example is implemented.

5.3.1 Enabling a leaf frame to handle packages and an object

A leaf frame "Info Page" is created (2.3.1) to handle packages. Code 13 then shows what has to be inputed for the frame's content (during creation or modification).

It is to note that Code 13 is the same whatever the leaf frame or the package are. Now, for the leaf frame's variables, as shown in Code 14, the contents depend on what should be handled in the frame's layout.

In this code, "PO" stands for *package object* and "14" is the identifier for that type; in between, a title for the object (which can be here either a festival or an artist) is given.

Code 13 A leaf frame's contents to handle packages, for a single object :

Headline models / **Modify** / **Info Page** :

```
«INCLUDE PackageObjectView.phs;_PackageId=«POPackageId»;
_ObjectTypeId=«POObjectType»;_ObjectId=«POObjectId»;
_ObjectAttrType=«POStyleAttrType»;_ObjectAttrValue=«POStyleAttrValue»»
```

Code 14 A package-oriented frame's variable for handling an object :

Headline models / **Variables** / **Info Page** :

```
PO;Festival/Artist;14;
```

5.3.2 PackageObjTypeList.phs

Code 15 shows how the code that proposes the choice between object types can be written.

Code 15 An example for `PackageObjTypeList.phs` :

```
<TABLE cellpadding="2" cellspacing="0" border="0">
 <TR>
  <TD><A HREF="javascript:selectObjectType(150,1)">Festivals</A></TD>
 </TR>
 <TR>
  <TR><TD><A HREF="javascript:selectObjectType(150,2)">Artists</A></TD>
 </TR>
</TABLE>
```

There, a choice between object types is laid out through a table ; each object type must use function `selectObjectType`, with for arguments the identifier of the package and a distinct number that will identify the object type.

Fig.5.5 shows the result of this script. On the *Headline Modify Page* for headline "Rock and Stuff" - which template accepts leaf frame Info Page - as frame is edited, the "Festival/Artist" variable previously defined for the frame is visible, along with a [Choose Object] button.

On click, the list of packages is shown and, upon selection of the "Festivals" one, choice is then offered between the two defined object types.

