Knop Web Application framework

Why framework

Advantages

Higher Productivity

Higher Quality

Features

Disadvantages

Less flexibility

Performance hit

Introducing Knop

Goals with Knop

Be flexible

Be lean

Be focused

Be helpful

Follow standards

Encourage a rich user experience

What is Knop?

Example 1 - a simple form

1-simple-form.lasso

1-simple-form-response.lasso

Example 2 - a form talks to a database

2-form-and-database-config.lasso

2-form-and-database.lasso

2-form-and-database-response.lasso

The Knop Modules

Knop_nav

Example of a virtual URL (path based navigation)

Example of a parameter based URL

3-nav/index.lasso

Knop_database

Knop_form

Knop_grid

Knop_lang

Knop_user

The Knop file structure

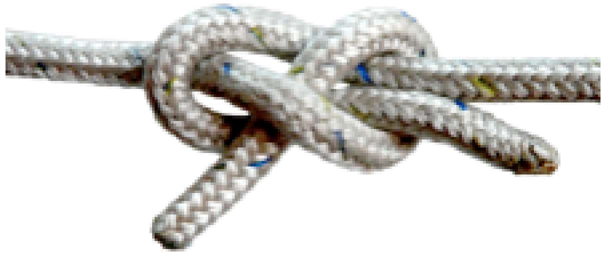
Knop application flow

Handle a form submission

Other Knop features

Debugging

Knop



Knop Web Application Framework
For Lasso Professional

Why Framework

Advantages

Higher Productivity
Using a framework lets you focus more on the core functionality of the web site or application, and worry less about the nitty details. It becomes easier to reuse code and modules. You get more done in less time.
Higher Quality
A framework becomes increasingly tested over time. The risk for silly and basic errors is reduced since the framework takes care of the basic functions. You can focus on the important things. You get less bugs to chase.
Features
A framework already has much of those bells and whistles you want to add to the web site but don't have time or budget to even think about. You get many things for free.

Disadvantages

Less flexibility
By using a framework you are more or less bound to the framework's view of the world. It can be problematic to do things in other ways than the framework has intended, or things that go beyond what the framework offers.
Performance hit
A framework puts you a little bit further up above the low level code. There is always some degree of tradeoff between efficient development and raw performance. Higher abstraction level costs CPU cycles.

Introducing Knop

Goals With Knop

Be flexible

Let the developer use the bits they want, allow for customizations and special needs.

Be lean

Stay lightweight, beware of feature bloat, take advantage of native Lasso features as far as possible.
`http://gettingreal.37signals.com/ch15_Beware_the_Bloat_Monster.php`

Be focused

Cover a few main areas where a framework is the most useful, and do it well, Don't try to solve every need.

Be helpful

Don't get in the way.

Follow standards

Encourage the use of modern standards-based and semantically correct HTML and CSS for presentation.

Encourage a rich user experience

Use client side scripting as progressive enhancement to improve user experience and application responsiveness, but do not rely on client side scripting for critical functionality. Use Ajax techniques where really motivated.

What Is Knop?

Knop is mainly two things:

1. **A set of custom types** that implements the core functionality of the Knop modules.
2. **A defined application flow** and file structure to support the site's functionality and application logic.

You can choose to only use selected parts of Knop. For example you can use just knop_form to handle forms, or you can use just knop_database to get a richer database abstraction than what Lasso's native inlines offer. But if you use all of Knop, this is what you get:

- A number of modules implemented as custom types that handle the core functions of the framework.
- A defined structure to handle the logic of a web site, to handle page requests, act on form submissions and show the resulting page. A Knop site or application is handled through a single control file, similar to the Onefile structure.
- A defined folder and file structure for the includes needed to handle and support the application logic.
- Support for a modular architecture, where a self contained module easily can be plugged into an existing site.
- Basic templates for HTML and css.
- Support for URL handling with virtual (abstracted) URLs.

