GongosViewOnCoreAndTiki

Hello all,

Please note the page navigation at the bottom of this page. Just in case you think this page is only a disclaimer 😀.

Disclaimer

This Page is setup as an open letter. It is not meant as a de facto standard, nor is it meant to be taken as a bit of random rambling. It will get quite long (in the end), so please try to stay with me 😉.

A lot of what is proposed here will seem a bit (too) ambitious, but I believe we can make this work and in the end wind up with a better integrated, modular product with at least the same amount of features (only more applicable to other datatypes).

There might be some content in this writing that can make it a bit harder to swallow for some ppl. I do not mean to be offensive, just summarize what was mentioned the last couple of weeks.

Oh, and one last thing ... do not start a replychain in the form of 'why not write it yourself, you lazy b*stard?'. The answer would be simply: because I do not want to form a fork and I do not want to /cannot do this on my own.

Here goes ... have fun 😉

Table of contents

...page... *Wiki page pagination has not been enabled.*

Tiki needs a rewrite

Say what?

It is a matter of taste, but in my opinion this opening up of tiki is NOT possible through small patches. It is better sometimes to take the knowledge that you have now and incorporate it in the new concept later. BTW, 'rewrite' does not necesserily mean that all is thrown away! But it is IMHO needed to at least review and reevaluate/rewrite every single line in tiki for this to work, if only to modularise everything.

This can be done feature by feature, writing unit tests along the way (the eXtreme Programming way).

Also, as Tiki claims to be a CMS/Groupware tool (which it is), I find it wrong to start from a Wiki (out of which this marvelous tool has grown) and then add as you go along. A nice framework would make extending easier as well as maintaining the different features more probable at their own pace, with minimal (if any) feature interdependency. In other words, start from a CMS view on the data and rework from there.

Arguments Pro

What I have in mind is not only going to solve a lot of issues to be found in the RFE and Bug Trackers on sourceforge, but will also enable us to do a whole lot more than is possible now and make an end to some common complaints as well as simplify the existing features without loss of functionality.

The following complaints are actually beginning to come through to the developer lists, and also by first hand user interviewing and the user list. Add to that usability issues.

1. Tiki is bloated (There are too much features that not everybody needs packaged into one big package)
2. Tiki is hard to maintain/to get familiarised with (a lot of features jump to everywhere in the code, making it a bit of spaghetti code)
3. It is a 3rd party feature unfriendly environment (you cannot just drop in your packages and when you use existing features hardcoded, you have to review them for every single release of tiki)
4. Permissions are an 'all or nothing' approach. (Either you as a user have rights to do a certain action on a feature, or not.) Some features have their own 'object permissions' implemented, but there is no common framework for them.
5. After rewrite, things might be ... no, SHOULD be more readable/maintainable. (that's a goal in itself)
6. After rewrite, the volume of tiki might go down (packagesize, ...) resulting in a cleaner environment with lesser, smaller files that is easier to maintain and extend.
7. Security issues can be handled beforehand and not like now when you want to move stuff outside of your public_html dir that you must manually either use apache rewrite rules/.htaccess to limit access to certain directories from the web or some equivalent setting in other webservers. Ie: no more hardcoded paths and only the most strictly necessary in the public document root (webaccessible) and that allow you to move everything on your system to where you want!

Contra

1. Some of us like it the way it is right now. Others might see it more as an inpenetrable jungle.
2. Every single line of code in tikiwiki needs to be redone (as in removed, reevaluated or rewritten). This might mean a cleaner environment if done properly. It might also (at first) result in minimal tiki.
3. **In short:** it will be a lot of work!

Long term effects if not done

It is still my belief that more and more users will come to the same conclusion as was posted a few times over the last couple of weeks. In phases this can be expressed as:

1. They like it
2. are overwhelmed by the shear volume of features
3. find it too hard to extend or too slow or too bloated
4. go over to a less complicated framework/CMS tool. Or they simply will stick to the version they have deployed (and that is no longer supported) because it is too much hassle to go through all setups again, changing their templates/themes again, ...

Long term benefits if done

Obviously there can be no guarantee that the number of users will increase. Neither can you say that with the 'if not done' path with absolute certainty.

However, it is my belief that if done right, and correctly, you will see an increase in active developers. It will also give you the chance to maintain documentation better and provide some unit testing to allow testing itself for conformance and functionality before any release of a package or the core framework/design.

