Permission Revamp

Permission Cleanup has been merged into trunk for release in Tiki 4.0, as part of the Workspace. Note that this page is being included dynamically from the Hello World wiki page.

It was previously in branches/experimental/perms-take2. Implementation details can be found in the code, especially in Perms.php

The objective of the clean-up is to provide an homogeneous interface to access permissions in a way that is simple and efficient. The interface used should not reflect how the permissions are stored. The previous interfaces used multiple functions across different libraries that were confusing and caused frequent WYSIWYCA problems in addition to being inefficient when filtering lists of objects.

The permissions used on categories were confusing and lacked the customizability expected in TikiWiki. The new layer uses exactly the same permissions on objects, categories and global permissions.

The redesign offers fully tested code with guaranteed behaviors.

From old to new

Validate a global permission

Previous code

```
<?php
// $tiki_p_* variables made available in tiki-setup.php

if( $tiki_p_edit_article == 'y' ) {
    // ...
}
?>
```

Replacement

```
<?php
// Available globally, by tiki-setup.php
$globalperms = Perms::get();

if( $globalperms->edit_article ) {
    // ...
}
?>
```

The previous method caused a lot of pollution of the global scope and forced the usage of multiple global variables inside functions. The new API allows to obtain a permission accessor from anywhere. The global permissions obtained will be the true global permissions, not those that may be overridden by other calls in the page.

Validate an object permission

Previous code

```
<?php
```

```
// $tikilib and $user defined in tiki-setup.php

if( $tikilib->user_has_perm_on_object( $user, 'HomePage', 'wiki page', 'tiki_p_view',
'tiki_p_view_categorized' ) ) {
    // ...
}
?>
```

Replacement

```
<?php
$objectperms = Perms::get( array( 'type' => 'wiki page', 'object' => 'HomePage' ) );

if( $objectperms->view ) {
    // ...
}
?>
```

Obtaining permissions on an object now follows the exact same pattern as obtaining global permissions. No need to remember which function had to be called and on which library it was located, what is the order of the parameters and which permission applies if it has to be resolved through categories.

Because the global permissions are loaded at the beginning of the script, verifying permissions on an object will take at most 3 additional queries.

- One query to verify object permission
- On miss, one query to obtain the categories on the object
- One query to obtain permissions on those categories, if applicable and not previously encountered.

The previous amount of queries qualifies as *hard to tell* due to caching schemes and complex control flows. However, it was likely to be up to 5 in normal cases, plus checks on parent categories.

Filter a list of objects

This code is not exactly accurate because the actual filter code was moved inside list_pages() between 2.0 and 3.0. However, the concept remains and the current code within list_pages() will be updated to benefit from the new API.

Previous code

```
<?php
$pages = $tikilib->list_pages();
$filtered = array();
foreach( $pages as $page ) {
    if( $tikilib->user_has_perm_on_object( $user, $page['pageName'], 'wiki page',
'tiki_p_view', 'tiki_p_view_categorized' ) ) {
        $filtered[] = $page;
    }
}
```

```
?>
```

Replacement

```
<?php
$pages = $tikilib->list_pages();
$filtered = Perms::filter( array( 'type' => 'wiki page' ), 'object', $pages, array(
'object' => 'pageName' ), 'view' );
?>
```

The previous code had to verify the permissions for each object individually. Considering an average of 3 queries per object, which is likely to be on the lower end of the spectrum, filtering a list of 30 objects would have required 90 queries.

The replacement benefits from bulk loading, also accessible through Perms::bulk() if the objective is not list filtering, and will filter the entire list in 3 queries or less, just like it would for a single object.

Key features

- Handling of lists in a constant amount of queries
- Re-use of loaded rules when possible
- Less reliance on global variables
- Removal of redundant tiki_p_ permission prefix
- Single point of definition of rules
- Extensible rule system hidden behind a facade allowing to update rules without affecting the rest of the code.
- Use different rules in unit tests to validate code without affecting the database.