Example 1 - A Simple Form

Before we get into the details of Knop it could be useful with a couple of demonstrations on how Knop can be used. One of the Knop modules is a forms generator. Knop_form is a custom type, just as the other Knop modules. It is used to create HTML forms and to help processing the form submission.

This page shows an HTML form.

1-simple-form.lasso

```
[
// Create a new form object
var: 'form'=knop_form;

// Add text fields and a submit button
$form -> (addfield: -type='text', -name='firstname', -label='First name');
$form -> (addfield: -type='text', -name='lastname', -label='Last name');
$form -> (addfield: -type='textarea', -name='message', -label='Message');
$form -> (addfield: -type='submit', -name='button_send', -value='Send message');

// Show the form on the page
]

<form action="1-simple-form-response.lasso">
[$form -> renderform]
</form>
```

To take care of the form input the response page also needs to know about the form, so we first have to define the same form object again (of course we normally define the form object in an include but for this demonstration we just repeat the form definition in the response file).

1-simple-form-response.lasso

```
[
// Create the same form object again
var: 'form'=knop_form;
$form -> (addfield: -type='text', -name='firstname', -label='First name');
$form -> (addfield: -type='text', -name='lastname', -label='Last name');
$form -> (addfield: -type='textarea', -name='message', -label='Message');
$form -> (addfield: -type='submit', -name='button_send', -value='Send message');

// Load field values from the submission
$form -> loadfields;

// Look at the fielvalues
$form -> updatefields;

]
```

The output from `$form -> updatefields` is a pair array, and one of the nice things with a pair array is that it can be used as dynamic input in an inline. This is very handy and can be used for example like this:

```
inline: -database=...,
      $form -> updatefields,
      -add;
/inline;
```

Example 2 - A Form Talks To A Database

Now we are going to combine two Knop modules to see how they can interact. We will create a database object that the form will interact with. We will also specify an HTML form action in the form object to make it self contained so a complete HTML form can be rendered easily. Finally we'll hook the database object up to the form object. This is where the fun begins.

In this example we move the configuration to an include file that can be reused.

2-form-and-database-config.lasso

```
[
// Create a database object
var: 'db'=(knop_database: -database='knopdemo', -table='customer', -keyfield='id');

// Create a form object
var: 'form'=(knop_form: -formaction='2-form-and-database-response.lasso', -database=$db);

// Add text fields and a submit button
$form -> (addfield: -type='text', -name='firstname', -label='First name');
$form -> (addfield: -type='text', -name='lastname', -label='Last name');
$form -> (addfield: -type='textarea', -name='message', -label='Message');
$form -> (addfield: -type='addbutton', -value='Send message');

]
```

The input page is a simplified version of example 1, since we've moved the configuration to an include and since weve made the form object self contained.

2-form-and-database.lasso

```
[
// Example 2 - A form talks to a database
// Configure database and form objects in an include
include: '2-form-and-database-config.lasso';

// Show the form on the page, this time the form object is a complete html form
$form -> renderform;
]
```

The response page is really simple. It actually just takes one single line of code to do what it should.

2-form-and-database-response.lasso

```
[
// Configure database and form objects in an include
include: '2-form-and-database-config.lasso';

// Handle the form submission
$form -> process;

// Show the result
]

[$form -> (renderhtml: -excludetype='submit')]
Adding record [$db -> error_msg]
```

Now we have seen the basic principles of a few common Knop operations.

The Knop Modules

The Knop modules are implemented as a number of custom types supported by a few custom tags. Some of the custom types can interact with each other.

The Knop custom types are described below.

Knop_nav

Knop_nav is the heart of a Knop site. It defines the site's structure and navigation. It also keeps track of and validates the visitors current location on a site. It is used to control the application logic of the website or application, keeps track of and processes all include files, generates the navigation menu (a nested ul/li list as default) and breadcrumb, parses the current URL, generates URLs for site internal href links. Knop_nav is the engine room for a site and acts as the main dispatcher for requests and actions.

Knop_nav supports both fully virtual URLs (using an atbegin-based URL handler) as well as parameter based URLs for situations where virtual URLs can't be used.