And last but not least:

- it will make tikiwiki also more attractive for companies to invest in because it will be more stable.

To put it quite frankly, I do not think that there are all that many now that find tikiwiki to be what they need right now, but rather a loosely patched up set of tools that is hard to maintain or make it do what they want (yes, I mean the non-technical users as well).

...page... *Wiki page pagination has not been enabled.*

How? No words, but code!

Now that I have laid the ground for the rest of this tikipage/article, I get to the biggest question of all: what do I propose this new generation of tiki looks like?

Start from a CMS view on the whole

A CMS is a content managment system. 3 keywords that have a meaning that means something different to everyone you ask it to. To me, it is:

- Content = any type of information you want to store/publish to the world (be it a wiki page, an article or even a category link)
- Managment = not only writing/editing and publishing, but managing permissions, view dates, ...
- System = a framework that lets you do the 2 other keywords with ease. It facilitates that and provides a standard way to extend it with other content-types, to retrieve that information in a more standard way and provides ways to render the content as well as manage in a consistent manner those content-types.

Provide a 'core' api

What this core api will do is still under debate, but it is certain that it will at least contain abstractions for:

1. Content displayal (templates, ...)
2. DB access (adodb/pear, ...)
3. security (acl's, logging in, ldap, pear, ...)
4. User/groups managment functionality with respect to security
5. Session managment and caching/retrieval

The core provides this functionality through utility classes that are loaded always (cfr tiki-setup inclusion) and automatically. It provides no front-end to this functionality in the form of configuration screens etc. The interface should abstract away several core functionalities and act as a wrapper around the respective used technologies (smarty, adodb, ...). In this way, you create a simple interface that does not change over time (unless when the core is upgraded) and allow changes to be reflected by all extensions more rapidly without the need to go through all of the code every time.

Provide extension packages

Provide (as an extension package) a (de)installer and packager

See also

- TikiPackager
- TikiPackageRemover
- TikiInstallFeatureDev

As a package to allow easy updates of these package(s) alone.

Packages

Aside from allowing 3rd party packages to be maintained seperately and not needing them to integrate on every new version, it helps also a lot of other users/usages to become easier to handle.

As a user you can then:

- only install what you need! (features)
- install on budget hosting sites as easy as on big hosting (with lots of space) if your required featureset is small enough (current tarball is 3x MB unpacked, which is too big, considering database and compiled templates add up to that and also the sheer number of files/mysql tables will limit the number of small users/businesses IMHO and possibly even shrink the number of them, which are tiki's core audience, since big firms have their custom CMSses/applications already).
- write his/her own features using the fully documented 'tiki core api'
- package and redistribute that new package

Main packages distributed with the core package

Some packages might be bundled with the core package to allow functionality to work well:

- TikiPackage(De)Installer (see TikiInstallFeatureDev and the installation wizard, maybe both integrated)
- TikiPackager
- Main theme (css and main layouting)
- User/Group managment package
- ...

Note however that the number of these packages are reduced as much as possible to have an as stable core as possible with little to none changes in the longer run or better put: with a slower rate of development.

...page... *Wiki page pagination has not been enabled.*

Object Tree / directory / namespace / ...

Others have given their take on this far more elaborate than I intend to do here. Look for Zaufi's and Gmuslera's userpages and read on from there.

The main idea is to have all the content into one big tree. A table structure would be something in the likes of:

TikiObjectTree table

Type	**Fieldname**	**description**
int	objectTreeID	the id of the object in the tree (auto_inc)
int	fk_parentObject	foreign key to TikiObject table pointing to the parent object in the object tree
text	name	the full name in the namespace
int	fk_objectType	pointing to an existing type in the types table (eg: 2 = 'wikiContainer')
int	objectID	an id to be interpreted by the using class to retrieve the correct object from the respective table containing all the objects
int	fk_creatorGroupID	the group or singletonusergroup that owns the object
int	fk_tiki_domain	the id of the tiki domain this object belongs to

Note: Users/groups can be incorporated in this tree as well although IMHO they need to be in a different structure

Advantages

You can see that in this way you can easily have namespaces. Wiki pages are standard imported in the /Wiki container. While the pages go into the respective table(s) - in this case the tiki_pages table and others - for each page there is created a leaf on the tree object. Interlinking of pages in the same container have the same specific prefix (in this case '/Wiki/' that is prepended to the page's name.

