PageBlocks Framework For Lasso: Introduction, Security Aspects (Data Validation, AJAX, Flash), Localisation, Handling of CSS and JavaScript

By Nikolaj de Fine Licht - <www.musicmedia.dk>
PageBlocks: <www.pageblocks.org>

1. Introduction

1.0 Scope

The scope of this presentation is to shortly introduce the Lasso-framework PageBlocks and then emphasize on how PageBlocks handles some typical challenges for developers: security and localisation. It also touches on the use of CSS- and Javascript-files.

The presentation doesn't pretend to be exhaustive, neither on PageBlocks in general, nor on the specific areas covered. Its mainly thought of as an "appetizer", an illustration of "how things can be done" if you build web applications with PageBlocks framework. As a consequence, almost all code samples only work in a PageBlocks environment.

1.1 Who Is This Aimed At?

Its my hope that anybody can gather some useful information from this presentation!

Chapter 2 is a short, general introduction to PageBlocks and shows how to download and install it. It should be accessible for anybody.

Chapters 3-5 are meant as demonstrations of how PageBlocks handles some typical challenges in web project development. Those chapters are probably not too accesible for beginners, and the techniques shown are generally not applicable right away outside of a PageBlocks context.

If you are some sort of "on-you-way" into Lasso, then changes are you are approximately where I was a couple of years ago when I started looking seriously at PageBlocks.
In such case I'll say: yes, the learning curve can be somewhat steep (it was for me). But anybody can do what he/she really wants to do, so don't give in! And know that its not either/or! You can set up a PageBlocks project and use it's architecture and some of it's features and tools and continue to do straight forward Lasso programming for the rest. Don't think you need to know the whole thing by heart before writing the first bit of PageBlocks code - I still, two years after, don't know and don't use all aspects of PageBlocks.

If you are new to Lasso but not to web programming, then defenitely my hope you can draw similarities to the knowledge you have and gather some insight in how aspects of beyond-basics web projects are treated with the combination Lasso/PageBlocks.

1.2 Aknowledgements

Anything "PageBlocks" is by nature in deep debt to it's creator, Greg Willits. Greg developed PageBlocks over some years (formerly FrameWork Pro) as a tool for his own projects, he wrote an extensive Developer Guide for it, and he set up a website and generously put it all at disposal there for the Lasso community and anybody else, free of any charge. When Greg signed out from the Lasso community in November last year, the PageBlocks project was taken over by me.

I'm also in debt to the Lasso Talk community and the fine people there, many of which certainly are more skilled developers than I, always willing to share their findings.

1.3 OK - So What IS A Framework?

The term "framework" meant nothing to me when I started diving into PageBlocks. I believe a word that gets close is tool - a framework is a tool for easier and/or better development of a website project.

But why use a tool? - There is Lasso, that seems to be "a tool", and if you code up your stuff with Lasso, then - given there aren't too many "glitches" - you end up with a working website.

Thats is true, but there are many standard issues and typical routines when making a website project that it soon gets annoying to solve over and over again. At this point you begin re-using code snippets, you start putting common code in a folder in the website root and include bits of it with `include();`, you maybe start making custom tags (functions) and custom types (methods) - all stuff you can use to solve some of these common tasks in each new project. You have taken your first steps in direction of what eventually could become a framework!

A framework is a standardised set of tools to handle standard issues and typical routines when programming - in our case a web application. A framework takes the language in question and makes ready-to-use building blocks with it. Rails is a framework for the Ruby programming language. Our framework, PageBlocks, takes Lasso and offers at set of structuring schemes, custom tags and -types, and routines to do what you otherwise could end up doing in a verbose, unpratical, not re-usable and - thus - time consuming and not very readable way.

An Example

If you want to look up a single record in a database and assign the field values to a corresponding set of variables you could do something like the following in Lasso:

```
inline(
   -database = 'myDb',
   -table    = 'myTable',
   -username = 'myUsername',
   -password = 'myPassword',
   -sql      = "SELECT * FROM mytable WHERE id = '" + $idValue + "'"
);
   // here you would add some code that assigns the field values to a set of vars for later
   // retrieval, could be:
   local('mapOfVars' = map);
   iterate(field_names, local('fieldName'));
      #mapOfVars->(insert(#fieldName = field(#fieldName)));
   /iterate;
/inline;
```

The same thing in PageBlocks would look like this:

```
var('myDataObject' = fwp_recordData('mytable'));
$myDataObject->(select(
   -select   = '*',
   -keyval   = $idValue,
   -withMakeVars
));
```

Done! You will have each single var value ready at hand, and you will have abstracted var names - that is, var names that differ from the field names. To continue the terminology from above: this is a little tool you can use to do a very common task in a web application.
Of course for this simple code to work there is a number of things that must be in place first; in this case, as the most important, there is a config file that must be set up once. It contains all the information about the table in use, including the field names and the corresponding (differing) var names and more.

1.4 "OK, But What's In It For Me If I Choose To Build With A Framework Like PageBlocks?"

As we saw in the example above you could not only do a common task with less code than otherwise necessary, you also got vars declared that use names differing from the corresponding field names. To explain why the this is desireable we have to touch an issue that we will deal with again later: security. The short explanation is: it is safer if you can't deduct the database field names from the var names.

This illustrates a very important point when thinking about using a framework like PageBlocks as opposed to knitting your code up from A-to-Z: not only can you do a lot of things quicker, you also get a lot of added value on top, you simply produce better quality solutions! As we schall see later in this paper you get, among others, built-in handling of security matters and of localisation. If you furthermore care about coding style, about separation of logic and presentation, about

emphasis on Object Oriented Programming, on readability for other developers than yourself etc., then you have a number of other good reasons for choosing to build with a framework like PageBlocks.

1.5 "I Guess PageBlocks Makes A Lot Of Assumptions Then..."

As probably any convenient framework, PageBlocks makes a number of assumptions in order to be time-saving and steamlined to work with. I don't have the background to compare PageBlocks with a variety of other frameworks, all I can say is that the assumptions are there, but they are very easily configurable in case you want to change them, and doing so won't feel as going against the framework.

I have sometimes come across assumptions that seemed either wrong to me or were simply missing. In such case its important to remember that PageBlocks is not a finished project, it can - and should - continue to be improved.

1.6 PageBlocks Is Overkill For The Little 7-Pages Website

I would say that PageBlocks pays for projects with at least 5-7 2nd-level areas, or some sort of URL abstraction, and/or some sort of access restricted area - a CMS or some other administrative area. It pays a lot if you need localisation at some level. I have happened to develop mainly administrative systems (Internet-based Intranets and CMS'es), and here PageBlocks is a fantastic tool.

1.7 PageBlocks Is Open Source

It is important to stress that PageBlocks is an Open Source project! It is so in the sense that all code is freely accessible and may be used for whatever purpose, also outside of PageBlocks context; it is also so in the sense that anybody can contribute to PageBlocks and have their improvements incorporated into the official release. But it is NOT so, however, in the sense of a public repository or anything else fancy sharing-wise set up. Not because it wouldn't be neat to have, but because of my lack of time and knowledge!

1.8 PageBlocks And The OS'es

PageBlocks is designed to be able to run on OS X, Linux and Windows; it has, however, not been tested on platforms other than OS X. PageBlocks still awaits a knowledgeable Linux- and/or Windows developer to get enthusiastic and dive in, test and apply tweaks possibly necessary to run on those OS'es.