Example of a virtual URL (path based navigation)

```
http://myhost.com/mypath/tothe/page/
```

Example of a parameter based URL

```
http://myhost.com/?mypath/tothe/page/
```

The two navigation methods result in URLs that look almost the same, the only difference is the "?" after the hostname. Switching navigation method is just a matter of changing a parameter of the navigation object so it's very easy to deploy a Knop based site no matter if it will be hosted on a server that supports virtual URLs or not.

The visitor's current location is called "path" and the current action (if any) is identified by "actionpath".

- Knop path = "where we are" (the page we are coming to).
- Knop actionpath = "what to do" (the page we came from).

Knop_nav can interact with knop_grid.

3-nav/index.lasso

```
[
//Example 3 - Knop_nav

// Create the parent nav object.
var: 'nav'=(knop_nav: -navmethod='param', -currentmarker=' »');

// Define the site structure
$nav -> (insert: -key='home', -label='Home Page');

// Create a child nav object
var: 'subnav'=knop_nav;
$subnav -> (insert: -key='latest', -label='Latest News');
$subnav -> (insert: -key='archive', -label='News Archive');

$nav -> (insert: -key='news', -label='News', -children=$subnav);

// Determine current location so the nav object knows where we are
$nav -> getlocation;

// Generate navigation menu
$nav -> renderhtml;
// Generate a breadcrumb
$nav -> renderbreadcrumb;

]

<h1>The current page is [$nav -> label]</h1>
<p>The current framework path is [$nav -> path]</p>
```

Knop_database

This is a database abstraction layer that sits on top of Lasso's own database abstraction. It supports both regular Lasso inlines and SQL syntax. MySQL and FileMaker databases are supported currently.
Knop_database provides convenient access to basic CRUD operations (Create, Read, Update, Delete) and has built-in support for record locking, safe random keyvalues and duplicate prevention. A found set of records can either be iterated, or Lasso's native records tag can be used to access the found set. Knop_database can maintain a persistent pointer to a specific record, much like Active Record.
Knop_database primarily uses pair arrays as field specifications, which makes it easy to integrate with standard Lasso inlines, but can also use SQL statements for some of the operations. When interacting with knop_form and knop_grid, pair arrays are normally used to exchange field data and other search parameters. The use of pair arrays for standard inlines is one way to provide greater flexibility.
Knop_database can interact with knop_form, knop_grid and knop_user (for record locking).

Here's an example to demonstrate how to use knop_database to output some fields from a specific database record.

```
// initiate the database object (normally in a config file)
var: 'db_news'=(knop_database: -database='acme', -table='news',
      -username='*****', -password='*****',
      -keyfield='id');

// perform a database search to grab the record (normally in a lib file)
$db_news -> (getrecord: -keyvalue=185);

// show some fields from the database record (normally in a content file)
<h3>[$db_news -> (field: 'title')] </h3>
<p>
[encode_break: ($db_news -> (field: 'text'))]
</p>

----

// The getrecord statement can be simplified slightly since the first parameter
// is the keyvalue
$db_news -> (getrecord: 185);

// The field calls can be simplified using a shortcut that maps unknown
// member tags to field names
<h3>[$db_news -> title] </h3>
<p>
[encode_break: ($db_news -> text)]
</p>

// If a more complex query is needed to get the record, an SQL statement can be used.
```

```
$db_news -> (getrecord: -sql='SELECT * FROM news LEFT JOIN ...');

// A general select can be used as well.
// The data from the first found record will be available as ->field.
$db_news -> (select: -sql='SELECT * FROM news LEFT JOIN ...');


// To output a record listing from a database search, different methods can be used.
// 1. A standard records loop (fastest)
      records: -inlinename=($db_news -> inlinename);
            field: 'title';'<br>';
      /records;
// 2. Iterate the database object
      iterate: $db_news, var: 'record';
            $record -> (field: 'title');'<br>';
      /iterate;
// 3. Use the record pointer
      while: $db -> nextrecord; // increment the record pointer
            // (nextrecord returns true as long as there are more records to show)
            // fetch data from the record the record pointer currently points at
            $db -> field('title');'<br>';
      /while;
```

Knop_form

Forms are one of the most tedious things to handle manually in a web application. First the form fields should be shown on the edit page. They need labels, proper styling and different properties. They may also need initial values to show in the form fields. The values can either be static, come from a database lookup, or from a previous submission of the same form if there was an input error that needs to be corrected, and in that case the erroneous fields or labels needs some highlighting to guide the user. Finally the form submission must be handled by validating the user's inputs and then storing the form data in a database.