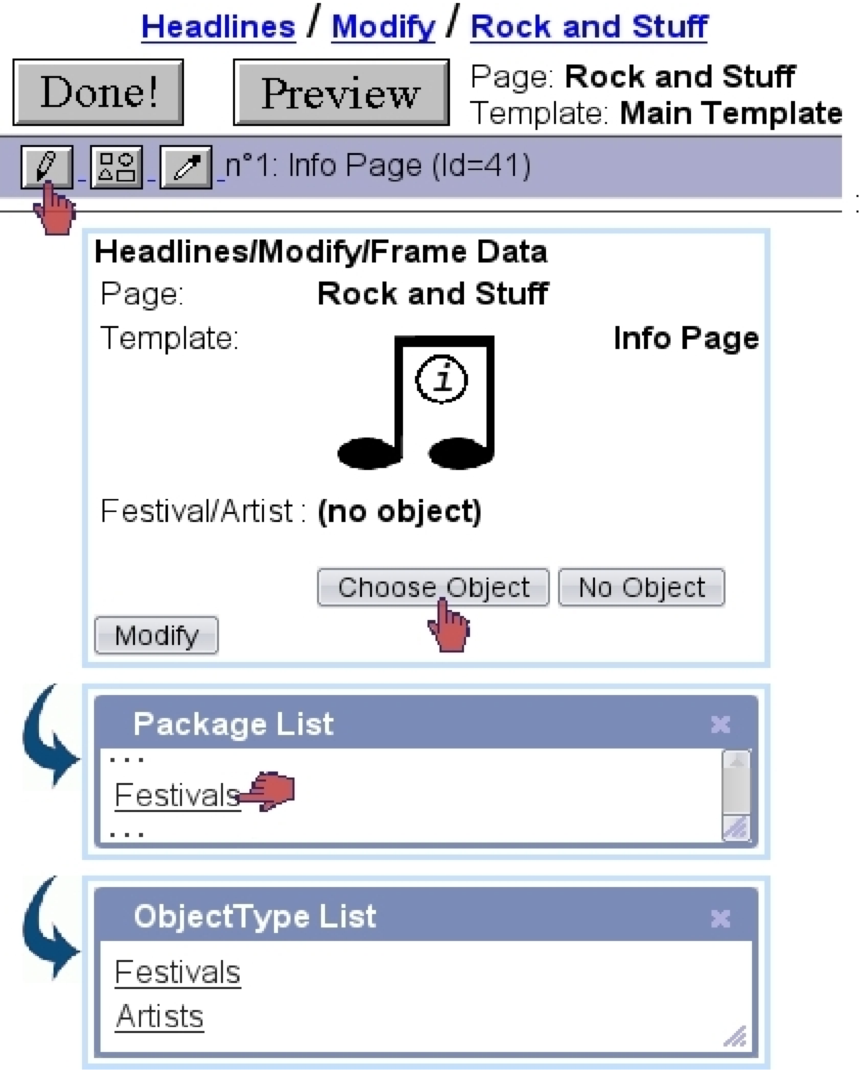


Figure 5.5: A selection of object types.

5.3.3 PackageObjectList.phs

This script must use the variable `iObjectType` as set by the previous script to select the right object list. As Code 16 shows, each object line has to redirect to the `selectObject` function, which arguments are package identifier, object type identifier, and a distinct number that will identify the object.

Code 16 An example for `PackageObjectList.phs` :

```
<TABLE cellpadding="2" cellspacing="0" border="0">
 «IF @toi(iObjectType)==1» «*Festivals*»
  <TR>
   <TD><A HREF="javascript:selectObject(150,1,0)">All festivals</A></TD>
  </TR>
  «SQLOUTPUT»
   <TR>
    <TD><A HREF="javascript:selectObject(150,1,«idFestival»)">«azName»</A></TD>
   </TR>
  «/SQLOUTPUT SELECT DISTINCT(azName), idFestival FROM OPS_FESTIVALS ORDER BY
  azName»
 «ELSEIF @toi(iObjectType)==2» «*Artists*»
  <TR>
   <TD><A HREF="javascript:selectObject(150,2,0)">All artists</A></TD>
  </TR>
  «SQLOUTPUT»
   <TR>
    <TD><AHREF="javascript:selectObject(150,2,«idArtist»)">«azName»</A></TD>
   </TR>
  «/SQLOUTPUT azName FROM OPS_ARTISTS ORDER BY azName»
 «ENDIF»
</TABLE>
```

The result of this script is featured at Fig.5.6, where, after "Festivals" from the object type list is clicked, the list of festivals is proposed. Upon final selection, the page shows well that "Festival/Artist" has been set to "Rock and Stuff".

If the wrong object or object type was selected, it is possible to reinitialize the Object variable with the No Object button.

The object name as featured on the final screen is set by a small script, `PackageObjectName.phs`, described as follows.

5.3.4 PackageObjectName.phs

As shown in Code 17, following a similar structure as the previous one, this script makes use of the now-existing `iObjectId` value (referring to the selected object) to simply fetch