1.9 PageBlocks And The Data Sources

PageBlocks is designed to be able to communicate with any datasource through a data source abstraction layer. Again, PageBlocks hasn't for some time been tested with other data sources than MySQL, simple due to the same facts as mentioned above.

2. Introduction to PageBlocks

2.0 You Want To Install PageBlocks Both Locally And On The Remote Web Server

You must have PageBlocks up and running both on your local development machine and on the web server where you eventually will publish your solution. For setting up a Lasso-enabled web server on a local development machine there are different ways to go depending on your OS and it's version. Mac OS X-instructions are provided in the Read Me accompanying the PageBlocks download package. You can also find such instructions elsewhere on the Internet. PageBlocks has a feature named Deployment Modes which essentially enables you to run exactly the same version of your project both locally and remotely but still have different names for databases, different levels of debug-info etc.

2.1 Then, What IS PageBlocks Practically Speaking?

Physically PageBlocks consists of a number of files and folders that have to be installed, some in the LassoSite in question and some in the webroot folder; and it consists of a number of tables that must be installed into a MySQL database
Configuration-wise PageBlocks is a set of configurations that has to be made in the LassoSite in question and the Apache webserving configuration file.
Structure-wise PageBlocks is a way of organising the files and folders that make up your project in a certain way which allows the framework to make a number of assumptions about where to find things when serving pages

Programming-wise PageBlocks is a way of coding that, to put it rough, is more about working with the PageBlocks tags and types than writing straight forward Lasso code.

(It would be convenient to be able to build an installer that installs and pre-configures PageBlocks, but it seems Lasso SiteAdmin doesn't currently have an API that allows for this.)

2.2 Got Lost Already?

OK, maybe above standing was enough to turn you off. The Apache webserving configuration file - huh? PageBlocks tags and types - what is a type?
There IS quite a few things to do before even starting to do anything "for real" with PageBlocks, I agree, but it CAN be done :) Below we will look closer at this. But before that I have to point out:

2.3 You Need A Collaborating Hosting Service Provider (HSP)!

Why? Because you need a few things installed and configured that aren't necessarily part of a typical, Lasso-enabled hosting plan. If you host your own solutions this isn't a problem of course, but if - like me - you don't, then you need this collaborative attitude. Hosting providers I know of that do setup Lasso/PageBlocks-enabled accounts without any extra charge are:

Point In Space <www.pointinspace.com>. Point In Space is the sponsor of the hosting of <www.pageblocks.org>
Flagship Hosting <www.flagshiphosting.com>
Falcon Internet <www.falconinternet.net>

2.4 These Are The Steps

1. Open a Lasso/MySQL-account hosting with your HSP.

2. Currently you will need to ask your HSP to a) put the LassoStartup folder into the account root folder and to b) create a symlink to this folder in it's original place (inside the site folder of the Lasso site in question), so that now you have a folder in your web server root named 'LassoStartup', its empty. By the time you read this, this step may have been eliminated.

3. Download the PageBlocks package from <www.pageblocks.org> and unpack the zipped file. You will now have a folder named 'pageblocks_5XX' (where 5XX represents the version number) wherever you unzipped it to.

4. In this folder grab the file 'error.lasso' in the folder named 'Put_in_LassoAdmin' and mail this file to your HSP, asking to put it in the folder named 'LassoAdmin' inside the site folder of the Lasso site in question.

5. Grab everything in the folder named 'Put_in_LassoStartup' and FTP this to the folder 'LassoStartup' we got in step 1. above. Restart the site. Again, by the time you read this, this step may have been eliminated.

6. Now follow the instructions in the file '__READ_ME_FIRST.txt'. Again, the first time you look at these instructions you may experience an "Oh-My-God"-reaction. Don't let this turn you off! Take a cup of tea or coffee and just follow the indications step by step. Forgetting a few things or running into problems in the beginning is part of the game, don't hesitate to direct any questions to me!

After having setup a PageBlocks-enabled account a couple of times it is actually possible to set up a new project ready for development in less than an hour!

2.5 Now You Have - err... Nothing!

After having followed the instructions mentioned above you will have installed files, you will have configured Lasso SiteAdmin, you will have installed a MySQL database and tables and configured the Apache webserver and a few other things.
What you now have is NOT a full CMS, NOR do you have a sample blog or a sample gallery or a sample something else fancy, you essentially have - nothing! Which isn't correct, because the PageBlocks package does contains a sample website with corresponding database tables, which is basically a copy of <www.pageblocks.org>, as well as a 'blank' project named "pbStart" which despite being 'blank' still includes core PageBlocks functionality such as user management (for access restricted areas) and multi-language string management, which gets us close to a simple CMS (see 5.4 below).

You will always want to install at least the 'blank' project, because this way you have the whole PageBlocks structure in place, including a number of folders and sub-folders, necessary for PageBlocks to find files in the places where it expects to find them.

But it is important to understand, that PageBlocks is a toolset, it doesn't do much until you start using those tools to build a project with.

2.6 What Is Next Then?

At this point you would start building your project, wether tweaking existing pages in one of the sample websites or simply adding new stuff to the 'blank' project. Its beyond the scope of this presentation to guide a way into building a new project. The PageBlocks download package includes a quite extensive Developer Guilde and a complete reference database, also found on <www.pageblocks.com>, for all the tags and types that make up the PageBlocks API (Application Programmar Interface).

3. Security In PageBlocks: Data Validation

3.0 Data Validation As Security Means

Security is an ever increasing concern when making web solutions. While a lot of responsibility for the securing of a server lays with the HSP, the developer is responsible for what the web application does in terms of opening holes into data and areas not meant for public access.
The problem is that the great power of the Internet and the great impact of good websites lay in their interactive nature, but this interactivity is obtained in your web application by the same means that opens almost all it's potential security risks.

While security is an area which naturally is constantly shifting, then there are basic rules that one can go with in order to get a very long way. PageBlocks is build up around a set of security principles which you find outlined here: <www.pageblocks.org/refc/refc_security>. Building with the PageBlocks framework and with these principles in mind increases chances considerably that your web application will stand well against attacks!

One very important principle is to setup not one but more security layers in order to prevent or decrease impact if one layer is defeated. And one very important such layer is data validation, that is, the checking of all incoming data for maliciously crafted content.

3.1 Data Validation As "Data Cleaner"

Data validation also has another aspect: to check if content conforms to expected formats and content types. We don't want anything else than numbers and maybe a +-sign in phone number fields, so don't let the user get away with typing letters into phone number fields.

3.2 Client Side And Server Side Validation

Data validation can take place directly in the browser (typically by JavaScript) before anything is sent to the server, or it can be taken care of on the server by the code receiving the data. Or both. It is important to know that from a security point of view, only server side validation can prevent intruders, while client side validation can be seen as mainly an interface enhancer.
The validation tools included with PageBlocks are exclusively for server side validation. PageBlocks doesn't currently offer any automisation of client side validation rutines, but this is on the list for future development!
For further introduction to data validation in PageBlocks see <www.pageblocks.org/ftrs/api_validate>.