All these tasks are a perfect target to make things easier for the developer.

First we define the form. We give it a form action, and we add the fields and other elements such as submit buttons that the form should contain. The fields can have the same properties as regular HTML form fields do, and they have additional properties to define the options of a select menu, the checkbox options of a checkbox field set, for interaction with databases and other purposes.
Then we fill the form fields with data. It can either come from a form submission or from a database lookup. In the case of a database lookup, the corresponding database field has been declared as property for each form field.
If we want we can set a template for the form to define how the form should be presented in HTML, or just let it use the default template.
Finally we render the form on the page. We can render the entire form at once, or specific fields at a time (we can even set different templates for every field), to have the flexibility needed to be able to accommodate the form in just about any HTML context.
The form object even generates some javascript for us that will warn the user if he navigates away from a "dirty" page (a page that has unsaved changes) and other useful features.
The next step is that the user submits the form. Now the form object makes its second entry by taking care of the form submission. Since all form fields are defined in the form object, it knows where to put each field when we tell it to load data from the form. Since it knows what kind of data is allowed in each field the form object can validate itself with a single call.

If the validation comes across an input error, the form objects prepares to show itself again but this time with the erroneous inputs highlighted.
If the validation passed, the form object comes back to our help once again and provides us with a complete pair array with field name and value pairs (the form fields knew what database fields they correspond to, remember?) that we can feed right into an inline to add or update a database record, or we can get an SQL string that we can put in an SQL statement of our liking.
Knop_form can interact with knop_database.

Knop_grid

This custom type is used to display record listings with sortable columns, pagination, detail link to edit a record, filtering/quicksearch, etc. It requires a reference to a knop_database object because they are so tightly related. It can highlight the affected record when returning to the listing after adding or editing a record.
We can also give it a reference to a knop_nav object, to get the right pagination links and other things. It can also provide a basic "Quicksearch" functionality integrated with the record listing.
Quicksearch and the sort headings generate pair arrays or SQL snippets to interact with knop_database. Sort parameters and the quicksearch query is automatically propagated through a knop_form, so the same set of records is selected after editing a record.
Knop_grid must interact with Knop_database and can optionally interact with Knop_nav.

Knop_lang

This custom type handles language strings for multilingual presentation of the user interface. A knop_lang object holds the language strings for all supported languages. Strings are stored under a unique text key, but the same key is of course used for the different language versions of the same string.
Language strings can be grouped into different knop_lang object instances (variables) if it helps managing them.

When the language of a knop_lang object is set, that language is used for all subsequent requests for strings until another language is set. The selected language is shared between all knop_lang objects on the same page for that visitor, unless another language has been set specifically for an individual knop_lang object.

If no specific language is set on the page, knop_lang uses the browser's most preferred language if it's available in the knop_lang object, otherwise it defaults to the first language (unless a default language has been set for the knop_lang object).

The strings in a knop_lang object can contain replacement placeholders, to be able to insert dynamic text when retrieving a string. The strings can also be a Lasso compound expression which will be evaluated at runtime when the string is retrieved.