Practical usage

List of object types

From a look at lib/tikilib.php, it seems that the available object types are:

- 'wiki page'
- 'tracker'
- 'blog'
- 'map'
- 'forum'
- 'file gallery'
- 'image gallery'
- 'topic'
- 'calendar'
- 'comments'
- 'map_changed'
- 'category_changed'

Extensibility

- Composable rules. Add or remove resolution rules from configuration. For example, removing a single line can disable object permissions. They can even be changed at runtime.
- Creation of new rules to lookup permissions differently. Global, Category and Object fit in this layer.
- Smarter resolvers. Want to introduce dependencies in permissions?
- Fallback resolvers to handle special cases, like creator permissions, admin permissions.
    - Direct : Default check on exact permission
    - Indirect : Check an alternate, like parent, permission instead
    - Creator : Check an alternate permission when the active user is the creator of the object (must be included in the context)
    - *More?*

Because Resolver is an interface and verifying permissions only rely on it, everything can be changed without affecting the rest of the application, as long as permission names remain the same. However, a compatibility layer can be added even to that.

ResolverFactories generate the resolvers. These look up in the data source. More can be created and composed in different ways.

Set-up

```
<?php
$groups = array( ... );

$perms = new Perms;
$perms->setGroups( $groups );
$perms->setResolverFactories( array(
    new Perms_ResolverFactory_ObjectFactory,
    new Perms_ResolverFactory_CategoryFactory,
    new Perms_ResolverFactory_GlobalFactory,
) );
$perms->setCheckSequence( array(
    new Perms_Check_Direct,
    new Perms_Check_Indirect( array(
        'view' => 'wiki_admin',
        // ...
    ) ),
    new Perms_Check_Creator( $user ),
) );

// Activate
Perms::set( $perms );
?>
```

Additional considerations

Related to these changes are the changes to categories. To remain efficient, category lookup on objects has to be limited to one query and so does the permission lookup on categories. The current adjacent node

model typically requires multiple queries to dig down the hierarchy. At this time, assigning permissions on a root category can optionally copy the permissions down to every node. The maintenance cost is a little higher, but it allows for faster lookup fitting the performance requirements.

For this case, which is the current implementation, only the direct categories are considered in the category lookup.

An alternative keeping permissions on the top level nodes only would be the nested set model.

Common problems

Mix up between global and local permissions. In multiple files, the object permissions overwrite the global permission variables. Effectively, this ignored object-level permissions. This can work correctly as part of a transition. However, the override must be done **before** any permissions are checked. TikiLib::get_perm_object($id, $type) is the recommended method to override global variables.

Custom code to validate permissions. Most of it must have been removed by now to be replaced with get_perm_object, but a lot of copy-pasted code contained logic to verify permissions that was outdated in many cases. Any chunk of code containing object or category permission logic should be removed from tiki-*.php should be replaced with the appropriate calls.

Listings require global permissions in many cases. The correct behavior would be to allow if any object can be listed. Because of the permissions on object and categories, this may be hard to determine. At this time, the permissions should be granted globally and restricted locally by adding category permissions. In the future, as part of the improved listing filtering, it should be possible to solve this one correctly. However, this is not expected before 5.0.

Troubleshooting

Because of the dynamic nature of the code, attempting to print values randomly from within the objects will not be very helpful. The behavior of the code is based on **object composition** rather than procedural code. The libraries must be treated as black boxes. Their correctness is demonstrated by the extensive unit test suite.

An important thing to know is that once an accessor is built, through Perms::get(...), it contains all the information it needs to resolve permissions for any set of groups that may be provided to it. They are also completely independent. Modifying their state will not affect the rest of the system. Thus, a simple **print_r** or **var_dump** on the accessor will show all there is to know.

Here is a simple output and how to interpret it.

Sample print_r of an accessor

```
Perms_Accessor Object
(
    [resolver:private] => Perms_Resolver_Static Object
        (
            [known:private] => Array
                (
                    [Anonymous] => Array
                        (
```

```
                        [view] => 1
                        [forum_read] => 1
                        [forum_post] => 1
                        [forum_post_topic] => 1
                        [post_comments] => 1
                        [read_comments] => 1
                        [wiki_view_comments] => 1
                    )

                [Admins] => Array
                    (
                        [admin] => 1
                    )

            )

    )

[prefix:private] => tiki_p_
[context:private] => Array
    (
    )

[groups:private] => Array
    (
        [0] => Admins
        [1] => Registered
    )

[checkSequence:private] => Array
    (
        [0] => Perms_Check_Alternate Object
            (
                [permission:private] => admin
                [resolver:private] => Perms_Resolver_Static Object
                    (
                        [known:private] => Array
                            (
                                [Anonymous] => Array
                                    (
                                        [view] => 1
                                        [forum_read] => 1
                                        [forum_post] => 1
                                        [forum_post_topic] => 1
                                        [post_comments] => 1
                                        [read_comments] => 1
                                        [wiki_view_comments] => 1
                                    )

                                [Admins] => Array
                                    (
                                        [admin] => 1
                                    )
```

```
                                )

                        )

                )

            [1] => Perms_Check_Direct Object
                (
                )

            [2] => Perms_Check_Indirect Object
                (
                    [map:private] => Array
                        (
                            [ws_view] => ws_admin
                            [ws_removews] => ws_admin
                            [ws_adminws] => ws_admin
                            // ... removed for readability ...
                            [admin_integrator] => admin
                            [admin_dynamic] => admin
                            [admin_banning] => admin
                            [admin_banners] => admin
                            [add_object] => admin_categories
                            [access_closed_site] => admin
                        )

                )

            [3] => Perms_Check_Creator Object
                (
                    [user:private] => admin
                    [key:private] => creator
                    [suffix:private] => _own
                )

        )

)
```