Example :

Wiki pages to import

- TikiPackager
- TikiPackageRemover

- there exists a public Wiki container that has objectTreeID = 15 and has a name '/Wiki'
- there exists a 'wikiPageObject' object that has objectTypeID = 7
- the creator of the objects is 'SingletonUserGroupAdmin' which has id 1
- the tiki domain of this site has ID 2 for example 'docs.tiki.org'

During import the following things happen

- 2 records are created in tiki_pages and are returned to have respective objectIDs of 19 and 20
- 2 records are made in TikiObjectTree
    - insert into TikiObjectTree (fk_parentObject, name, fk_objectType, objectID, fk_creatorGroupID, fk_tiki_domain) values (15, '/Wiki/TikiPackager', 7, 19, 1, 2);
    - insert into TikiObjectTree (fk_parentObject, name, fk_objectType, objectID, fk_creatorGroupID, fk_tiki_domain) values (15, '/Wiki/TikiPackageRemover', 7, 20, 1, 2);

As you may or not see from this example, a lot more than the 2 inserts actually goes on, since this is done through the framework as well (events, triggers, checking to see if the /Wiki object exists, the usergroup exists, there are wikiPageObjects allowed in that object, general security etc ...).

You also can see that having this centralised objecttree requires minimal changes to current tables.

Finally, you can surely see that this centralised objecttree allows us to make a phpgacl like object permission system.

And last but not least, I hinted to a way to have multiple tiki's onto one database.

Aliasses or categorizing and renaming pages

Renaming

A rename of a container and other objects can simply happen through the use of renaming the values with the same prefix until the last slash (or what other separator is to be used) and can maybe happen in one operation. For instance renaming the '/Wiki/' container object to something like '/myWiki/' is simple enough. When you encounter an object in the treelist that is of the same tiki_domain with that exact name, you alter it to the new name '/Wiki' to '/myWiki'. Every object that has the name with a suffix '/' prefixed to it is renamed to the new name with a suffix of '/' before it.

Example :

3 objects are to be changed in the object tree.

- Object '/Wiki' has objectTreeID = 15
- Object '/Wiki/TikiPackager' has the generated objectTreeID of 215
- Object '/Wiki/TikiPackageRemover' has the generated objectTreeID of 218

Operation is '/Wiki' to '/myWiki' (if not specified, '/' is always prepended)

'/Wiki' has its name changed to '/myWiki' because of the first rule. This is only one object for objectspace fk_tiki_domain = 2 ('docs.tiki.org')
'/Wiki/TikiPackager' has the prefixed '/Wiki/' and is altered to '/myWiki/'+'TikiPackager' because of the second rule.
same for '/Wiki/TikiPackageRemover'

same applies to all the objects in lower nested 'directories'

This is a functionality of the core ofcourse, since no objects should concern themselves with this kind of activity and table knowledge.

Categories

With an eye on security and object level permissions it should not be possible to generate a second tree branch up to the category. Indeed, if you do not have access to '/Wiki/TikiPackager', you should not have access to the '/Features/CurrentVersion/TikiPackager' if that object points to the same wiki page. Otherwise it would be simple to categorise it as a bunch of general container objects in the object tree and have such objects for '/Features', '/Features/CurrentVersion' and a wikiPageObject named '/Features/CurrentVersion/TikiPackager'.

Thought should be put into this with security and the object permissions in mind.

A proposed something could be using the aliasses implementation beneath.

Aliasses

They are just simple versions of categories. Each object type allows an alias to be appended to it (as a leaf). So, as our example above, we could still have the tree parsing for permissions (except that nobody can change the permissions for an alias type). An alias object type works on objectTreeIDs in the TikiObjectTree as its objectID.

Lets see this with an example:

We want to categorize/create an alias to a page: '/Wiki/TikiPackager' to '/Features/CurrentVersion/TikiPackager'

As data we have:

- '/Wiki/TikiPackager' has objectTreeID = 215
- aliasObjectType is ID = 4
- creator of this alias has id 915

What is inserted is this:
insert into TikiObjectTree (fk_parentObject, name, fk_objectType, objectID, fk_creatorGroupID, fk_tiki_domain) values (215, '/Features/CurrentVersion/TikiPackager', 4, 215, 915, 2);

No rights are allowed to be set on this objectType (all rights applicable to the parent object are applicable to this object as well)!