3.3 Data Validation In PageBlocks

In PageBlocks, validation has been centralized into a single unit (a custom type) which is then used by the database action unit to automate validation of form- and other inputs. The validation unit can also be used to validate inputs outside of database action situations. Essential to the workings is that:

input name <-> database/table field name <-> data type <-> validation instruction

are tied together in a central config file, one config file for each database table (data model). The validation instructions are in the form of validation codes, that refer to a set of built-in validation instructions and/or custom made validation

instructions. If you have used AutoValidate you will know this princip.
This enables the database action unit to:

1. find the field names in the given database/table
2. update values from or read values into the corresponding variable/input names
3. underway invoke the validation unit using the validation instruction given for each data

Here is an excerpt from such a config file showing some table fields and their corresponding input names and validation instructions - the {{ and }} are used by the PageBlocks parser to extract the config data:

```
{{tableModel___
#inputName       fieldName          dataType  validation codes
#-----------     ---------------    ------    ----------------------
m_ctMaTle        cnctMainTitle      string    maxlen=30, isAlphaNumeric
m_ctMaNmeF       cnctMainNameF      string    maxlen=20, -extn
m_ctMaNmeL       cnctMainNameL      string    isRequired, maxlen=20, -extn
m_ctMaIsStd      cnctIsStandard     string    maxlen=1
m_ctMaBirthday   cnctMainBirthday   string    maxlen=10, isDate=euroDate
}}
```

If we take the first line then it instructs that:
- the input name (the input field name in forms, Lasso var name) is 'm_ctMaTle'
- the corresponding field name in the database/table is 'cnctMainTitle'
- the data type is 'string'
- the validation codes are 'maxlen=30' and 'isAlphaNumeric' which means that the string length can't be more than 30 characters and only alphanumeric characters are accepted.

'isAlphaNumeric', 'maxlen=XX' and 'isRequired' are examples of PageBlocks built-in validation codes. For a complete list of built-in validation codes please see the validation section in the Developer Guide that comes included with the PageBlocks package.

The second line shows, among others, the validation code '-extn'. This code refers to a custom validation instruction written by the developer (it is suggested to make all custom validation codes start with a hyphen to easily distinguish them from the built-in ones). In this case an instruction that allows a set of extended European Unicode characters and a few symbols.

3.4 Before Going On: Separation Of Logic And Display

Before going on with the implemenation of the validation mechanism there is a basic principle in PageBlocks to point out: all code that will eventually create "a page in the browser" is split in two: logic and display. Logic code is the code that for examples issues a database call and retrieves the values, display code is the code that contains or generates HTML and Lasso display code, or in other words, the code that eventuelly will output what we see in the browser.
The principle of "PageBlocks", each page block consisting of a logic block and a display block, that get inserted into a template, is illustrated on the front page of <www.pageblocks.org>.

3.5 Implementation: The Logic

Lets say we are dealing with a simple database update action with values from form fields. Using the table model configs from 3.3 above, the logic in the response file would look like this in PageBlocks:

```
var('myDataObject' = fwp_recordData('mytable'));
$myDataObject->(update(
  -keyfld   = 'rcrdNo',
  -keyval   = $idValue,
  -inputs   = 'm_ctMaTle, m_ctMaNmeF, m_ctMaNmeL, m_ctMaIsStd, m_ctMaBirthday'
));
```

The neat thing here is, that the ->update member tag (function) of the fwp_recordData object automatically invokes the validation mechanism. The same goes for all other member tags that adds or update values. If one or more validation problems occurs the database action will not take place, instead validation error messages are generated and inserted into the $fw_validator instance of the fwp_validator object.

If you want to trap for no validation errors already in the logic block you would add the following after the update code above:

```
if(!$fw_validator->('errorMsgs')->size);
  redirect_url('/contacts/listing?fw_s=' + $fw_s);
/if;
```

If there were no validation errors the redirect will take place, otherwise we will stay with the file specified in the form submit action. `$fw_s` is the standard session var in PageBlocks.

3.6 Implementation: The Display

The display file accompanying the logic above would contain the form. Something like the following (maxlength and tabindex are left out):

```
<fieldset>
  <legend>Personal Data</legend>
  <form action="[response_filepath]?fw_s=[$fw_s]" method="post" name="form01" accept-charset="utf-8">
    <input type="hidden" name="idValue" id="idValue" value="[var('idValue')]" />
    <label for="ctMaTle">Title:</label>
    <input type="text" class="size2" name="m_ctMaTle" id="ctMaTle" value="[var('m_ctMaTle')]" />
    <label for="ctMaNmeF">First Name:</label>
    <input type="text" class="size2" name="m_ctMaNmeF" id="ctMaNmeF" value="[var('m_ctMaNmeF')]" />
    <label for="ctMaNmeL">Last Name:</label>
    <input type="text" class="size2" name="m_ctMaNmeL" id="ctMaNmeL" value="[var('m_ctMaNmeL')]" />
    ...etc
  </form>
</fieldset>
```

If the submition causes validation errors we want to display messages about these problems. One way could be to add the following above each input field (shown only for the Last Name input field):

```
[$fw_validator->('errorMsgs')->find('m_ctMaNmeF') ? $fw_validator->('errorMsgs')->find('m_ctMaNmeF')-
>first->value]
<input type="text" class="size2" name="m_ctMaNmeF" id="ctMaNmeF" value="[var('m_ctMaNmeF')]" />
```

But that soon becomes tedious if there are many fields (and many forms), so maybe we want to display all validation error messages in one, centralised feedback area somewhere on the page. We can acchieve that this way:

```
<?lassoscript
  if($fw_validator->('errorMsgs'));
    '<p>';
    iterate($fw_validator->('errorMsgs'), local('i'));
      (loop_count) > 1 ? '<p>';
      #i->second + '</p>';
    /iterate;
  /if;
?>
```

Pretty simple! The validation error messages to display all reside in one single config file, where the fwp_validator object pulls them from.

But now we have another problem! A typical error message is "This field can not be left empty". It makes sense when you display the message next to the field causing the problem, but not when we display the messages in one, centralised feedback area as suggested above. We need each message to display the field label it refers to.
To obtain this we add information to the validation codes from the table config file shown in 3.3., the validation codes taken separately would now look like this:

```
#... validation codes
#... ----------------------
...  maxlen=30, isAlphaNumeric, haslabel=Title
...  maxlen=20, -extn, haslabel=First Name
```

```
...  isRequired, maxlen=20, -extn, haslabel=Last Name
...  maxlen=1
...  maxlen=10, isDate=euroDate, haslabel=Birthday
```

Because of the 'haslabel'-code, the validation error message from above, when referring to the label "Last Name", would now be "The field Last Name can not be left empty" where 'Last Name', furthermore, would be marked up with a class in a span tag `<span class="inputerrfldnm">Last Name</span>`.

But maybe we still want to improve the interface, maybe we also want to highlight the labels of the fields that caused problems. This can be done by doing the following (shown only for the Last Name input field label):

```
<label for="ctMaNmeL"[$fw_validator->('errorMsgs') ? $fw_validator->('errorMsgs') >> 'm_ctMaNmeL' ? '
class="inputerrfldnm"']>Last Name:</label>
```

This code will add the class `inputerrfldnm` to the label tag enabling you to highlight it with CSS.

But there is still more to the display of field labels - see 5.8 below!