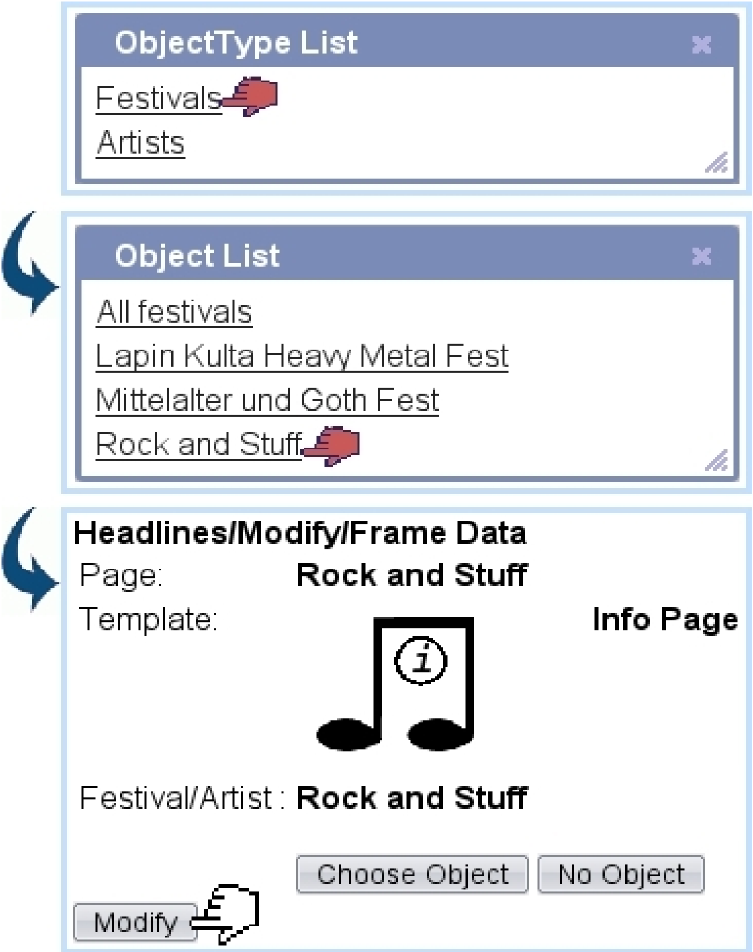


Figure 5.6: Finally choosing the object.

its name.

Code 17 An example for `PackageObjectName.phs` :

```
«VAR NEW _Name=""»
«IF @toi(iObjectType)==1» «*Festivals*»
 «SQLEXEC STRING _Name=SELECT azName FROM OPS_FESTIVALS WHERE
 idFestival=«=@toi(iObjectId)»»
«ELSEIF @toi(iObjectType)==2» «*Artists*»
 «SQLEXEC STRING _Name=SELECT azName FROM OPS_ARTISTS WHERE
 idArtist=«=@toi(iObjectId)»»
«ENDIF»
«<B>_Name</B>»
```

5.3.5 PackageObjectView.phs

This script, as executed when the Modify button is clicked after object was set, will finally generate the headline.

Variables available there are `_ObjectTypeId` and `_ObjectId`, respectively identifying the selected object type and object.

Code 18 then shows a template of how `PackageObjectView.phs` can be composed : a first test on `_ObjectTypeId` steers to the right part of the script, where an other test on `_ObjectId` decides which festival data has to be shown.

It is to note that this script, thanks to the object concept, works with any festival or artist for which an information page has to be done : **a single script to make them all**.

Code 18 An example for `PackageObjectView.phs` :

```
(...)

«IF @toi(_ObjectTypeId)==1» «*Festivals*»

 «VAR NEW _Where=""»
 «IF @toi(_ObjectId)<>0»
  «LET _Where=" WHERE idFestival=@toi(_ObjectId) "»
 «ENDIF»

 «SQLOUTPUT»

  (Display formatted festival information)

 «/SQLOUTPUT SELECT * FROM OPS_FESTIVALS «_Where» ORDER BY azName»

«ELSEIF @toi(_ObjectTypeId)==2» «*Artists*»

(...)

«ENDIF»
```

5.4 Using attributes

The above described how to produce specific pages with the help of an "Object" variable. Need may arise though to particularize a page even more ; Blue Chameleon Content Management allows it, with the help of an "Attribute".

Basically, an attribute functions works as an object ; it is defined in the first place as a leaf frame's variable, then dedicated scripts (following the same pattern as the Object ones, with different variable names) are written to handle attribute types and subsequent attributes.

Finally, when a headline's package-enabled leaf frame is edited, the attribute variable will be featured below the object as new configuring option.

5.4.1 Enabling a frame to handle attributes

An attribute variable is added in the frame's variables ; it is named "POStyle" and with the identifier "15". In this example, it will bear the name "Style/Mode" :

Headline models / **Variables** / **Info Page** :

Table 5.1: Scripts, functions and variables as used for attribute setting

`PackageAttTypeList.phs`

Shows a list of attribute types :

```
...
<TR><TD><A HREF="javascript:AttributeType(1)">Style</A></TD></TR>
<TR><TD><A HREF="javascript:AttributeType(2)">Mode</A></TD></TR>
...
```