What happens then when the user requests to open the '/Features/CurrentVersion/TikiPackager' page?

- the system determines that it points to an aliasObjectType.
- it loads the handling class
- that determines its parent object, loads the handling class and does nothing further
- it detects a wikiPageObject, loads the wikiRenderer class and calls its renderRequest() function
- the wiki page gets rendered.

What with WikiSyntax?

Wiki syntax is parsed before saving the object into the database. All references that can be resolved, are transformed to ' alternative text ' where id is the objectTreeID. If not found (does not exist), the id '-1' is used.

Using as much IDs as possible does a few things on the database:

- make it less humanreadable
- makes indexing tables far more easy/possible/efficient
- makes refactoring/renaming more efficient
- caution must be met when importing/exporting auto_inc fields and fk's (as is always the case!)

Tiki_domains?

Yes, have multiple databases with a common root object in your database and still allow for complete distinctness of the data.

...page... *Wiki page pagination has not been enabled.*

Extensions for Outputfilters/contentfilters

Outputfilters are those things you need when you finalise your content-rendering. It is the final step towards the rendering of the formatting code to the client's browser or viewer.

- Wiki syntax outputfilter
- HTML->PDF outputfilter
- HAWIKI & VoiceXML
- printerfriendly version filter
- gzipping the content if possible and enabled

Enable all the other features to use these kind of outputfilters (render to PDF for instance) with maximum ease. Just define the filter order for the objecttype/object and let the framework do its work.

Ofcourse we also need a minimal inputfiltering as well. Making sure that hazardeous or malicious content cannot get through and that is filtered out accordingly. Also other parsing might need to be done as well on the content (if it conforms to certain standards, ...) and possibly altering the content along the way on its way to the db.

...page... *Wiki page pagination has not been enabled.*

Extensions for Object type handling

JSR word for it: portlets

In my opinion (if I understood it correctly), portlets in java are small pieces of components on a website (think a loginbox, a last changes box, ... modules in TikiSpeak). As we define how every object is linked with others in the object tree, and we use a 'one-file-serves-all' approach, we do also define how we embed everything else.

A Wiki-content-object component knows how to render a wiki page for instance. An action component knows how to render an action (tab for instance). And a blog-listing knows how to display a blog listing. Or a comment knows how to display a comment.

If we have a container, or even a wiki page, we can embed more easily all the underlying or interlinking content. For instance the history tab, the edit/rename/lock actions on wiki pages, the comments/attachments tab, ...

comments/attachments/... for all

If you have a few killer apps, it would be trivial to add a subcontainer to it having a history, a bunch of comments pertaining to it, a few attachments all with little to no effort. In fact, it might even be configured by the admin user(s).

It does not matter then anymore where you do your linking to (forum, wiki, ...) for the comments lib it all boils down to the object tree.

But it does not stop there. Every subobject can have subobjects as well. Take for instance a rating feature you link on a comment that is linked to a tracker that is linked to a wikipage.

MVC design pattern

The main idea is that you abstract the coupling and only provide 'visualisation', 'actions' and 'db logic' for every type of object in your object tree.

This is done quite good in current tiki's (when it is good, we need to say it as well, so bravo), separating the model (db logic) in the lib class, the controller (actions) in the respective tiki-... .php files and the view (visualsation) in the .tpl files.

The one big problem here and there is that it is not loosely coupled enough. It is far too interdependent.

...page... *Wiki page pagination has not been enabled.*

Standard embedding possibilities

You want to embed something? Just put { EMBED objectpath } inside your document. The embedding syntax gets this objectpath and queries if the object exists for that tiki-domain, the user has rights to it to view it and if so, what handles the class.

It then loads the appropriate class to render the objecttype and calls this through the framework to generate it into a smarty variable at the current position.

The main page loaded can be handled as such as well, containing a mere { EMBED page parameter }.

As long as you have a standard way of rendering an object 'inline', you can include ANY object in any kind of object. (for instance modules like they are now can be done like this; but also an inline image shown or a tracker item, or ...)

Ofcourse any underlying objects in the object tree and all of the actions that can be done on the current object itself in its current state (if in edit mode for instance, you have preview, apply, save, cancel; in view mode, you can have history, comments, rename, (un)lock, edit/create, remove, ...) have their own representations. And certain syntaxes (wiki syntax for instance) can make a difference as well in representation and showing of subobjects (for instance a WikiLink inside a wiki page).