3.7 Validation Outside Of Database Actions - Database Actions Without Validation

Sometimes you need to perform validation without database actions, that is, without `fwp_recordData->(add())` or `fwp_recordData->(update())` or similar involved. This can be accomplished with the following:

```
var('myDataObject' = fwp_recordData('mytable'));
$myDataObject->validateInputs('m_ctMaNmeF, m_ctMaNmeL');
```

`validateInputs()` accepts the keyword `-useGet` which instructs it to validate on GET-params rather than the default POST-params.

And, again, you can check for validation errors by adding this right after:

```
if(!$fw_validator->('errorMsgs'));
  // there was no error so we continue
else;
  // do something else
/if;
```

If you need to do a database action that normally invokes the validation mechanism but want to avoid validation (for example because it was already done by code similar to the one shown above or by the code shown in 3.8 below), you simply add the `-withoutValidate` keyword to the action:

```
var('myDataObject' = fwp_recordData('mytable'));
var('newIdValue'   = fwpStr_randomID(8)); // PageBlocks random string id generator
$myDataObject->(add(
  -keyval   = $newIdValue,
  -inputs   = 'm_ctMaNmeF, m_ctMaNmeL'
  -withoutValidate
));
```

3.8 Validation Of Data From And To Web Services And Of Cookie Data

PageBlocks offers ways to validate data that is NOT part of the pre-configured data models. It could be validation of data from GET or POST params from a web service, including validation of data from containers such as cookies, XML or JSON; or it could be validation of data before being sent to a web service.

Incoming data could be response to a request in the form of the following JSON string:

```
{"result": {"shape": "circle", "color": "cc9900"}, "error": "", "id": "1234"}
```

We extract the result and insert into a var:

```
var('jsonData' = map);
$jsonData = decode_json('{"result": {"shape": "circle", "color": "cc9900"}, "error": "", "id":
"1234"}')->find('result');
//-> map: (shape)=(circle), (color)=(cc9900)
```

Next, we would have a -formSpec of validation specifications that might look like this:

```
var('jsonDataValidations' = map);
$jsonDataValidations      = map(
  'shape'= 'isRequired, isAlpha, hasMaxLength=24',
  'color'= 'isRequired, isHTMLColor'
);
```

And before using the JSON data we would perform a validation and act accordingly like this:

```
var('dataIsInvalid'  = false);
$dataIsInvalid   = $fw_validator->(validate(
  -usingPairs    = $jsonData,
  -formSpec      = $jsonDataValidations
));
```

```
if($dataIsInvalid);
  // data didn't pass so do some error reporting
  // you can manually insert validation errors into $fw_validator like this sample:
  // $fw_validator->(insertErrorMsg(-valcode=9999, -input='jsonData'));
else;
  // proceed to processing
```

Exactly the same procedure could be followed to validate data from a cookie, only would the process of extracting the data in the first place be different.

Similar, say you have a small form that upon submition sends a request to a web service and you want to validate the values before sending them off. It could look like this:

```
var('inputValidations'  = map);
var('inputsAreInvalid'  = false);
$searchValidations = map(
  'searchString'   = 'isRequired, hasMaxLength=128',
  'maxResults      = 'isPositiveInteger'
);
$inputsAreInvalid  = $fw_validator->(validate(
  -usingPOSTForm,
  -formSpec   = $inputValidations
));
if(!$inputsAreInvalid);
  // the following is in princip but untested
  json_rpccall(
    -method= 'myMethod',
    -params= array($searchString,$maxResults),
    -id    = 1234,
    -host  = 'http://myhost/services/lookup.jsonrpc'
  );
/if;
```

4. Security: Securing AJAX And Flash

4.0 Any File That Receives And Handles Data Must Be Secured

With the increasing use of AJAX-techniques and other extended JavaScript-usages in web applications a whole new attack target area has seen daylight. AJAXified pages typically send requests to helper files on the server, and its a piece of cake for hackers to bypass the original AJAX-calls and issue maliciously crafted requests to those helper files.
Also Flash can be used to communicate with the server, and while it can be a little trickier to figure out the locations of the helper files for a Flash movie it can certainly be done.

I will here look shortly at some securing techniques as applicable in a PageBlocks context. This chapter is by no means exhausting, instead the reader should look into other contributions in the LDC manual focusing entirely on security.
In order to profit from the following you must have a fairly good understanding of how AJAX-calls and -responses and related work.

4.1 Data Validation In AJAX Helper Files

The good news is that the whole validation mechanism described in the previous chapter is at hand for securing AJAX helper files in PageBlocks!

In order to show how this is accomplished I have to shortly describe how a request for a "normal" web page is handled in PageBlocks: PageBlocks works entirely with extensionless URLs. When a request comes in, it passes through config files that will:
- check if the file is listed as existing
- initiate a number of Lasso vars, instantiate a number of objects etc.
- add general template code such as <head>- and <meta>-tags to the HTTP response
- add specific template code for the page requested to the HTTP response and run the code in the blocks
- and more...

But in the case of AJAX helper files you don't want the overhead of general or specific template code. It IS possible to bypass the process entirely by for example calling a .lasso-file which isn't listed as existing in the config files (this is, by the way, the reason you can combine PageBlocks architecure with standard .lasso files and pages). But if you do this, then you won't have the PageBlocks tools at hand, unless you add the following to the very top of your .lasso-file:

```
<?lassoscript
  fwpPage_init;
  // here your code
```

In case of an authenticated page you must be sure the AJAX-request includes the PageBlocks session var, and you add this to the fwpPage_init-tag:

```
<?lassoscript
  fwpPage_init(
    -fw_pgAuthRequired = true, // means authentication is required for this access
    -fw_pgPrivilege    = 'fw_user->prof.loginValid' // sets the privilege needed for access
  );
  // here your code
```

But an easier and more conform way is to use the special AJAX- and Service-templates in PageBlocks. They are only "templates" in the sense that they standardize the request processing, they don't add anything to the response. So by using an AJAX-template you can continue to use the whole PageBlocks toolset, the extensionless URLs, the logic/display separation etc. without any downsides. Here is how:

In each subfolder (in PageBlocks terminology a 'module') there is a config file named '_pageConfig.lgc' which, among others, lists the pages available to be requested in that subfolder (module). Each existing page is listed and has some configuration attached, and in that list you configure your AJAX helper file as follows, assuming it's name is 'ajaxHelper':

```
select($fw_requestPage->('name')); // select list of available pages
  case('ajaxHelper');
    $fw_pgTemplate   = fw_kPageIsAjaxHandler;
    $fw_pgPrivilege  = 'fw_user->prvlg.loginValid';
  case(...);
  etc...
/select;
```

The constant 'fw_kPageIsAjaxHandler' used as value for $fw_pgTemplate instructs the PageBlocks template builder unit to proceed without adding anything.
Your AJAX helper file will physically consist of two files, a logic file and a display file. This is, as already mentioned, a basic concept in PageBlocks architecture. If your AJAX helper file doesn't return anything for display, then you just leave

the display file empty (but it must be present). So in our case you would have these two files on disk in the current subfolder (modul):

```
ajaxHelper_ajax.lgc
ajaxHelper_ajax.dsp
```

You will notice the endings '_ajax' which instructs PageBlocks of the nature of these files, and you will notice the file extensions .lgc (logic) and .dsp (display) - these extensions are used throughout all PageBlocks to distinguish the task of the page blocks.