`PackageAttValList.phs`

Shows the list of attributes, corresponding to the selected type :

```
...
«IF @toi(iAttrType)==2»
 <TR><TD><A HREF="javascript:AttributeValue(1)">View only</A></TD></TR>
 <TR><TD><A HREF="javascript:AttributeValue(2)">Ticket book</A></TD></TR>
«ENDIF»
```

`PackageAttrName.phs`

Outputs the name of the selected attribute :

```
...
«ELSEIF @toi(iObjectType)==2» «*Mode*»
 «IF @toi(iAttrValue)==1»
  <B>View only</B>
 «ELSEIF @toi(iAttrValue)==2» «*Mode*»
  <B>Ticket book</B>
 «ENDIF»
«ENDIF»
```

```
PO;Festival/Artist;14

POStyle;Style/Mode;15
```

5.4.2 Scripts for attribute

Table 5.1 shows how what scripts are attribute's own, along with code snippets.

When the `PackageAttTypeList.phs`, `PackageAttValList.phs` and `PackageAttrName.phs` scripts have been written, the `_ObjectAttrType` and `_ObjectAttrValue` variables can then be used within the `PackageObjectView.phs` script along with the object variables. Here is an example on how they can be integrated :

```
(...)
«IF @toi(_ObjectTypeId)==1» «*Festivals*»
```

```
 «VAR NEW _Where=""»
 «IF @toi(_ObjectId)<>0»
  «LET _Where=" WHERE idFestival=@toi(_ObjectId) "»
 «ENDIF»

 «SQLOUTPUT»
  (Display formatted festival information)
  «IF @toi(_ObjectAttrType)==2 && @toi(_ObjectAttrValue)==2 »«*Mode :  Book
ticket*»
   <IMG SRC="«#SQLWWWHOME#»/img/BookNow.gif" STYLE="cursor:pointer"
onClick="onBookNow(«idFestival»)"/>
  «ENDIF»
 «/SQLOUTPUT SELECT * FROM OPS_FESTIVALS «_Where» ORDER BY azName»

«ELSEIF @toi(_ObjectTypeId)==2» «*Artists*»
(...)
```

5.5 Featuring multiple objects on the same headline

The previous sections aimed at describing how a headline containing a single object could be composed ; now, the flexibility of Blue Chameleon Content Management System's object and attribute system allows, as a matter of fact, more than one object to be featured on the *same* headline.

Indeed, a need for such might arise for instance when multiple elements (as gathered from the database) should be displayed on the screen, but along *different* styles ; and administrator should be able to manage them easily.

Fig.5.7 shows an instance of that need, in the current example we followed. A headline called "Featured Artist of the Day" aims to feature in an extensive manner a particular artist (i.e. displaying all possible available information on her/him), while *also* featuring, at the bottom the page (and prefaced by such phrase as 'Other artists of interest...') two other artists, but in a far more minimalistic fashion.

Each artist to display will be treated as an object, but each will have a *different attribute* : for instance the attribute "HUGE showcase" for the main one, and "Small showcase left", "Small showcase right" for the two others. Thus, to implement it : a leaf frame "3-artist frame" will be defined as follows, handling 3 objects and 3 attributes.