...page... *Wiki page pagination has not been enabled.*

WYSIWYCA (what you see is what you can access)

In the light of what was mentioned on the previous page, you need to check every link or your rights for certain actions (like wiki create page) if you can display it.

Default way to display any one link is by outputting the alternative name without a link to a tiki-object in the objecthierarchy.

If you have the view right on an embedded link, the appropriate (portlet) code would get called to display the link as type 'embedded' and the normal display output would then get generated.

In some cases it might even be possible to show only stuff you have access to and those you do not have access to get not displayed at all.

This might bring some consequences on the 'last x yyy'ed' modules performancewise though, but would allow the admin or the creator of the resource to restrict access to the page 'to selective eyes only'.

Conversely, if you do not have the right to perform an action, the action is not displayed. Code needs to check any requests on actions anyhow to see if a user has the rights to them anyhow when it receives those requests and take appropriate action if the user has/does not have the right to perform the action. Same goes on viewing a 'guessed' resourcename.

Those last 2 things should be in the framework anyhow, all tucked away into the startup/include-code for every page request.

Garnering for this would make Tikiwiki a bit more secure nonetheless.

...page... *Wiki page pagination has not been enabled.*

One file to serve them all!

All of this is served by one single php file:
tiki-index.php or something similar

Maybe it would be easier to have certain db contents fetched easier (images for instance) that just need to check if the user has the rights to that object and if so, return an image from the database after setting the correct mime-type etc....

Parameters and rewrite rules simplification

Remember the above about actions on content-object-types? It is easy to see that with the unique documentname, you can actually have a parameterlist of 2 on a GET parameterlist.

example:

tiki-index.php?page=/users/myPage&amp;action=edit

You can also simplify (for apache users) the rewrite-rules. All you need to do is just append the path to the object to the URL and find a way to encode the action parameter.

All of this is simple to sustain/implement if you keep 2 things in mind:

1. All other parameters get sent through a post request
2. All objects/subobjects are in the tree and have unique alias names anyway!

...page... *Wiki page pagination has not been enabled.*

Custom and predefined event system inside API/db

Concept

Sometimes you want behaviour to be done after an action is done on an object. A few examples:

User setup on creation

Whether the admin creates a new user, or a user registers him-/herself to your site is the same: you create a new user. You can think of users as well as objects. Now, take for instance that on one site somebody (the admin) wants that a singleton usergroup is set up with only that new user in it.

new User object: Gongo
new group object : SingletonUserGroupGongo -> add Gongo

You want that user to have a Userspace, so you add an event to the User->OnCreate eventhandlerslist.

You also want a blog and a personal calendar to be set up for this user in his/her own space, with private rights to those objects.

new User object: Gongo
Event 1 triggered :
create SingletonUserGroupGongo and add Gongo to it
Event 2 triggered :
create a new object inside /users called /users/Gongo of type 'Container' with creator = SingletonUserGroupGongo
create a /users/Gongo/myBlog object of type 'Blog' with creator = SingletonUserGroupGongo
create a /users/Gongo/myCalendar object of type 'Calendar' with creator = SingletonUserGroupGongo

Every time a user is created for that domain, these events are called in the order they had been defined.

Wiki history

The Wiki history tracking can be redefined and reimplemented as such an event. A BeforeUpdate event on 'wiki' objects can be
coupled to the creation.

Before Wiki page '/mainWiki/HomePage' gets updated (version 5)
Event 1 triggered :
move the current object to /mainWiki/HomePage/Version/6
create a new object of type WikiPage and place its id in the object tree for '/mainWiki/HomePage'
Update the object in '/mainWiki/HomePage' with the new data
Trigger the afterUpdate events

Article reading counter update

Every time an article is viewed, you can update the counter, or update a specific table.

Adding a new domain

This can also be used to setup the functionality to let the admin create a new tiki-domain.

Mail notifications

You can also place the mail notifications on these events.

Implementation concepts

Every object type not only defines what actions can be taken on them, but also on those actions(on those objects) what events can happen.