Inside the `ajaxHelper_ajax.lgc` file you can now do validation on the data passed by the AJAX-call exactly the same way we did in 3.5 above:

```
var('myDataObject' = fwp_recordData('mytable'));
$myDataObject->(update(
  -inputs= 'input1, input2, ...etc,
  -keyfld= 'rcrdNo',
  -keyval= $id,
  -useGet
));
```

However, notice the `-useGet` keyword. This instructs the validation mechanism to validate GET-params instead of the default POST-params. Most AJAX-calls use the GET-method as default method, and you wouldn't get anything validated if you didn't add this instruction. As long as the param names you pass in your AJAX-call to the helper file are the same as the input names from the data model (see 3.3) in question, then the param values will be processed.

Again, for AJAX requests in an access restricted area you must make sure the PageBlocks session var is sent along with the request.

To resolve the problem of getting possible validation error messages back into the mother page that issued the call the technique will vary a little depending on the AJAX-library you use. But basically, you can add this (same as in 3.6) in your AJAX display page:

```
<?lassoscript
  if($fw_validator->('errorMsgs'));
    '<p>';
    iterate($fw_validator->('errorMsgs'), local('i'));
      (loop_count) > 1 ? '<p>';
      #i->second + '</p>';
    /iterate;
  /if;
?>
```

If your AJAX-call script has indicated the id of the -tag that should receive any injected HTML, then the validation error messages will appear in that very -tag. The way validation errors are presented to the user can be refined depending on time and JavaScript-skills!

You may also need to make sure the values in the form fields in the mother page get updated with new values, you can do this by adding some JavaScript to your AJAX-display page:

```
<script type="text/javascript">
  getElementById('ctMaNmeL').value = '[var('m_ctMaNmeL')]';
  ...etc
</script>
```

4.2 Securing Flash File Calls

I will here take a look at one example: a Flash uploader. A Flash uploader can take params from the HTML-environment, it can take values hardcoded into itself, and it submits those together with the file upload to a helper file, which does the post-processing on the server. To prevent a hacker from bypassing the Flash uploader and send directly to the helper file,

both the Flash uploader and the helper file must be secured. While the securing of the helper file follows the principles indicated in 4.1 above, there are a few things that can be done to the Flash uploader itself:

Use the SWFObject JavaScript library to add the Flash movie, it can be found here <code.google.com/p/swfobject>. Not so much out of security concerns but because it is the only generally compatible, easily configurable way. With the SWFObject library loaded, the code for adding a Flash movie to your HTML looks something like this:

```
<div id="flashcontent">
  Error during load
</div>
<script type="text/javascript">
  var so = new SWFObject("illustrationUpload.swf","illustrationUpload","400","140","8","#f7f7ef");
  so.addVariable("fw_s", "[$fw_s]"); // session var
  so.addVariable("lang", "[$fw_client->('language')]"); // language var
  so.addVariable("maxsize", 2000000); // max allowed file size
  so.addVariable("filedest", "illustrationUpload.lasso"); // post processing file
  so.addVariable("fileexts", "*.jpg;*.gif"); // allowed extensions
  so.write("flashcontent");
</script>
```

This example is taken from an access restricted area (a CMS), in a public area you would hardcode the vars maxsize, filedest and fileexts into the movie.
Notice that we pass a value for the PageBlocks session manager, $fw_s, and a value for the PageBlocks client language. The former enables us to check for a valid session in the helper file, the latter enables us to display language dependent text strings in the Flash movie (more on localisation later).

In the ActionScript of the Flash movie, the session var is retrieved and later sent as a GET-param along with the request from the Flash movie to the post processing file (ActionScript):

```
//declare internal vars and assign values from environment
var fw_s:   String = fw_s;     // session id
var la:     String = lang;     // language var
var fl_pp:  String = filedest; // destination for post processing file (server script)
 ---
//build the request string
var my_lv:LoadVars = new LoadVars();
my_lv.fws = fw_s;
var getparams = (my_lv.toString());
url = fl_pp + "?" + getparams;
```

Now, as for the allowed file extensions (file types), the limitation through Flash is client-side only. In the post processing file there is no way to check the uploaded files based on MIME-type as Flash sends everything as application/octet-stream. As far as I know, the only way to be sure an uploaded file is really what it pretends to be is a) to make sure it wasn't uploaded bypassing the Flash uploader and b) to save it in the post-process file with the extension given.

To make sure a file was really sent to the post-processing file by the Flash uploader, and not by a hacker attempt, you can do a very simple trick: hardcode a var into the ActionScript that is sent along with the request to the helper file by the Flash uploader! Here is an example from a public area not using the PageBlocks session var, $fw_s, but a standard Lasso session. The session id is passed into the Flash uploader the same way as above, except we use a different var name (ActionScript):

```
//declare internal vars and assign values from environment
var sid:    String = sid;      // standard Lasso session id
var chk:    String = "mySecretCheckerValue"; // any alpha-num value
var fl_pp:  String = filedest; // destination for post processing file (server script)
```

And the request is now constructed like this:

```
var my_lv:LoadVars = new LoadVars();
my_lv.chk = chk;
```

```
my_lv.sid = sid;
var getparams = (my_lv.toString());
url = fl_pp + "?" + getparams;
```

If we make the helper file according to the first example shown in 4.1 above, the single .lasso-file, it could look like this:

```
<?lassoscript
  // start session
  client_getparams->(find('sid'))
    ?
    session_start(
      -id         = client_getparams->(find('sid'))->first->value,
      -name       = 'a',
      -expires    = 60,
      -cookieexpires= '',
      -useauto
    );
  // add some conditional here for session passed or not
  // if session passed, check if the secret checker is present and has the right value
  client_getparams->(find('chk'))
    ? var('chk' = client_getparams->(find('chk'))->first->value);
  if($chk == 'mySecretCheckerValue');
    // check passed so we start the PageBlocks page initialisation
    fwpPage_init;
    // here post processing code...
  /if;
?>
```

5. Localisation In PageBlocks

5.0 Localisation On Two Levels

The term *localisation* is here treated as the ability to display in more than one language. Localisation in terms of time zones, date formats, currency, etc. is left out.
We will look at localisation in PageBlocks on two levels: more than one website language, and more than one system language (as in an administrative system such as a CMS for example).

5.1 More Than One Website Language: Pulling The Right Data From The Data Source

There are several ways to acchive language localisation of a website, also within a PageBlocks context. This example uses language specific database tables, one for each language, for the dynamic content, and a mixture of those tables and of CSS for the navigational content. The two languages in this example are English and German, language codes 'en' and 'de'. All way through the code the language defaults to 'en' if 'de' isn't explicitly present.