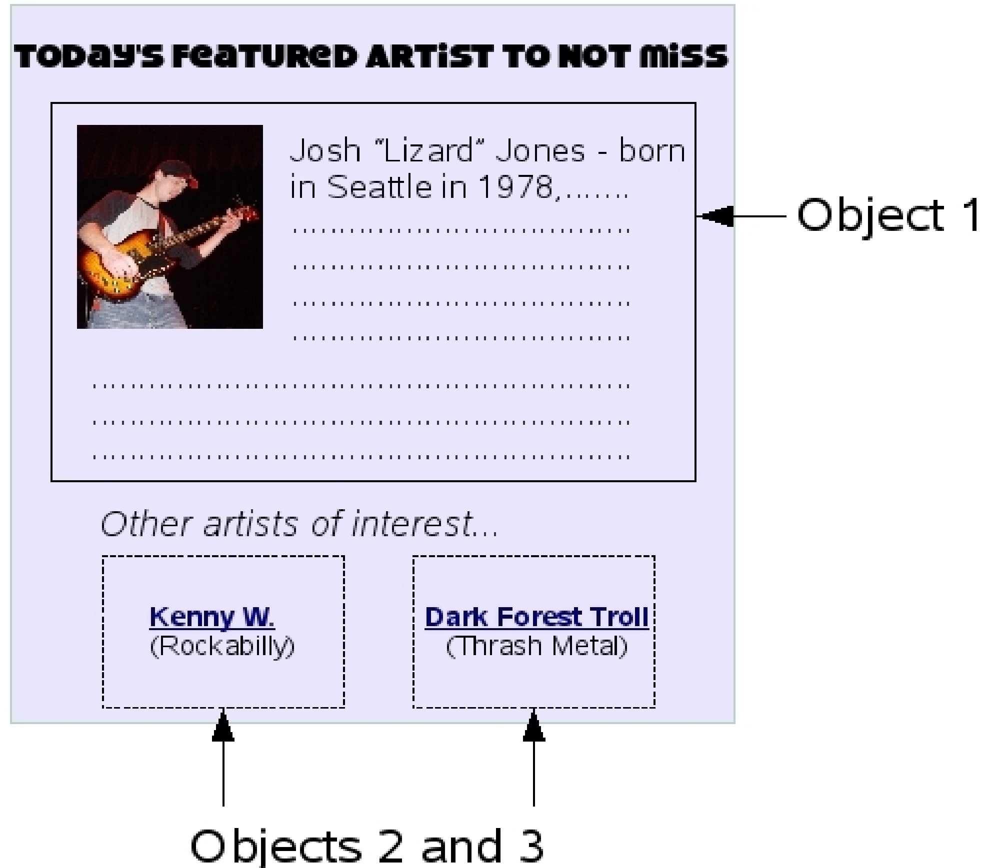


Figure 5.7: A page design that calls for multiple objects to be featured.

5.5.1 A leaf frame that handles multiple objects

In Code 13, it can be seen that a single call to system function `PackageObjectView.phs` was made, with a single family of arguments (`POObjectType`, `POObjectId`,...).

Now, in order to display the multiple objects that we need, it is no simpler than to call the same function, but as many times as we need it (3 times here), and each time with a new set of variables : Code 19 then shows the result. It can be seen that variables are named `PO1{...}`, `POStyle1{...}`, `PO2{...}`, `POStyle2{...}`, etc. for each to be unique.

The contents must feature the page's `HTML` formatting in advance ; for instance, here, the `TABLE` formatting to display the 'lesser' artists one besides the other.

As for the frame's variables, they will thus be defined as shown in the same Code, at the bottom.

Code 19 A leaf frame's contents and variables to handle packages, for 3 objects :

Headline models / Modify / 3-artist frame :

```
<!- Main Artist ->
«INCLUDE PackageObjectView.phs;_PackageId=«PO1PackageId»;
_ObjectTypeId=«PO1ObjectType»;_ObjectId=«PO1ObjectId»;
_ObjectAttrType=«POStyle1AttrType»;_ObjectAttrValue=«POStyle1AttrValue»»

<TABLE>
 <TR>
  <TD> <!- Artist on the bottom, left ->
   «INCLUDE PackageObjectView.phs;_PackageId=«PO2PackageId»;
   _ObjectTypeId=«PO2ObjectType»;_ObjectId=«PO2ObjectId»;
   _ObjectAttrType=«POStyle2AttrType»;_ObjectAttrValue=«POStyle2AttrValue»»
  </TD>
  <TD> <!- Artist on the bottom, right ->
   «INCLUDE PackageObjectView.phs;_PackageId=«PO3PackageId»;
   _ObjectTypeId=«PO3ObjectType»;_ObjectId=«PO3ObjectId»;
   _ObjectAttrType=«POStyle3AttrType»;_ObjectAttrValue=«POStyle3AttrValue»»
  </TD>
 <TR>
</TABLE>
```

Headline models / Variables / 3-artist frame :

```
PO1;Main Artist;14
POStyle1;Display style 1;15

PO2;Artist, Left;14
POStyle2;Display style 2;15

PO3;Artist, Right;14
POStyle3;Display style 3;15
```

From the unicity of the Object and Attribute scripts, a new package could be defined so as to keep various functionalities in the same publisher. Indeed, as a `_PackageId` variable is used for each `PackageObjectView.phs` call (Code 19), ***a single frame can use different packages****.*
As a matter of fact, as Fig.5.5 well showed, the selection of an object always starts by a list of available packages. .