Data in that table can consist of a few fields:

Eventtype table

Type	Fieldname	description
int	action_event	the id of the eventtype (auto_inc)
int	fk_action	foreign key to action table
bool	is_beforeEvent	is it a type that occurs before that action?
bool	is_afterEvent	is it a type that occurs after that action?
text	documentation	additional information for the event happens here. How the class is supposed to look like, what its parameterlist looks like etc
text	script	either the phpfile containing the content (with classname/instance and method), or the content itself
int	order	the order in which it happens if there are more than one
text	name	the name of the action (in addition to the description)
bool	active	if the event is active

For each instance of an objecttype you can define what file contains what class to be invoked with respect to this event. This is ofcourse stored in a seperate table that combines the objecttree objects with the eventtypes and what classes need to get invoked/created in what order and if the event is enabled.

It is possible to set this up for every object in the object hierarchy, or it is inheritable like permissions from the main root.

Retrieval of the events and their order for a certain object and action needs to be incorporated in the core as well. Invocation and instantiation of objects is the responsibility (as well as the timing for them) of the objecttype handling class.

Other ways

The attentive reader would also note that this would open up a lot more than just this type of dynamic control. In fact, it would be trivial to rewrite for instance the History type of plugin to be a beforeEdit and afterAdd eventhandler. It would then be trivial to extend the feature of a history not only to wiki page objects, but to ANY kind of object you'd like (written and to-be-written).

In fact, all it would do is this:

- in the object tree, it would make a subcontainer of type 'history container' if it did not exist already right under the object itself we are about to edit (or add it afterwards if 'add/insert')
- it would make a node inside it with the version and history container as name, and as parent the history container object
- it would copy the current record (type, object id, ...)
- change the objectID to null in the record itself (meaning that an insert in the object's table will take place and the received id gets placed in this)

Et voila, all you need now is a feature that handles the null=create new record to make it handle a history.

This is just one example of a wiki feature as a package that depends on nothing (anymore) and that can be reused at nauseam.

Addendum

You could let this all get done dynamically. Making a script that gets included as a function into a class object. Invocation of the $tiki_coreWrapper->doBeforeActions($object, $objectID) would then load all the scriptlets from the database (in order), make php functions out of them and run them in order. The $tiki_coreWrapper object would be accessible to all of those functions.

Likewise would an invocation of $tiki_coreWrapper->doAfterActions($object, $objectID) do the after actions on it inside the same transaction (for those databases that support transactions).

An editor for the script should be provided like the one for galaxia (maybe even through galaxia).

...page... *Wiki page pagination has not been enabled.*

Initialisation phase and hardcoding paths belongs to the past

---

An idea brought forth by zaufi is this:
Have an init dir, where all kind of small installation initialisation scripts reside, setting up and destroying all tiki needs to get going.

For instance, load your db abstraction layer class files, smarty etc here.

It mainly stems from the reversing of the idiom (as stated in a previous page 'one file to serve them all') the way a request is handled. Before you enter the displayal and business logic and data retrieval code, a lot is already initialised. You no longer need to define what tiki-setup or tiki-setup_base or other php files to include to do the initialisation, since it is in fact that code that will call your class (the other way around).

Another aspect linked to this is the fact that all the includes can be centralised into a few directories, that you provide certain 'environment variables' that point to those directories and then include/require(_once) them like so:

```
include_once('$tiki_libdir/myfeature/mylib.php');
```

That way you do not need to have your lib dir in a certain place, where you cannot move it from. Ie: move the lib dir out of the public webspace area.

...page... *Wiki page pagination has not been enabled.*

Installer module

See also

- TikiPackager
- TikiPackageRemover
- TikiInstallFeatureDev

It is easily seen that with the advent of both database abstraction and modularisation (cutting tiki into smaller independent features), it would become easier to implement a feature/module/extension to manage all the upgrades, installs, deinstalls.

Features might get updated more/faster as bugs are fixed in them and do not need to wait until everything else becomes stable again until a release.

It might even provide developers with a scriptable way to exchange small database changes with eachother for features.

Requirements

1. Database abstraction layer
2. Scripting abstraction layer that enables version changes between scripts and current db to be detected. and be processed on an incremental way, completely database independent. (xmlschema is a big leap forward for this to happen, but lacks 3 crucial things for the moment: conditionals like <... platform="sql3" feature="wiki" version="<= 2.0">; quoting for db inclusion ; a diff functionality in adodb that enables you to alter a table rather than have to run one of a few different scripts and maintain them all (update script, migration script, create script, ...)
3. packager itself
4. a way to get to non-hardcoded paths for inclusion.