A request for a page in PageBlocks passes the steps outlined in 4.1 above. To determine the language to display in a given webpage, the following code is added in the previously mentioned page config file, '_pageConfig.lgc', that loads before anything else in a given subfolder (module). It uses a standard Lasso session and a cookie for storing the visitors language preference:

```
// if a cookie is present use that language preference
// else try to set cookie with default language 'en'
if(client_cookies->(find('lang')));
  var('lang'  = cookie('lang'));
else;
  cookie_set(
    'lang'    = 'en',
    -domain   = $fw_requestPage->('domain'),
    -path     = '/',
    -expires  = 525600
```

```
  );
/if;

// start session to hold language preference
session_start(
  -name       = 'a',
  -expires    = 60,
  -cookieexpires= '',
  -useauto
);

// at this point, if there is no value for var lang yet or if its different from de
// then we define it with default value en
if((!var('lang')) || (var('lang') != 'de'));
  var('lang'    = 'en');
/if;

// for var lang we now have either the value retrieved from the cookie or the default value
// however, every page has a language switch link pointing to the page itself,
// so we have to check if that link has been clicked and update values accordingly
protect;
  if(client_getparams->find('lang'));
    if(client_getparams->find('lang')->first->value == 'de');
      $lang  = 'de';
      cookie_set(
        'lang'    = 'de',
        -domain   = $fw_requestPage->('domain'),
        -path     = '/',
        -expires  = 525600
      );
    else;
      $lang  = 'en';
      cookie_set(
        'lang'    = 'en',
        -domain   = $fw_requestPage->('domain'),
        -path     = '/',
        -expires  = 525600
      );
    /if;
  /if;
/protect;

// now we have a language var, wether it came from a cookie, was chosen by the user by
// clicking the language switch link, or was the default 'en'.
// so now add resulting value for var lang to the session
session_addvariable(-name='a', 'lang');

// also update the PageBlocks client language var with the current value
// this var is used by PageBlocks for internal localisation procedures, see later
$fw_client->('language') = $lang;

// now we can instantiate the fwp_recordData object based on the correct,
// language specific table
$lang == 'de'
  ? var('scndPages' = fwp_recordData('pages_scnd_de'))
  | var('scndPages' = fwp_recordData('pages_scnd_en'));
```

The two tables, 'pages_scnd_de' and 'pages_scnd_de', are identical in structure and content, except for the variation in language for the stored data.
The `$scndPages` instance can now be used for pulling data for display with the standard PageBlocks code:

```
$scndPages->(select(
  -select   = '*',
  -keyval   = 'myPageId',
  -withMakeVars
));
```

5.2 More Than One Website Language: The Navigational Elements

Continuing the example above, we also want to change the navigational elements based on language. Lets say the main navigation HTML looks like this:

```
<ul id="mainnav">
  <li class="p1">
    <a href="/home"></a>
  </li>
  <li class="p2">
    <a href="/products"[$fw_myUrl->('path') >> 'products' ? ' class="sel"']></a>
  </li>
  ...etc
```

Where `$fw_myUrl->('path')` is a PageBlocks var which returns the path from the current request. The CSS for this navigation section is stored in a separate, language specific CSS. For English the file is named 'nav_en.css' and can look something like this:

```
ul#mainnav {
  list-style-type: none;
  float: left;
  margin-top: 46px;
}
ul#mainnav li {
  display: inline;
  height: 21px;
  text-align: center;
  float: left;
}

ul#mainnav li.p1 {
  width: 84px;
  background: url(/site/media/images/mainnav_01_en-over.gif) no-repeat;
}
ul#mainnav li.p2 {
  width: 89px;
  background: url(/site/media/images/mainnav_02_en-over.gif) no-repeat;}
...etc

ul#mainnav a {
  display: block;
  height: 21px;
}

#mainnav li.p1 a {
  width: 84px;
  background: url(/site/media/images/mainnav_01_en.gif) no-repeat;
}
#mainnav li.p2 a {
  width: 89px;
  background: url(/site/media/images/mainnav_02_en.gif) no-repeat;
```

```
}
...etc.

ul#mainnav a:hover {
  background: transparent;
}

#mainnav li.p1 a.sel {
  background: url(/site/media/images/mainnav_01_en-over.gif) no-repeat;
}
#mainnav li.p2 a.sel {
  background: url(/site/media/images/mainnav_02_en-over.gif) no-repeat;
...etc.
```

Notice the background images that are language specific. There is in this case a similar CSS file for German, referring German background images.
Now, all we have to do is to pick the right CSS file while the HTML markup stays the same. This is done in the page config file, '_pageConfig.lgc', already mentioned above, with a conditional checking upon the language var we established in 5.1 above, and a specific PageBlocks method for adding (or removing) CSS file references in the <head>-tag (see more on this in 6.1 below):

```
$lang == 'de'
  ? $fw_headContent->(addCssFile(($fw_sPath->'css') + 'nav_de.css'))
  | $fw_headContent->(addCssFile(($fw_sPath->'css') + 'nav_en.css'));
```

A website built with PageBlocks which uses these techniques can be seen here: <www.hillenet.net>.

5.3 More Than One Website Language: Error Messages

Eventhough we got a website up and running with two languages here, we still need to think about error messages in case something goes wrong. This is covered in 5.7 below.

5.4 More Than One Website Language: Another Approach!

Another approach one can choose deserves mention here because it makes use of built-in features in PageBlocks that bring us close to a simple CMS - one can actually setup multi-lingual pages in very short time!
PageBlocks has the possibility of assigning socalled 'blockTemplates' in a given sub-directory (module). The feature can be turned on/off in the page config file, '_pageConfig.lgc', with this command (more on $fw_pageModes in 6.1 below):

```
$fw_pageModes->enableBlockTemplates;
```

You would then typically make a combination of one or a few repeating blocks (content that doesn't vary from page to page) and blockTemplate-files that handle the individual page content. The idea in this approach is that:
- each single page does not have a corresponding file but must, however, still be listed in the list of available pages for the subdirectory (100% abstracted URLs takes a little more but is certainly possible)
- the individual content for each page is entered via the PageBlocks -> SiteManager -> SiteStrings (more on this in 5.10 below) which handles an arbitrary number of page string languages

For an example of this approach see <www.pageblocks.org/ftrs> and the corresponding sub-directory (module) in the PageBlocks download package.

5.5 More Than One System Language

By "system language" I mean the language of all static texts in for example a CMS. It includes all navigational texts, headers, subheaders etc., quest-, feedback- and error messages, field labels and button texts in forms, help texts and other documentation etc. If you want to enable more than one system language in an admintrative area, then you are presented with a more complex task than the case above which was only the website.

PageBlocks includes a manager named MVStrings - Multi View Strings - which is the key to the kind of localisation we are speaking of here. MVStrings builts on some elements:
- language specific config files holding all error- and validation error messages

- language specific config files holding all other messages and text fragments
- database tables holding language specific text strings and value list strings
- objects handling string lookups, caching & more

The MVS manager can actually determine which string to display in a given situation based on not only language, but also on media (ie. display on handheld device or in desktop browser) and on an arbitrary *variant*. Its a complete system for very refined string display!

5.6 More Than One System Language: User Language Preference

PageBlocks chooses the language to use for display based on the value of $fw_client->language, which defaults to the core language, which, again, defaults to 'en' for English. You can set a different value for the core language (if you understand the consequences) and/or a different default value for $fw_client->language.

Handling language preference can be done as follows. Somewhere in the top of the '_pageConfig.lgc'-file for the current sub-directory (module) you add:

```
if(var('m_uLang')); // check if user submitted a language choice (see below)
  var('userLanguage' = $m_uLang);
else;
  !var('userLanguage') ? var('userLanguage' = fw_kCoreLanguage); // default to core language
/if;

$fw_client->(setLanguage($userLanguage));
```

The last line makes sure the language is kept throughout the session.
To make a language select option for the user to choose from you can make a select menu in for example the administration welcome page (simplified):

```
<select name="m_uLang" id="uLang">
  <option value="" selected="true">Choose System Language...</option>
  <option value="da">Dansk</option>
  <option value="en">English</option>
</select>
```

You store the language preference for the current user by adding the following in the logic block of the page with the select menu:

```
$fw_user->(addKeeper('userLanguage'));
$fw_user->(storeKeepers);
```

`$fw_user->(addKeeper())` and `$fw_user->(storeKeepers)` are PageBlocks functions that will store information about the current user for retrieval on next login. The values are restored automatically, meaning that the last stored value of var('userLanguage') is immediatly available after a successful user authentication. That is why we first check if that var is defined before defaulting to fw_kCoreLanguage above in _pageConfig.lgc (remember, _pageConfig.lgc loads before any logic- or whatever file).