Examples
```
	var: 'lang_messages'=(knop_lang: -default='en');
	$lang_messages -> (addstring: -key='welcome', -value='Welcome to the home page', -
	language='en');
	$lang_messages -> (addstring: -key='welcome', -value='Välkommen till hemsidan', -
	language='sv');
	$lang_messages -> (addstring: -key='loggedin', -value='You are logged in as #1# #2#', -
	language='en');
	$lang_messages -> (addstring: -key='loggedin', -value='Du är inloggad som #1# #2#', -
	language='sv');
```

```
// proper call, defaults to the browser's preferred language
$lang_messages -> (getstring: 'welcome');
// shorthand call
$lang_messages -> welcome;

// change language
$lang_messages -> (setlanguage: 'sv');
$lang_messages -> welcome;

// proper call with replacements
$lang_messages -> (getstring: -key='loggedin': -replace=(array: (field: 'firstname'),
(field: 'lastname')));

// shorthand call with replacements
$lang_messages -> (loggedin: -replace=(array: (field: 'firstname'), (field: 'lastname')));
```

-> addlanguage is suitable if you use config files to configure strings, like the strings in knop_grid. It looks like this:

```
#lang -> (addlanguage: -language='en', -strings=(map:
      'quicksearch_showall' = 'Show all',
      'quicksearch_search' = 'Search',
      'linktext_edit' = '(edit)',
      'linktitle_showunsorted' = 'Show unsorted',
      'linktitle_changesort' = 'Change sort order to',
      ...
```

Knop uses knop_lang internally to handle text strings. By providing access to the internal lang object that a Knop module uses, it is ieasy to add custom localizations or modified strings also to the core Knop modules without actually altering Knop itself. As an example if you want to localize an instance of knop_grid to another language on the fly, you can first find out what strings that need to be localized by calling $grid -> lang -> keys. This gives you an array of all string keys that are used across all defined languages.

Then you can just add the new language like this (for quasi Danish), since the ->lang member tag returns a reference to the internal knop_lang object:

```
$grid -> lang -> (addlanguage: -language='da', -strings=(map:
      'quicksearch_showall' = 'Finn alt',
      'quicksearch_search' = 'Søk',
      ...
```

Knop_user

The Knop_user custom type handles user authentication, maintains information about the user and keeps track of permissions for the user.

Authenticating a user checks the login credentials against a specified table, with support for one way encrypted passwords (with salt) and delays between repeated login attempts to prevent brute force attacks. User authentication can also be performed through custom code outside of Knop_user.

Knop_user prevents session sidejacking by comparing a client fingerprint between each page request.

Knop_user is the only Knop custom type that is intended to be stored in a session variable, and actually relies on this.

When a user is being authenticated, all available fields from the user table are stored in the Knop_user variable so user information can be retrieved easily throughout the session. Any additional custom data for the user can also be stored manually in the Knop_user variable.

Knop_user can keep track of user permission by storing arbitrary permission information in the Knop_user variable.

Knop_user enhances Knop_database objects by keeping track of record locks set by the user, and releasing record locks for example when navigating to a list of records without saving an edited record.

A basic example

```
var: 'session_user'=(knop_user: -userdb=$users);
session_start: -name='test';
session_addvar: -name='test', 'session_user';

$session_user -> (login: -username=(action_param: 'u'), -password=(action_param: 'p'));

if: $session_user -> auth;
      if: $session_user -> groups >> 'admin';
            $session_user -> (setpermission: 'candelete');
      /if;
else;
      'Authentication failed, ' + ($session_user -> error_msg);
      abort;
/if;

'Welcome, ' + ($session_user -> firstname) + '! ';
if: $session_user -> (getpermission: 'candelete');
      'You are allowed to delete records. ';
/if;
```

The Knop File Structure

A Knop site is built around a single file (for example index.lasso) that acts as a "control center" or main dispatcher, similar to the "Onefile" concept. The main files are of the following types:

- Config - (page specific configuration) configures the request handler, configures the business logic
- Action - request handler, manipulates data
- Library - user interface logic, prepares information to display to the user
- Content - display the information to the user

Files are named with a prefix that tells what kind of file it is, then the Knop path with “/” replaced by “_”. A few special files are named with a double underscore after the prefix. Files are grouped by their type into folders named _config, _action, _library and _content.