...page... *Wiki page pagination has not been enabled.*

Multisite Tiki's on one database

As I hinted before, this might be accomplished as well.

One installation

If you install only one db, you need to install one codebase also. it is that simple. Placing a lot of dynamic logic into the database, it is necessary that all tiki sites get updated at the same time. I mean 'class logic' and 'database structure' are so intricately intertwined that they need to be updated at the same time. It goes without saying that installing a package that alters a database structure and installs some files to handle the new structure needs to be executed either for all databases at once, or for one at a time.

Therefore, if you only want to serve from one database, you serve from one codebase as well to eliminate out of sync updates.

Conversely, if you host from one codebase, you either have to do the setup of the databases seperately or host from one db as well.

It goes without saying that it is always a bad idea to have more than one codebase going to the same database (even with current incarnations of tikiwiki).

DB provisions

- It should be possible to have a seperate userbase on all the databases, or a (mainly) overlapping one on all sites you host on that database.
- it should be possible to share resources, including all data linked to it from the object tree between any number of sites
- rights should be maintained on those resources

The resourcetrees could all start from a common root. Every tiki domain could then have an object with this root object as its parent. The virtual root object of every tiki-domain is then this subobject of the common root.

Shared content

Some things are shared content. The general admin content, all the options under it, general layout etc could then move to a /manager object on which a mega-super-user (admin) only has access to. It contains all the default options. /$tikidomain/manager conversely allows admin users of the respective domain to override the default options.

Btw, did I mention that these configurations get pulled out of the database and displayed dynamically iso the way they are handled now?

All the stuff in the application menu (also dynamically loaded with wysiwyca) under the "Admin (Click!!)" item is also on a per-site basis, accessible to respective administrator users other then 'admin'.

...page... *Wiki page pagination has not been enabled.*

Migration issues

Then, what about current existing Tikiwiki sites? How would they be migrated to this new Tiki?

The answer is easy and not so easy and knows a few paths of thought:

1. Export feature on all features (error-prone, you might forget some) and import those feature's data on your new tiki with the appropriate modules/plugins/extension installed in predefined and reserved paths on / (for instance /Wiki/HomePage etc...)
2. New tiki can be installed in same database, and uses other tables. While it is highly likely that these tables change and evolve, it would bloat the database even more and some features would need to be restored later on.
3. Multiple databases and import from the old one the features you need directly. Version differences make this rather almost impossible. (only 1.8/1.9 is supported?) an 'import feature from old Tiki' would then be needed for every feature.
4. One can suspect that the features that were used before are going to be needed and installed on the new tiki. Installing all those features and maintaining them might present a far greater maintenance cost for the admin, although this is not necessarily so. Also, as mentioned before, table structures might change, tables might merge, others might split.
5. ...

...page... *Wiki page pagination has not been enabled.*

Performance issues

With the wysiwyca etc, it is clear that a lot of db access needs to be done. Especially if group-inclusions are found in a tree like hierarchy.

It is therefore in some cases handy to allow caching to occur on the moment you log in. It is then determined to what groups your user belongs in what level. If you belong to a group only after 2 linkings that allows you to do an action on a resource, but you also belong to a group after 1 (direct) linking that disallows that action, you need to cache some things if not all, maybe even in the session.

- userid
- username

- usergroupid[][][][][]... level id's cached as deep as those where the user is in.

This would speed up the acl checking significantly. You only need to launch a 'where in (' usergroupid level x ids ')' on the resource itself until you find a permission for a resource.

There are other less obvious caching items, but they all have one thing in common: fetched once per session, and valid for the duration of the session.

...page... *Wiki page pagination has not been enabled.*

Conclusion

I have tried to explain a possible future roadmap for Tiki. I also tried to explain some principles of a CMS view on the data we want to keep in our tiki's as well as related security issues that can be handled from the ground up.

I realise that this is a very ambitious take on the whole and that I probably will not make myself very popular posting this to some, but I also know that there are a lot of developers (tiki and external) that agree (even to a certain degree) with what I wrote above.

I hope I've convinced some of you of the necessity of this rewrite (these ambitions are not realisable with small patches, or you'd end with another abomination, a bit like Frankenstein). I also hope that I have given hope to some that these changes would in fact make it easier for developers (both tiki and 3rd party) to develop stuff for it. Far more easier than it is now.

So, maybe it is time for a poll ...

Sincerely,

Dimitri Smits
aka Gongo