At the very top, the **resolvers** property contains the set of rules that apply for the requested object. In this case, global permissions were obtained. However, there are no differences with object or category permissions. The ResolverFactories always obtain the permissions and provide them as a static resolver. The content is simply a map between groups and the permissions that are granted to them.

The prefix is an optional **prefix** that can be used when checking permissions. $perms->view or $perms->tiki_p_view will provide the exact same permission. This was used for backward compatibility.

The **context** is the array that was used to obtain the permissions. In this case, the context is empty because it was fetched for the global scope. However, the object may also contain an object type, id and creator. Other properties could be used in the future if needed. The context is essentially used by the resolver factories to identify which one applies and where to obtain the information from. It is kept in the accessor as a reference, but can also be used by the check rules.

The **groups** property simply contains the list of groups currently being verified. These can be changed using setGroups(). By default, they are set to the groups of the current user by Perms::get() for convenience. Except for a few rare cases, permissions are checked for the currently active session.

The **check sequence** determines how the permissions will be verified. As the name indicates, they are verified sequentially. The verification stops as soon as one of them provides a positive response. In this sequence, there are 4 steps:

- The alternate check first verifies if the groups verified have the global admin privilege. To perform this verification, the check contains a reference to the global resolver. This check does not use the context's resolver at all. By being the first check made, it insures that the global admins will always be allowed to perform any action.
- The direct check verifies if the permission is granted directly by the resolver.
- The indirect check in this case is configured to verify the admin privilege related to the privilege currently validated. A user with admin_wiki will be granted all rights related to the wiki feature. These rules are loaded from the users_permissions table.
- The creator check is not widely used at this time, but would allow for a standardized _own permission naming convention. It verifies an alternate permission if the current user matches the creator in the context.

Responsibilities

The Perms component was designed separate the concerns of the user land and the internal rules. As part of normal usage, one only needs to know of Perms::get() to obtain the set of rules they should use and of the property accessing technique to verify individual permissions. More advanced usage may include list filtering, as mentioned earlier.

The accessors provided by Perms::get() are meant to be self-contained.

Typical usage

```
<?php
$perms = Perms::get( array( 'type' => 'wiki page', 'object' => $page ) );

if( isset($_POST['save']) && $perms->edit ) {
    // Do stuff
}
```

The Perms class acts mainly as a facade to the subsystem. It holds the rules and the default values to be assigned in order to keep the code simple in the rest of the system. It's primary purpose is to build accessors. Most of the values required, like the default groups to assign and the check sequence to be performed, are provided as configurations. However, the central component in the accessor is the resolver. The resolvers are independent rules that apply for a given object, or context. These resolvers are obtained through the resolver factories.

When queried, the resolver factories inspect the context and identify if they need to make a verification on it. Each factory verifies a very specific set of rules. As they are validated in sequence, the first one to provide a set of rules will be the ones that are used. In a typical installation, the sequence is the following:

- object to verify if permissions were assigned directly on the context object,

- category to verify if any of the categories in which the object is in contain any permission,
- global as a fallback.

The resolvers are also responsible of the efficient fetching of their resources, especially when requested in bulk. As an example, the category factory first fetches the list of all categories that apply to the requested objects. It then verifies which of those categories it has not yet fetched the permissions for and fetch the permissions for all of those categories in a single query and index them. Afterwards, it gathers the permissions from it's internal cache to build the list of the individual objects.

Alias

Alias names for this page:
PermissionCleanup | PermissionRevamp | Permission Cleanup | Permissions Cleanup