Example file structure:

```
_config/
      cfg__global.inc
      cfg__nav.inc
      cfg_customer_edit.inc
      cfg_customer_list.inc
      cfg_news_archive.inc
      cfg_news_latest.inc
_action/
      act_customer_edit.inc
      act_customer_list.inc
      act_news_archive.inc
      act_news_latest.inc
_library/
      lib_customer_edit.inc
      lib_customer_list.inc
      lib_news_archive.inc
      lib_news_latest.inc
_content/
      cnt_customer_edit.inc
      cnt_customer_list.inc
      cnt_news_archive.inc
      cnt_news_latest.inc
index.lasso
```

Knop Application Flow

To explain the application flow of Knop, let's assume we have a web application where the user submits a form and we will walk through the processing of the form submission.

Every page request has one or two vital parameters:

- path (required)

This is the visitor's current location in the application. The path tells the application “where we are”.

- actionpath (optional)

If the current page request is the result of a form submission, the application needs to know what to do with the input. The actionpath tells the application “what do to”.

Don't mix up the actionpath with the 'action' HTML attribute of the form tag itself!

Handle A Form Submission

A) Take care of the input (controlled by actionpath)

1. Find out the actionpath, which is where the submission comes from (this is determined by knop_nav -> getlocation)
2. Load the config for actionpath to define the forms etc for the page we came from. This is crucial to be able to handle the form submission.
3. Perform the actual action by loading form data, validate input and execute the logic needed in response to the form submission.

B) Prepare the output (controlled by path, which was also determined by knop_nav -> getlocation)

4. Was action successful? (form validation ok, database action wihtout errors etc) ->

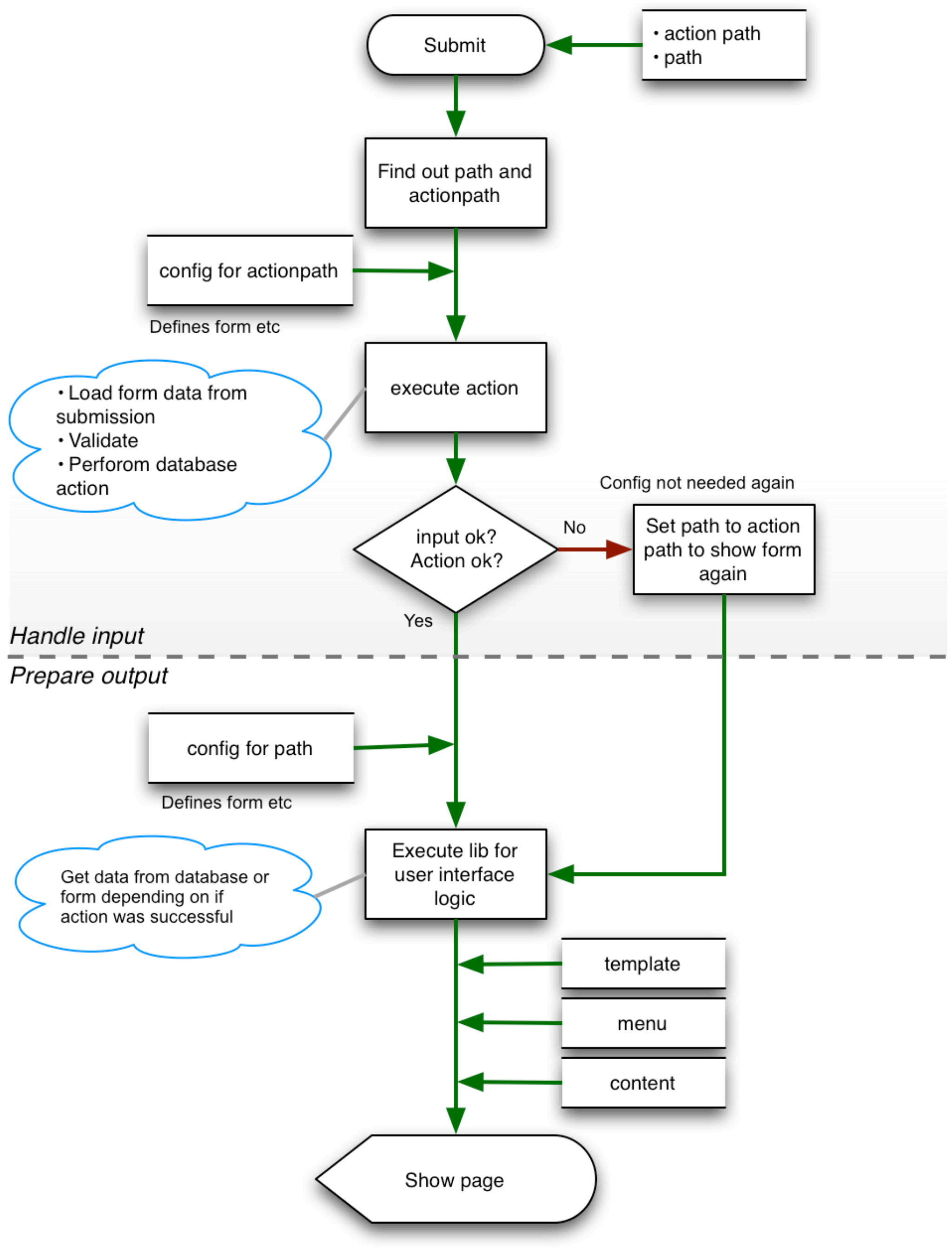
    4a. Load config for 'path' to define form, grid etc for display