5.7 More Than One System Language: Error- And Validation Error Messages

Error- and validation error messages are organised in 2 config files. They reside in a folder named '_pbStrings' in the root folder, their names are:

strings_coreErrors_en.cnfg
strings_coreValErrors_en.cnfg

Notice the language suffix 'en' for English. In the files, the messages are defined with an error number and a string.

The first thing to do when you want to add one or more languages to your website (as in 5.1-3 above) or to your project is to make copies of these files, save them with the appropriate language suffix and translate the messages in there. A discrete number of translations is, however, already available on the PageBlocks website.

Custom error messages and custom validation error messages are added as a combination of the methods for adding custom error routines and custom validation routines and custom string config files. While the methods for adding custom error routines and custom validation routines are outside of the scope of this presentation, the custom error string files need mention: those files are parallel to the core config files listed above, they live on the path /site/strings/ and are named as follows with 'app' for application:

strings_appErrors_en.cnfg
strings_appValErrors_en.cnfg

Those files are empty in a new PageBlocks project, if used they need translated versions exactly as their core counter parts.

5.8 More Than One System Language: Other Messages And Text Fragments - App Strings

You can store messages and text fragments other than the error messages in:

strings_coreStrings_en.cnfg

which lives in the same place as the two core error message files mentioned in 5.7 above, or in a file named:

strings_appStrings_en.cnfg

that lives on the same path as the two application error message files mentioned in 5.7 above.
Again, when doing localisation you would copy those files and save them with the appropriate language suffix. You wouldn't, however, store strings in 'strings_coreStrings_en.cnfg' unless you intentionally make up some core strings you plan on using in several projects.
Strings stored in '/site/strings/strings_appStrings_XX.cnfg' are available throughout the whole application. You can store strings one level deeper, in a PageBlocks *module*, which equals a sub-folder on root level. Such module string config files are named equal to the application wide file from above - the difference is that they live on a different path: '/<module>/_resources/strings/strings_appStrings_XX.cnfg'.

Strings in 'strings_appStrings_XX.cnfg' are defined with a name and a string and maybe media- and variant sub-divisions. Here is an example of a string definition from a module config file, 'strings_appStrings_da.cnfg', for language Danish:

```
{{modHeader:
variant_moduleContacts:
msg___ Kontakter og musikere
/variant;
}}
```

The string name is 'modHeader', the variant is 'moduleContacts' and the string itself is 'Kontakter og musikere'. The variant is here actually only used to keep the PageBlocks' strings caching mechanism on right track.
To apply that string in a display file you simply do:

```
[$fw_appStrings->modHeader]
```

This is possible because the MVS manager catches all application string names and makes them member tags of the `$fw_appStrings` object.

You would typically use the 'strings_appStrings_XX.cnfg' files - wether application-wide or module-wide - to store all text fragments that need localisation: feedback- and warning messages, button value texts, link- and title texts, legend texts etc. etc.

5.9 More Than One System Language: Field Labels, A Special Case

Field labels are typically defined in '/<module>/_resources/strings/strings_appStrings_XX.cnfg' as they usually are module-specific. Defining a string for a field label is done the exact same way as in the example shown above. I tend to use the input names (see 3.3 above) as string names in the 'strings_appStrings_XX.cnfg' files because it decreases the number of things to remember, but their names could be anything. Here the Danish definition for the label 'Last Name' we used in 3.5 and 3.6 above:

```
{{m_ctMaNmeL:
variant_moduleContacts:
msg___ Efternavn
```

```
/variant
}}
```

To use in display you do:

```
<label for="ctMaNmeL">[$fw_appStrings->m_ctMaNmeL]:</label>
<input type="text" class="size2" name="m_ctMaNmeL" id="ctMaNmeL" value="[var('m_ctMaNmeL')]" />
```

Now, this creates a problem: As we saw in 3.6 above, the PageBlocks validation mechanism takes the label names for use in it's error messages from the `-haslabel` validation code in the table config files. But now we just made a nice, language dependend solution for the input field labels via the MVS manager - how do we combine those two?

Fortunately, the `-haslabel` validation code inside the table config files can process Lasso! This means that the syntax for retrieving a string value shown in 5.8 above can be used in the table config file as well. If you make comparison to the validation codes shown in 3.3 and 3.6 above, we would now instead write (dataType substituted by <>):

```
{{tableModel___
#inputName    fieldName       <>   validation codes
#-----------  --------------  <>   ----------------------
m_ctMaTle     cnctMainTitle   <>   maxlen=30, isAlphaNumeric, haslabel=[$fw_appStrings->m_ctMaTle]
m_ctMaNmeF    cnctMainNameF   <>   maxlen=20, -extn, haslabel=[$fw_appStrings->m_ctMaNmeF]
m_ctMaNmeL    cnctMainNameL   <>   isRequired, maxlen=20, -extn, haslabel=[$fw_appStrings->m_ctMaNmeL]
m_ctMaIsStd   cnctIsStandard  <>   maxlen=1
}}
```

Voilà, field label names are now both language dependend in display AND as they appear in the validation error messages!

5.10 More Than One System Language: Strings Longer Than Texts Fragments - Page Strings

The MVS manager in PageBlocks has still more to offer! As touched upon in 5.4 above, language dependend strings can also be stored in a database table, instead of in the config files we looked at so far This is for example useful for texts longer than a few words and for content text on a public website, eventhough there isn't any technical limit to the length of strings defined in the config files. (For the completeness of this presentation it should be mentioned that application strings also can also be stored in a database table, taking application strings even further.)

Page strings stored in the database table are entered and edited in the PageBlocks access restricted administration system. Any PageBlocks installation comes with a ready-to-use Site Manager, which is an administrative interface to the user management, named Administrators, and to the page strings, named Site Strings and a few other things.

To add a page string you need to specify:
- the path to the page where you want to use it
- the language code (if different from core language)
- a name
- a media name (defaults to 'all')
- a variant name (defaults to 'default')
- the string (can be styled with PageBlocks styling markup)

The interface is pretty straight forward. You will always have a core language in which you define your strings in the first place. You can then edit other language versions next to the core version, allowing for a fairly intuitive process.

Strings stored in the config files as shown 5.8-9 are called with `[$fw_appStrings->stringName]` as already mentioned, while strings stored in the database table are called with `[$fw_pageStrings->stringName]`. Contrary to $fw_appStrings, $fw_pageStrings are, among others, defined by the path to the page for which they are to be used, allowing for page strings with the same name but different page paths.