5.5.2 Packages : defining more of them

In the current example, a "Festivals" package had been previously defined, with its Object and Attribute scripts. Now, for the needs of the page "Featured Artist of the Day" we aim to build, we need to define new scripts. In order to not delete the previous ones, **it is possible to define a new package (for instance called 'Artist News')** where we will define our new scripts.

In the new `PackageInstall.phs` script, no `SQL` table would have to be created if our new needs are covered by the previously-created tables (`OPS_ARTISTS`, for instance) : **any table defined in a package context is accessible by any package**.

Nonetheless, a new package is a great occasion to expand available information : for instance, this new package could entail the creation of a `OPS_ARTISTS_MORE`, aimed at containing far more detailed information about artists (biographies, member(s), instruments played,...).

5.5.3 Object and attribute scripts for this case (guidelines)

The Object scripts (not `PackageView.phs`), in the current example, would be aimed at artist selection only. It could be then quite practical to define object types as musical styles and thus offer an easier, quicker way to select which artists to be featured on the headline :

`PackageObjTypeList.phs :`

```
    (...)
    <TD><A HREF="javascript:selectObjectType(200,«idMusicalStyle»)">
«azMusicalStyle»</A></TD>
    (...)
```

`PackageObjectList.phs :`

```
    (...)
    <TD><A HREF="javascript:selectObject(200,«idMusicalStyle»,«idArtist»)">
«azArtist»</A></TD>
    (...)
```

For attributes, there is no need to define several types of them : one default type is enough, holding three attributes, one for each display style :

`PackageAttTypeList.phs :`

```
<TD><A HREF="javascript:javascript:AttributeType(1)">Default</A></TD>
```

`PackageAttValList.phs :`

```
«IF @toi(iAttrType)==1»
 <TABLE>
  <TR><TD><A HREF="javascript:AttributeValue(1)"></A>HUGE showcase</TD></TR>
  <TR><TD><A HREF="javascript:AttributeValue(2)"></A>Small showcase left</TD></TR>
  <TR><TD><A HREF="javascript:AttributeValue(3)"></A>Small showcase right</TD></TR>
 </TABLE>
«ENDIF»
```

As for the scripts used for displaying names, `PackageObjectName.phs` and `PackageAttrName.phs` are built in accordance with `PackageObjectList.phs` and `PackageAttValList.phs`.

The result of those, on the *Headline Modify Page* is featured at Fig.5.8.