Action not ok? ->

    4b. 'path' is set to 'action path' (config does not need to load again) to show the form again

5. Execute the library file for 'path' to prepare the page output
6. Include page template to build the page HTML framework with navigation menu, content area, sidebars etc.
7. The template includes the content for 'path' to generate the actual page contents
8. The finished page is served to browser.

All the include files that are needed to handle the application flow are chosen automatically by Knop_nav.


Submit
• action path
• path
Find out path and actionpath
config for actionpath
Defines form etc
execute action
• Load form data from submission
• Validate
• Perforom database action
input ok? Action ok?
No
Yes
Config not needed again
Set path to action path to show form again
Handle input
Prepare output
config for path
Defines form etc
Execute lib for user interface logic
Get data from database or form depending on if action was successful
template
menu
content
Show page

Other Knop Features

Debugging

All Knop modules log debug information internally, which can be output easily to be able to track what's happening inside each module.

Caching

Caching can be used to reduce the overheads from configuration that is mostly static, used globally and normally doesn't change such as navigation, database objects, language strings , etc. Configuring all of this repeatedly on every page load is a waste of resources.
The Knop cache consists of two custom tags that can be used like in this example:

```
if: !(knop_cachefetch: -type='knop_lang');
      var: 'lang_nav_key'=(knop_lang: -default='sv');
      $lang_nav_key -> addlanguage(-language='sv', -strings=(map:
            'hem'='hem',
            'loggain'='loggain',
            ...
      knop_cachestore: -type='knop_lang';
/if;
```

`knop_cachestore` stores all page variables of the specified type in a global variable. It does this by iterating through all page variables and checking their type, copying the variables that have a matching type. This way the cache is populated mostly automatically and transparently. There is almost nothing to configure and it just works.
`knop_cachefetch` tries to recreate all the cached page variables of the specified type from the global variable, if there's a cached version available. It returns true if it was successful and false if there wasn't a cached copy available. If there wasn't a cache available, no page variables are created and the configuration needs to be set up from scratch. This is easy to do by using knop_cachefetch in a condition around the normal configuration.

The idea is to call knop_cachefetch first to try to have all knop object instances recreated as page vars, and if knop_cachefetch returns false then configure the knop objects the normal way and call knop_cachestore to cache them for the next page load.
The default cache expiration is 10 minutes (600 seconds). This means that instead of setting up the configuration repeatedly for every single page load for every user, the configuration only needs to be loaded once every 10 minutes by a single user for the benefit of all others.