As mentioned in 5.4 above, this is close to be a simply CMS. Actually, the Site Strings system can be taken as inspiration for a CMS of your own. Or you can indeed give access to it as-is for certain users to edit strings, it is, however, maybe not intuitive enough for all kinds of users, and the PageBlocks Site Manager itself isn't localised as of current ;-)

5.11 Value Lists

PageBlocks also has a language dependend value list manager (API). Value lists for select menues, radio buttons and check boxes can be defined with language dependend string values and are stored in a database table. To create value list with a set of values stored in the database you do something like this in the logic-block:

```
var('instrTypes' = fwp_valueList(
  -table    = 'vlists',
  -list     = 'instrumentTypes',
  -scope    = 'site',
  -titleOption= $fw_appStrings->menu04link03a + '...',
  -attributes = map(
    'name' = 'vl_id',
    'id' = 'vl_id',
    'onchange'  = 'if(this.selectedIndex != 0){ this.form.submit(); }'
  )
));
```

The values will be language dependend because `fwp_valueList` uses `$fw_clint->language` to determine which language version to pick. The select menu is then generated in the display-block with this simply line:

```
[$instrTypes->drawAsPopup]
```

5.12 Switching The MVS Manager Off And On

Despite caching of all application strings, the MVS manager causes some overhead. Currently page strings aren't cached (its on the TODO-list), so the use of page strings will cause database lookup on each page.

Its quite a time consuming challenge to convert an already-made project with all strings hardcoded into it to a language sensitive project, so you will always want to apply the techniques described in this entire chapter from day one, except of course if you are 100% sure you will never need localisation. For such cases the MVS manager can be switched either completely off for the whole project in the master config file, or switched off for a module in the page config file with:

```
$fw_pageModes->disablePageStrings;
```

Actually, 'disabled' IS the default setting for the MVS manager! So what you need to do is really the opposite: if you want to use it - either project-wide or module-wide - you switch it ON with:

```
$fw_pageModes->enablePageStrings;
```

6. Handling Of CSS- and JavaScript Files In PageBlocks

6.0 Customizing automatic processes

As PageBlocks is a tool that takes care of the page assembly process, some areas are "hidden" in automatic procedures. But "hidden" isn't the right word, there is nothing in this process that can't be taken care of on a more individual basis.

Everything inside the <head>-tag is a good example of this. You can set a few vars in the project master config file, and PageBlocks will render a discrete set of tags inside the <head>-tag.

If you need more control (and you usually do), you enable a couple of switches in the application init files, and now you can control basically everything that happens inside the <head>-tag via three config files living in a folder named '_pbLibs':

_defineCSS.lgc
_defineJavaScript.lgc
_definePageHead.lgc

Lets say you want to load a special CSS file based on a condition looking for a certain browser type (did I say MSIE...?) then you would code this conditional in '_defineCSS.lgc'.

6.1 $fw_headContent And $fw_pageModes

But there are some customisations that can be applied very easily with two of those handy tools in PageBlocks:

`$fw_headContent`, already mentioned in 5.2 above

`$fw_pageModes`, already introduced in 5.4 and 5.10 above

$fw_headContent can add and remove certain content from the <head>-tag on a project-wide, a module-wide or a page-wide basis. $fw_pageModes is used to control a number of page assembly options. The different usages of those two controllers are covered in the Developer Guide, here I just wanted to mention a few that are connected with the use of CSS and external Javascript files.

To add a CSS file named 'admin_import.css' living on the PageBlocks standard path for CSS files, '/site/css/', you do:

```
$fw_headContent->(addCssFile($fw_sPath->('css') + 'admin_import.css'));
```

To remove the file you do:

```
$fw_headContent->(removeCssFile($fw_sPath->('css') + 'admin_import.css'));
```

Notice the `$fw_sPath`. The 's' stands for 'site', there is a parallel one, `$fw_mPath`, with 'm' for 'module'. They both return a number of standard paths in PageBlocks and can be used for things like the code example above.
To add or remove a Javascript file named 'scripts_main.js' living on the PageBlocks standard path for Javascript files, '/site/js/', you do:

```
$fw_headContent->(addScriptFile($fw_sPath->('js') + 'scripts_main.js'));
$fw_headContent->(removeScriptFile($fw_sPath->('js') + 'scripts_main.js'));
```

As I said before, this code can be inserted at any level, down to a per-page basis. So you could use it for including page specific Javascript files. PageBlocks has another option for this, however, using `$fw_pageModes`:

```
$fw_pageModes->enableJSPerPage;
```

Again, it can be used on a project-wide, a module-wide or a page-wide basis, but a typical use would be module-wide or page-wide. What it does is that it looks for a Javascript file with the same file name as the page currently being called on the PageBlocks standard path for module-resources: '/<module>/_resources/js/'. So if the page currently being called has the name 'contactEdit' inside the module 'contacts' and it has `$fw_pageModes->enableJSPerPage` specified, then PageBlocks will look for this file: '/contacts/_resources/js/contactsEdit.js'. PageBlocks will add the Javascript file if its in there, otherwise it fails silently.

To make it a little easier to load a few standard AJAX/Javascript libraries, PageBlocks comes with these libraries pre-installed ready to add with `$fw_pageModes`:

```
$fw_pageModes->enablePrototype;
$fw_pageModes->enableScriptaculous;
$fw_pageModes->enableJquery;
```

They all load a local library included with the PageBlocks package, they don't load an external library resource as often is done in the case of jQuery. This could be added to the TODO-list if wished.

6.2 Dynamically Generated Javascript

PageBlocks doesn't at present have a pre-defined way of adding dynamically generated Javascript. It is my wish to add this functionality at some point; what would be ideal would be to have Javascript validation scripts automatically generated based on the validation codes entered in the table config files (see 3.3 above).

There is, however, a straight forward way to add dynamically generated Javascript based on the values of Lasso variables. Since all Lasso logic code must be executed before starting to generate dynamic Javascript, the method makes use of the fact that you can add Javascript at any point in the page rendering process.
PageBlocks has the ability to enable - or disabling - repeating blocks of code for a any number of pages. Again, `$fw_pageModes` is used for that:

```
$fw_pageModes->(setRepeatLogic(array('left','main')));
$fw_pageModes->disableRepeatBlocks;
```

This usually serves other purposes such as adding repeating code to a layout - for example a navigation bar.

But the variant

```
$fw_pageModes->(setRepeatBelow(array('main')));
```

has the advantage that it executes after anything else. So if you add a file in your module named 'repeatBelow_main.dsp' and enables it via `$fw_pageModes` where needed, then you have a perfect place for adding dynamically generated Javascript like the following (based on Prototype):

```
<script type="text/javascript">
  var lang = $F('cl_l');
[iterate($contactObject->getPhones, local('i'))]
  new Form.Observer(
    'phoneform_[#i->(get:2)->(get:1)]',
    0.3,
    function() {
      $('updatephoneitem_[#i->(get:2)->(get:1)]').setStyle('background:
url(/site/media/images/saveBtn-a_' + lang + '.gif) no-repeat left top;');
    }
  );
  $('updatephoneitem_[#i->(get:2)->(get:1)]').observe('click',
updatePhoneItem.bindAsEventListener(this, [#i->(get:2)->(get:1)]));
  $('delphoneitem_[#i->(get:2)->(get:1)]').observe('click', deleteItem.bindAsEventListener(this,
'phone', [#i->(get:2)->(get:1)]));
[/iterate]
</script>
```

Nikolaj de Fine Licht, Brazil/Denmark, July 2008