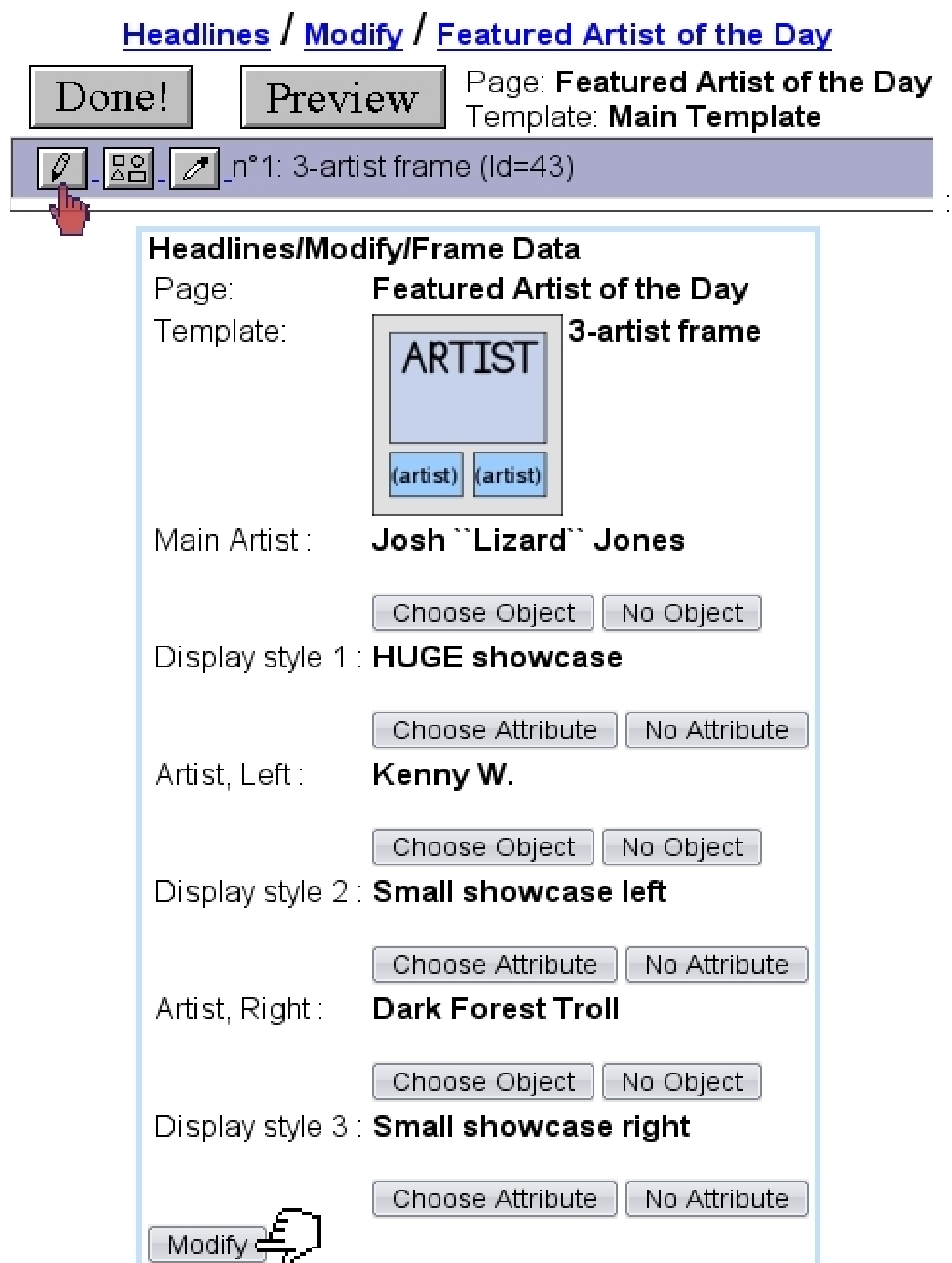


Figure 5.8: The 3-object frame after all relevant choices of objects and attributes have been made.

5.5.3.1 The `PackageObjectView.phs` script for this example

Finally, the `PackageObjectView.phs` script for which a sketch is shown at Code 20, will generate the headline ; it will be executed three times (i.e., as many times as there are `INCLUDE`'s in the frame's code) by the system upon click of the Modify button. For each time, the system will set the right variables for the current object type/object and the attribute type/attribute value.

Code 20 An example for `PackageObjectView.phs` for the 'Artist News' package :

```
(...)

«IF @toi(_ObjectAttrValue)==1» «*Main Artist Display*»

 Today's featured Artist to NOT miss
  «*Fetch artist information using _ObjectId *»
 (...)

«ELSEIF @toi(_ObjectAttrValue)==2 || @toi(_ObjectAttrValue)==2» «*Display for
the two other artists*»

 «*Fetch artist information using _ObjectId *»
 (...)
 <A HREF="#" onClick="onViewArtistDetails(«=@toi(_ObjectId)»)">«_Name»</A><BR>
 («_MusicalStyle»)

«ENDIF»
```

More precisely, at the first turn, (mentioning only the variables of interest) the object value will correspond to Josh "Lizard" Jones' identifier and an attribute value `_ObjectAttrValue` of 1, thus producing the 'Today's featured Artist to NOT miss' part. On the next two turns, identifiers will correspond to the two other artists, and the attribute values of 2 and 3 will produce the smaller display.

It is to note that the table disposition of those two last objects was already done when frame was defined (Code 19), so there is no need to do it here.