Some example timings just to give an idea of how much it can help:
Without caching:

- Created langauge strings 23 ms
- Created database objects 73 ms
- Created navigation 142 ms

With caching:

- Created langauge strings 5 ms
- Created database objects 8 ms
- Created navigation 7 ms

The cache can be forced to refresh simply by adding a condition to cache_fetch:

```
            if: $cache_refresh || !(knop_cachefetch: -type='knop_database');
```

```
                        ...
```
If $cache_refresh is true in this example then thew cache will be ignored so the configuration will be set up again.

The caching can also be done per visitor by specifying a session name to use for cache storage. The specified session must be started before using the knop_cache tags with session. knop_cachestore adds the session variable $_knop_cache to the session. Using session to store the cached data is useful for example for navigation, where the configuration can be different for each visitor.
Example:
```
        if: !(knop_cachefetch: -type='knop_nav', -session=$session_name);
                var: 'nav'=(knop_nav:
                        -default=($lang_nav_key -> hem),
                        -root=$siteroot,
                        -navmethod=$navmethod);
                $nav -> insert(-key=($lang_nav_key -> hem), -label=($lang_nav_label -> hem), -
        url='/');
                ...
                knop_cachestore: -type='knop_nav', -expires=1200, -session=$session_name;
        /if;
```
The global variable used for caching is named uniquely for the current site (based on server_name and response_localpath - response_filepath) and it's also possible to specify a -name to further isolate the cache storage if needed (for example if multiple sites are running in the same virtual root and hostname). The global variable is accessed using thread locking to provide a thread safe caching mechanism.

Multiple Ways To Work With Site Modules

Knop's framework folder structure is actually quite liberal. It lets you collect all files in a central _knop directory to be able to centralize modules so they can be shared between different sites. It also lets you modularize parts of a solution in separate _mod directories.
The defined Knop directory tree consists of folders with name _knop, _config, _action, _library, _content or with names that begin with _mod_

Knop looks for framework include files in no less than 10 locations for each file naming convention you specify. For the framework path customer/edit, the actual name and location of the library file for that path can be any of the following:
A) -filenaming='prefix' (this is the default if -filenaming is not specified)

1. _mod_customer/lib_customer_edit.inc	// modular prefixed with module name
2. _mod_customer/lib_edit.inc	// modular
3. _mod_customer/_library/lib_customer_edit.inc	// modular separated, prefixed with module name
4. _mod_customer/_library/lib_edit.inc	// modular separated
5. _library/lib_customer_edit.inc	// collective ("all modules together") separated
6. _knop/_mod_customer/lib_customer_edit.inc	
7. _knop/_mod_customer/lib_edit.inc	
8. _knop/_mod_customer/_library/lib_customer_edit.incname	
9. _knop/_mod_customer/_library/lib_edit.inc	
10. _knop/_library/lib_customer_edit.inc	

B) -filenaming ='suffix'
example: _library/customer_edit_lib.inc

C) -filenaming='extension'
example: _library/customer_edit.lib

Knop And MVC

Knop translates to the Model-View-Controller pattern in the following way:

Model

The domain-specific representation of the information on which the application operates

- Config and Library, together with a database

View

Renders the model into a form suitable for interaction, typically a user interface element

- Content

Controller

Processes and responds to events, typically user actions, and may invoke changes on the model

- Config and Action

Why “Knop”?

“Knop” is Swedish for knot, and a knot is what keeps a lasso together. A good knot makes a good lasso experience. The meaning is the same as English knot, which is both used for the speed of boats or airplanes (one nautical mile, 1852 meters, per hour), or a knot on a rope. The speed measurement comes from the rope knot meaning, where they measured how many knots on a rope passed in a given time when they measured the speed of ships in the old days. The word stems from the Dutch word knoop with the same meaning, which is also related to knopp (knob in English).
Knop is pronounced with a sounding “k” and a long “o” just as in groove.

Knop is created and maintained by Johan Sölve, Montania System AB

Credits

Greg Willits' PageBlocks manual has been a valuable inspiration when specifying some of the components of Knop.